THÈSE

présentée par

Cécile CAPPONI

pour obtenir le titre de

Docteur de l'Université Joseph Fourier

(Arrétés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité

Informatique

Identification et exploitation des types dans un modèle de connaissances à objets

Soutenue le 19 octobre 1995 devant le jury composé de :

M.	Jean-Pierre	VERJUS	Président
M.	Michel	HABIB	Rapporteur
M.	Jean-François	PERROT	Rapporteur
M.	Hassan	AÏT-KACI	Examinateur
M.	Philippe	JORRAND	Examinateur
M.	Laurent	TRILLING	Examinateur
M.	François	RECHENMANN	Directeur

Thèse préparée au sein du laboratoire LIFIA/IMAG

Je remercie...

toutes celles et tous ceux qui m'ont apporté leur confiance, leur soutien ou leur sourire, et qui m'ont consacré un peu de temps... Parmi ces personnes, je tiens à remercier plus particulièrement :

Monsieur Jean-Pierre Verjus, Professeur à l'Institut National Polytechnique de Grenoble, Directeur de l'INRIA UR Rhône-Alpes et Directeur de l'IMAG ; qu'il sache que je suis très sensible à l'honneur qu'il m'a fait en présidant mon jury de thèse.

Monsieur Laurent Trilling, Professeur à l'Université Joseph Fourier de Grenoble, d'être présent dans mon jury de thèse. Je lui suis très reconnaissante de porter ainsi de l'intérêt à mon travail.

Monsieur Philippe Jorrand, Directeur de Recherche CNRS et Directeur du LIFIA, de m'avoir acceptée au sein de son laboratoire, et d'avoir accepté de participer au jury de ma thèse.

Monsieur Hassan Aït-Kaci, Senior Chair of Intelligent Software Group, Simon Fraser University, d'avoir accepté sans hésitation de faire ce grand voyage pour prendre part au jury de ma thèse. Je le remercie également pour m'avoir fait confiance et acceptée en post-doc dans son équipe à l'issue de ma thèse. Je considère cette année à venir comme un privilège.

Monsieur Jean-François Perrot, Professeur à l'Université Pierre et Marie Curie de Paris, d'avoir immédiatement accepté d'être rapporteur de ma thèse, et de m'avoir reçue à Paris pour une discussion très enrichissante. Je le remercie sincèrement d'avoir porté de l'intérêt à mon travail, de m'avoir maintes fois encouragée et ainsi réconfortée.

Monsieur Michel Habib, Professeur à l'université II de Montpellier, d'avoir accepté d'être rapporteur de ma thèse, et de m'avoir conseillée très justement sur des questions critiques. Je le remercie également pour sa disponibilité.

Monsieur François Rechenmann, Directeur de Recherche à l'INRIA, de m'avoir acceptée dans son équipe et si bien dirigée dans mes recherches. Quelques louches de liberté de penser, quelques autres de conseils avertis, un grand verre de disponibilité, et un bol de confiance, le tout dans une casserole de rigueur scientifique, ont été les ingrédients principaux de sa recette pour mon encadrement ; je voudrais ici lui dire que le cocktail était à mon goût. Je n'oublierai pas de sitôt tout ce qu'il a fait pour que je parvienne sans peine à mes fins.

Patrice Uvietta, capitaine centimaître, pour avoir toujours été à l'écoute, et pour avoir eu la patience de lire et corriger les premières versions de mes écrits. Si quelqu'un doit un jour m'apprendre à garder mon calme, c'est bien toi.

Florence Lemaire, reine des surprises, pour avoir toujours été disponible avec cette générosité naturelle qui te caractérise. Les "calmants" ne sont rien à côté de tout ce que tu as bien voulu partager avec moi pendant ces trois ans. J'espère simplement que cela ne va pas s'arrêter là.

Jérôme Gensel (bientôt Giant) et **Pierre Girard**, les deux larrons de SHERPA, pour tous ces bons moments passés ensemble autour d'un café ou d'autre chose, même si je regrette qu'il y ait eu trop souvent TROPES à côté du café. Merci pour votre soutien quotidien, tant moral que scientifique, il a été plus qu'important.

Les autres membres de l'équipe SHERPA qui ont tous contribué à l'aboutissement de cette thèse : **Nina** (chapeau pour ta patience à mon égard), **Jérôme** (merci pour tes conseils et ta confiance), **Jean-Yves** (l'ouverture d'esprit incarnée), **Gilles** (goûts et couleurs, ça se discute !), **Danièle** (l'infatiguable), **Petko** (prince des théories), mais aussi **Jutta, Pierre, Sueli** (notre mère à tous), **Ysabelle, Jean-Marc, Katia** et **Françoise**. Je n'oublie pas tous ceux qui sont déjà partis, mais qui ont grandement contribué à rendre cette équipe très sympathique. Je pense en particulier à l'inévitable **Olga** (oui, j'irai en Colombie), **Mathias, Bruno, Olivier, Alain** et **Nathalie**. Je voudrais dire aussi un énorme merci à **Katie**, qui a fleuri mon bureau sans se douter qu'elle a réussi, ce jour là, à mettre des couleurs dans une tête très assombrie.

Amalia. J'ai vécu cette thèse au même rythme que toutes nos années d'études partagées depuis si longtemps : tu as supporté mes caprices et mes joies avec toujours autant de tact et de spontanéité (le mot est dit !). Je te remercie pour ton amitié et pour ton sens du partage. Je te dédicace ces remerciements, et aussi le chapitre 2 que tu m'as aidé à percevoir.

Alejandro. Parce que ton amitié m'a permis d'aller toujours de l'avant avec le sourire. Je te remercie d'avoir été Starman, guapo caballero, danseur de rock et de salsa, poète, chanteur de Brel et tout le reste. T'avoir eu près de moi pendant cette dernière année m'a donné l'impression d'être entourée de mille personnes à la vitalité contagieuse...

Isabelle, pleine de vie, d'affection et de perspicacité. **Michelle**, l'executive woman au cœur tendre et au sourire réconfortant. **Richard**, l'amoureux des sciences en délire. Pour avoir été vous-mêmes et près de moi du début à la fin, je vous remercie. Merci à tous mes amis non cités ici, à ceux qui sont loin et ceux qui sont près, car ils ont tous contribué à faire de ma thèse une épreuve moins difficile que de coutume. Merci aux joueurs de l'AGBS, et en particulier à **Aina**, parce que rien de tel qu'un sport en équipe pour se défouler et oublier les difficultés d'une thèse.

Benoît, mon frère, mon ami, une épaule solide et tendre sur laquelle je me suis souvent laissée aller.

Bernard et **Gil**, mes parents. Toujours là, confiants, affectueux et disponibles, bien plus que le devoir ne l'exige... En ce 19 octobre, je te souhaite un bon anniversaire, maman.

*Cette thèse est aussi celle de tous ceux qui m'ont entraînée vers la recherche, et qui m'ont appris à aimer ce métier où la curiosité n'est pas un vilain défaut. Avec une pensée toute particulière pour **Marc** et **Alexandre**.*

≪ Imaginez maintenant l'usage suivant du langage : J'envoie quelqu'un faire des achats. Je lui donne un billet sur lequel se trouvent les signes : cinq pommes rouges. Il porte le bulletin au fournisseur ; celui-ci ouvre un tiroir sur lequel se trouve le signe "pommes" : puis il cherche sur un tableau le mot "rouge" et le trouve vis-à-vis d'un modèle de couleur : à présent il énonce la série des nombres cardinaux – je suppose qu'il les sait par cœur – jusqu'au mot "cinq" et à chaque mot numéral il prend une pomme dans le tiroir, qui a la couleur du modèle. – C'est ainsi et de façon analogue que l'on opère avec des mots. – Mais comment sait-il où il doit vérifier le mot "rouge" et ce qu'il lui faut faire du mot "cinq" ? – Eh bien, je suppose qu'il agit de la façon que j'ai décrite. Il y a une limite même aux explications. – Mais quelle est la signification du mot "cinq" ? – Il n'en était pas question ici, sinon de savoir comment on se sert du mot "cinq".≫

Ludwig Wittgenstein, *Investigations philosophiques*, traduction de Pierre Klossowski, Gallimard, "Bibliothèque des idées", Paris, 1961.

Table des matières

Introduction

Cette étude se situe dans le cadre de la *représentation des connaissances par objets*, et de ce fait, elle évolue au carrefour des approches à objets classiques, et des systèmes de modélisation et de déduction. En pratique, les modèles de connaissances à objets sont développés pour atteindre simultanément deux objectifs :

– permettre la représentation structurée et fidèle des connaissances issues d'une modélisation du monde,

– favoriser l'exploitation homogène des connaissances représentées afin d'en inférer de nouvelles.

Le premier de ces objectifs est atteint grâce au développement de langages de représentation dont la composante descriptive de base est l'*objet*, les objets d'un même monde étant répartis au sein de *classes* d'objets et l'ensemble des classes étant muni d'un ordre partiel appelé la *spécialisation*. En ce sens, les modèles de connaissances à objets encouragent une modélisation du monde orientée vers l'identification de ses individus, qui sont reconnus pour appartenir à des groupes hiérarchisés : `Matisse` est un `peintre`, tout `peintre` est un `artiste`, de même que tout `musicien`, et tout `artiste` fait partie de la catégorie des `êtres-humains` qui sont eux mêmes des `êtres-vivants`, etc. Un modèle de connaissances est donc dédié à la représentation d'un monde modélisé (le domaine d'application).

Le second de ces objectifs est réalisé par le développement de mécanismes ayant tous un comportement homogène au regard des objets de représentation, et dont l'activation et le contrôle sont effectués par le système de représentation lui-même. Parmi ces mécanismes, la *classification* a pour rôle la recherche de la classe la plus spécifique susceptible d'accueillir un objet donné (il s'agit ici d'un algorithme qui a pour rôle d'enrichir la connaissance portée sur un individu du monde).

Problématique

Nous traitons de la différence fondamentale qui existe, au sein des systèmes de représentation, entre ce que les concepteurs cherchent à caractériser au travers des structures qu'ils définissent, et la façon dont le système interprète ces structures. Cette distinction est nécessaire à la validation de l'adéquation entre la "réalité" et sa représentation. Nous montrons que tout système de représentation des connaissances devrait d'ailleurs guider les concepteurs vers la minimisation de cette différence.

Structures de représentation : description et dénotation

Le langage de représentation d'un modèle de connaissances à objets offre des structures destinées à accueillir la description d'éléments d'un monde modélisé. Ces structures sont des *objets de représentation*. D'un point de vue structure de donnée, un objet (une instance) est un doublet :

$$\text{Objet} = \langle \text{ identificateur} \,;\, \text{description } \rangle$$

où l'identificateur est une référence à l'élément du monde représenté par cet objet, et la description de l'objet correspond à un ensemble de propriétés de l'élément. L'élément représenté par l'objet est unique dans le monde modélisé (ce qui est informatiquement traduit par l'unicité de l'identificateur) ; on appelle cet élément la *dénotation de l'objet*, matérialisée par l'existence de l'identificateur. Sur la figure 0.1, l'objet du système comporte la description de la Lune, qui est un élément unique du monde modélisé (l'astronomie). Nous partons de l'hypothèse suivante : un des buts de la représentation des connaissances est que la description contenue dans l'objet soit une *caractérisation* de l'élément dénoté, vis-à-vis des autres éléments du même monde. Autrement dit, la description contenue dans un objet *devrait* véhiculer l'unicité de la dénotation de cet objet. En conséquence, le comportement du modèle, face à la confrontation de deux objets différents contenant des descriptions égales, doit être de mettre en évidence la contradiction sous-jacente : si ces objets contiennent la même description, cela signifie qu'ils dénotent la même chose ou que ces descriptions ne sont pas suffisamment complètes pour caractériser les éléments dénotés.

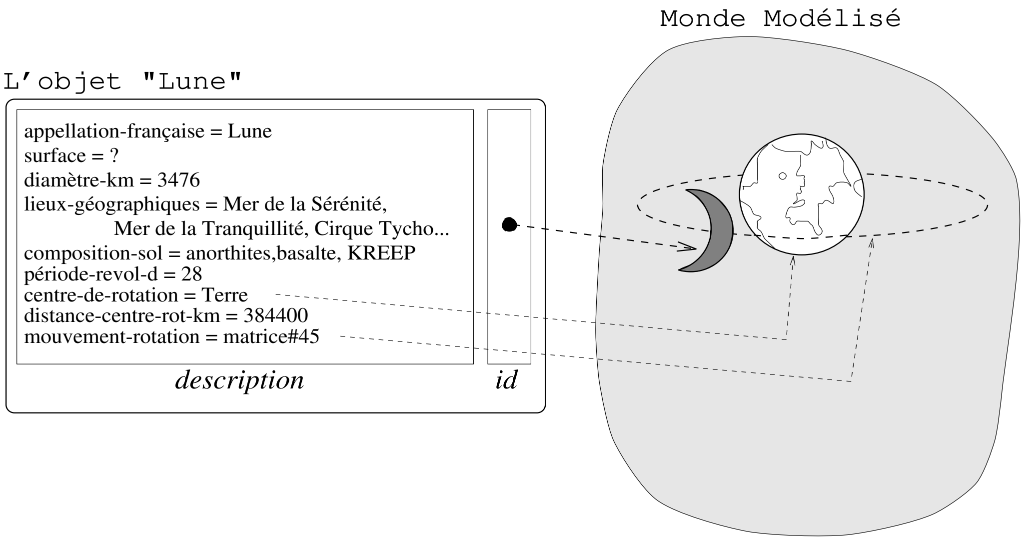


FIG. 0.1 - : La Lune est décrite, en termes du langage de représentation, au sein d'un objet. La partie *id* (identificateur) de l'objet scelle l'interprétation de l'objet `Lune` dans le monde modélisé.

Comme l'illustre la figure 0.1, la description d'un objet se compose de relations avec d'autres objets (ex. la description de la `Lune` fait référence à la `Terre`, par le biais de la propriété `centre de rotation`), et de propriétés représentées par des valeurs. Ces valeurs se distinguent des objets de représentation par le fait qu'elles n'ont pas de signification dans le monde modélisé, lorsqu'elles sont considérées seules. Dans l'exemple, c'est en qualité de `diamètre en km` de la `Lune` que `3476` acquiert une signification, `3476` pouvant aussi être le nombre de cratères de la `Lune` qui mesurent plus de 10 km de diamètre. À l'opposé, la `Terre` a une signification (une dénotation) même si l'on oublie son rôle de `centre de rotation` de la `Lune` : ce n'est pas sa liaison avec la `Lune` qui lui donne une existence dans le monde. On relève ainsi deux catégories de relations dans une description d'objet.

Dans le modèle, les objets appartiennent à des classes. Une classe est aussi une structure qui comporte une description. Le rôle de la classe en représentation des connaissances est de regrouper un ensemble d'éléments du monde modélisé. Cet ensemble d'éléments qu'une classe cherche à regrouper constitue son *extension*. Malheureusement, dans la pratique, cette extension est rarement finie, ni même figée, et, de toutes façons, n'est pas toujours connue. De ce fait, le regroupement des éléments passe par une description des raisons de ce regroupement, que l'on appelle l'*intension* de la classe. L'intension d'une classe est l'ensemble des propriétés communes des éléments que cherche à regrouper la classe (étymologiquement, il s'agit donc d'un type). Ces éléments sont matérialisés, dans une base de connaissances, par les objets (figure 0.2). Et naturellement, la description d'un objet doit satisfaire celle de la classe à laquelle il appartient. Ainsi, similairement à un objet, la classe de représentation est un doublet :

$$\text{Classe} = \langle\ \{\text{identificateurs}\}\ ;\ \text{description}\ \rangle$$

Où {identificateurs} est l'ensemble des identificateurs des objets appartenant à la classe, cet ensemble étant ainsi le reflet de l'extension de la classe ; la description est l'intension de cette classe.

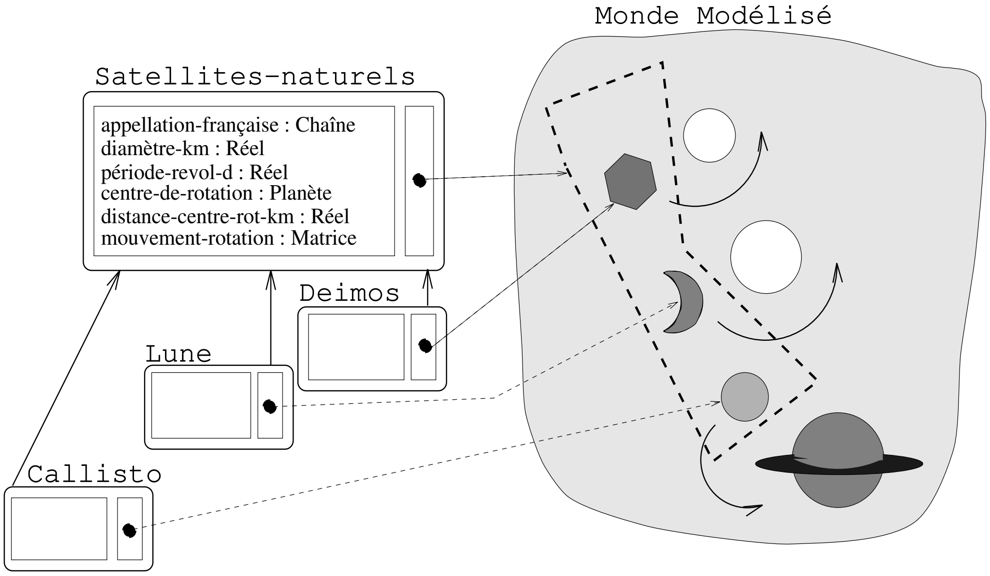


FIG. 0.2 - : La classe des `Satellites-naturels` regroupe des objets qui ont une signification dans le monde modélisé. La classe contient une description des caractéristiques pertinentes pour ce regroupement.

Nous avons vu que la description contenue dans un objet devrait refléter l'unicité de sa dénotation. De même, l'intension d'une classe (la description qu'elle contient) doit caractériser au mieux son extension (l'ensemble des éléments qu'elle cherche à regrouper). Plus formellement, la description contenue dans une classe devrait *se rapprocher d'une définition* de l'ensemble des éléments qu'elle dénote, à défaut de ne pouvoir en être une.

Il existe des systèmes, notamment les logiques terminologiques, qui supposent que la description contenue dans une classe[1] représente un ensemble de conditions *nécessaires et suffisantes* pour qu'un élément appartienne à l'ensemble dénoté par la classe. Pour notre part, nous considérons que cette hypothèse, même si elle est pratique pour l'automatisation des calculs, n'est pas toujours réaliste. En effet, cela supposerait que la description de la classe soit une énumération des descriptions des éléments dénotés, ou que le monde modélisé soit si bien structuré que les éléments d'un ensemble représenté par une classe aient des descriptions adjacentes. Nous relachons donc cette hypothèse,

[1] Si l'on reprend leur terminologie, il s'agit d'un *concept défini* plutôt qu'une classe.

et admettons cependant que l'ensemble des éléments satisfaisant l'intension d'une classe couvre l'extension de cette classe (la description est trop large).

Les deux entités de représentation à la base des modèles de connaissances à objets sont donc l'objet et la classe. Dans les deux cas, il s'agit de structures comportant une description, cette dernière étant censée caractériser la dénotation de l'entité dans le monde modélisé. Il existe donc un lien étroit entre descriptions et dénotations des entités de représentation.

Mécanismes d'exploitation : coopération description/dénotation

La sémantique d'un modèle de connaissances à objets doit intégrer les deux composantes des structures de représentation. Cette sémantique est généralement dénotationnelle, à valeur dans le domaine modélisé. Nous parlerons d'*interprétation en intension* d'un objet (resp. classe) pour parler de la signification de cet objet (resp. classe) relativement à sa description. Parallèlement, nous parlerons d'*interprétation en extension* d'un objet (resp. d'une classe) pour parler de sa signification relativement à son (ensemble d') identificateur(s). En guise de métaphore, nous pourrions dire que l'extension est à l'intension ce que l'esprit est à la lettre.

Les mécanismes qui considèrent les classes d'objets sont alors tenus de considérer le fait que les deux interprétations, en intension et en extension, ne sont pas équivalentes, et que la seconde est incluse dans la première. Par exemple, lorsqu'on tente de rattacher un objet à une classe, il s'agit de vérifier au préalable, d'une part que la description contenue dans l'objet est compatible avec celle de la classe, et d'autre part que l'identificateur (la dénotation) de l'objet appartient à l'ensemble des identificateurs de cette classe. Il en est de même pour l'acceptation d'un lien de spécialisation entre deux classes d'objets. Autrement dit, toute manipulation, effectuée par le système, des représentations du monde, donc des classes et objets, se fera au regard de leur *description* et de leur *dénotation dans le monde.*

Les mécanismes calculatoires opérant sur les intensions des classes, c'est-à-dire considérant leurs descriptions, sont généralement réalisables en machine, puisque les descriptions ne sont rien d'autres que des références vers d'autres descriptions et/ou vers des données primitives, à savoir des structures syntaxiques homogènes manipulables par des processus automatiques. Par exemple, il est possible de calculer la surface de la Lune à partir de son diamètre : cela relève d'une fonction qui, à partir d'un réel et d'une constante, calcule un autre réel. Si certaines inférences sont faites sur les descriptions des classes et objets, elles doivent aussi être valides au regard de l'interprétation en extension, afin d'être acceptées dans une base de connaissances. Dans l'exemple précédent, cela signifie que le résultat de la fonction de calcul du diamètre doit être validé pour être explicitement associé à l'objet `Lune`. Plus généralement, tout mécanisme du système manipulant des entités de représentation des connaissances doit intégrer dans sa sémantique les deux interprétations de ces entités ; la classification et le filtrage font partie de ces mécanismes, tout comme le rattachement d'un objet à une classe.

On remarque cependant, dans les systèmes actuels, un manque de clarté quant à l'exacte coopération entre interprétations en intension et en extension, lorsqu'elles sont effectivement distinguées. En particulier, on constate que l'interprétation en extension est souvent délaissée au profit de l'interprétation en intension, dans la mesure où cette dernière est directement programmable de par la mise en œuvre de manipulation de structures de données élaborées. C'est comme cela que l'on peut arriver à l'intégration, dans la base de connaissances, du fait que le satellite `Spot` est un satellite naturel, d'après les descriptions respectives de l'objet et de la classe, simplement parce que le système ne s'est basé que sur des descriptions. Il s'agit ici d'un cas d'incomplétude de la modélisation : il manque à la description de `Satellites-naturels` certaines caractéristiques illustrant leur différence d'avec les satellites artificiels.

Le rôle de l'interprétation en extension est essentiel ; c'est elle qui est garante de la pertinence des objets représentés au regard du monde modélisé. Mais effectivement, seul un concepteur de base de connaissances (ou un utilisateur spécialisé dans le domaine de modélisation) peut réaliser cette interprétation, puisqu'il est le seul médiateur entre le monde et la représentation qui en est faite en machine. Le modèle de connaissances doit aider le concepteur à compléter les descriptions contenues dans les entités de représentation, de manière à minimiser l'écart entre les interprétations en intension et en extension de ces entités. Cette recherche de la complétude doit être une ligne directrice du comportement d'un modèle de connaissances. Ainsi, dans l'exemple précédent, le système doit permettre au concepteur, non seulement de détecter immédiatement l'erreur du rattachement de l'objet dénotant `Spot` à la classe représentant les satellites naturels, mais aussi d'enrichir la description de cette classe dans l'optique de minimiser l'écart entre ses interprétations en intension et en extension, afin d'éviter que ce type d'erreur se reproduise.

Plus la définition en intension d'une classe peut se rapprocher de son extension, plus les mécanismes opérant sur les descriptions des entités seront valides dans le monde modélisé. Une équivalence (certes idéaliste) permettrait même d'arriver à une automatisation totale des inférences.

Descriptions et structures de données

Qu'il s'agisse d'une cause ou d'un effet du manque de clarté dans la définition sémantique des modèles de connaissances à objets, les classes de représentation sont parfois utilisées comme support de définition de structures de données, par exemple `Matrice`. En réalité, les concepteurs de bases de connaissances font souvent appel, dans leur modélisation à des données structurées ayant un comportement similaires à celui des propriétés qui caractérisent les objets. Ces structures de données ne sont pas toujours disponibles dans le langage hôte. Deux alternatives se présentent alors :

- les concepteurs définissent ces structures de données en termes du langage hôte, et les intègrent de ce fait au même titre que d'autre types de données comme `entier` ou `chaîne` ;
- les concepteurs définissent ces structures de données en termes du langage de représentation du modèle, elles sont alors intégrées au même titre que des objets ou classes ayant une signification dans le monde modélisé.

Aucune de ces solutions n'est satisfaisante, et ce pour des raisons différentes. En effet, ces structures ainsi définies doivent appartenir à un corps possédant certaines propriétés d'organisation exigées par les mécanismes d'inférence et de vérification de cohérence du modèle. L'appartenance à ce corps passe par la programmation d'opérations complexes sur ces structures (inclusion ensembliste, disjonction et conjonction de sous-ensembles de données ainsi structurées, gestion de l'infini de ces sous-ensembles, etc.).

Dans la première solution, la tâche du concepteur devient donc ardue, d'autant plus qu'il doit aussi signaler dans le code du modèle de connaissances la présence des opérations propres à la nouvelle structure.

La raison pour laquelle la seconde solution n'est pas satisfaisante est, quant à elle, de nature conceptuelle. Le concepteur choisit le langage de représentation pour définir sa structure de données parce qu'il n'a pas d'autre solution. Pourtant, choisir ce langage devrait être motivé par le besoin de représenter quelque chose du monde modélisé, et non pas pour définir des structures à vocation calculatoire. Prenons l'exemple de la définition de la structure de `Matrice`. Si le monde modélisé est l'algèbre linéaire, alors il n'y a pas de problème, toute matrice y est unique, et cette unicité est d'ailleurs traduite par la description de l'objet `Matrice` correspondant. En revanche, si le monde modélisé est l'ensemble des constituants et phénomènes de l'espace, une matrice n'y a pas de signification directe ; et dualement, sa description en possède plusieurs, à partir du moment où elle est associée à plusieurs objets ayant, quant à eux, une signification dans le monde. Dans ce cas, le partage d'un objet de la classe `Matrice` n'est pas toujours pertinent, seule sa description est intéressante lors de la modélisation du monde, ainsi que les opérations de manipulation de cette description (figure 0.3). Dans cet exemple, `Matrice` et `Satellites-Naturels` appartiennent à **deux niveaux d'interprétation différents**, souvent confondus parce que liés au regard du monde modélisé : l'interprétation d'une matrice dans ce monde est tributaire de l'interprétation de l'objet auquel elle est associée.

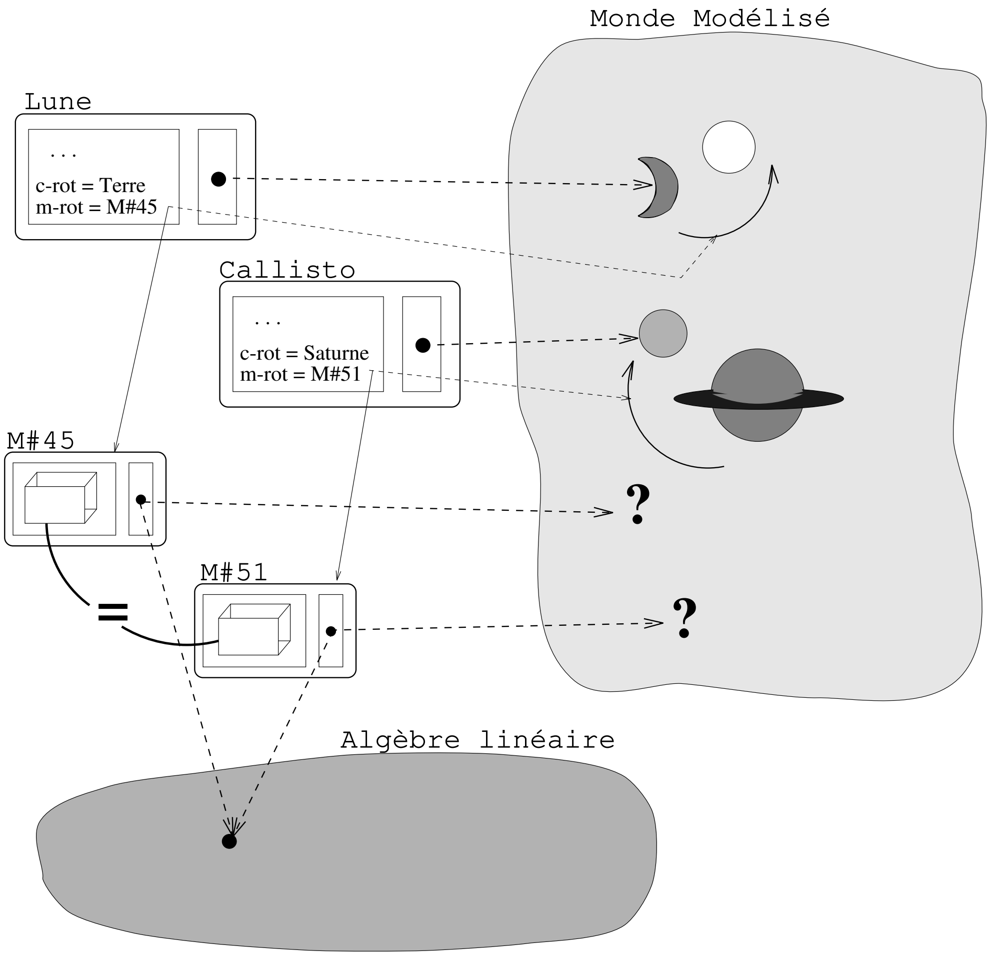


FIG. 0.3 - : Les mouvements de rotation de la Lune autour de Terre, et de Callisto autour de Saturne, sont modélisés par des matrices. Si les descriptions des deux matrices sont égales, il n'en est pas de même de leur dénotation, à partir du moment où elles sont représentées au sein de deux objets de représentation différents. Pourtant, leur interprétation dans le monde modélisé passe par celle des objets `Lune` et `Callisto` auxquels elles sont associées. Indépendamment de ces deux objets, l'interprétation des deux matrices dans le monde modélisé n'est pas pertinente, alors qu'elle l'est si le domaine d'interprétation est celui de l'algèbre linéaire. En particulier, c'est dans ce domaine que sont significatives les opérations de manipulation de matrices : leur résultat ne prend corps qu'à partir du moment où il est associé à un objet représentant une partie du monde modélisé.

Le fait de décrire (ou définir), en termes du même langage de représentation, les classes `Satellites-naturels` et `Matrice`, possède deux inconvénients majeurs.

- Le langage de représentation d'un modèle de connaissances à objets encourage la déclarativité, c'est-à-dire l'expression des connaissances interprétables statiquement, et n'est en conséquence pas adapté au développement de structures de données dont l'intérêt, lors d'une modélisation, provient des opérations de manipulation de ces données structurées. Les concepteurs de système de représentation peuvent alors vouloir adapter le langage de représentation, et de ce fait sa sémantique, dans le but louable de satisfaire les exigences des utilisateurs. Mais une telle adaptation ne serait pas pertinente comme nous le montrons dans le point suivant.
- Le modèle suppose que les descriptions de classes et d'objets ne sont pas des définitions : il se comportera avec les objets `matrices` comme il le fera avec les objets `planètes`. En particulier, toute matrice sera alors interprétée dans le monde modélisé, de façon unique, et le système cherchera à tendre vers l'équivalence de la description et de la dénotation des objets matrices, avec l'aide de l'utilisateur. Par exemple, chaque fois qu'il constatera que deux objets de la classe `Matrice` ont la même description mais pas le même identificateur, il sollicitera l'utilisateur pour lui signifier une incomplétude dans la description des connaissances modélisées. Parallèlement, si l'utilisateur juge que les descriptions n'ont pas à être modifiées (car elles sont complètes), alors les deux objets sont appelés à être fusionnés en un seul pour assurer l'unicité de la description. En conséquence, l'objet contenant la description de la donnée structurée est partagé, bien que ce partage n'ait aucune signification dans le monde modélisé (dans l'exemple de la figure 0.3, si l'objet résultant de la fusion de `M#45` et de `M#51` était partagé par `Lune` et `Callisto`, cela signifierait que ces deux satellites ont le même type de mouvement de rotation, la modification de l'un entraînant la modification de l'autre). Il est aisé de constater qu'un tel comportement du système (exiger l'unicité des descriptions) n'est pas adapté à des objets qui n'ont pas de dénotation directe dans le monde, ou plutôt qui en acquiert plusieurs selon les objets auxquels ils sont attachés, qui eux, possèdent chacun leur propre signification.

Ces deux inconvénients mènent à une attitude des concepteurs de systèmes de représentation des connaissances qui est peu adaptée : modifier la sémantique du modèle de connaissances, et en particulier, biaiser la coopération entre intension et extension des classes de représentation. En effet, nous pourrions penser que la description de types de données en termes du langage de représentation est une solution esthétique parce qu'elle entraîne une homogénéité dans les traitements des structures. Mais cette homogénéité est obtenue en modifiant les mécanismes du modèle de connaissances pour qu'ils s'adaptent à la gestion de structures étrangères à la modélisation car pouvant être définies indépendamment du monde modélisé.

Ainsi, il n'existe à l'heure actuelle aucune solution convenable quant à l'intégration, dans une base de connaissances, de structures de données utiles à la modélisation d'un monde.

Pourtant, il est certain que les diverses applications que l'on peut rencontrer en représentation des connaissances peuvent avoir besoin de structures de données complexes qui ne sont pas disponibles dans le langage hôte d'un modèle. Par exemple, les bases destinées à la représentation des connaissances en biologie moléculaire nécessitent, pour la modélisation du concept de `gène`, la structure de données `Séquence de Caractères pris parmi 'A', 'C', 'G', 'T'`, ainsi que les opérations sur les valeurs ainsi structurées. Ces structures de données, tout comme les types de données primitifs du langage hôte (`Entier`, `Réel`, $\cdots$), sont utilisées dans l'écriture des descriptions des objets de représentation. Autrement dit, le système de représentation doit être capable d'intégrer les interprétations simultanées de ces structures de données et des descriptions des objets.

En particulier, sous ces considérations, le monde modélisé comme domaine d'interprétation n'est plus valide pour les données : une séquence de ces caractères (et les opérateurs qui la manipulent) n'y a en effet de signification qu'à partir du moment où elle est explicitement associée à un gène particulier, dont elle représente la séquence d'ADN.

La **problématique à laquelle nous nous attachons** est alors d'amener les concepteurs de bases de connaissances à cerner les deux niveaux d'interprétations, au moment même de la modélisation du domaine d'application, en leur proposant un système et un langage pour chacun de ces deux niveaux.

Rappelons que les types de données annexes à une modélisation sont présents dans les descriptions contenues dans les entités de représentation. Du coup, le modèle exige qu'ils aient certaines propriétés, dictées par le lien étroit qui existe entre les descriptions des entités et leur dénotation dans le monde. C'est une des raisons pour lesquelles l'intégration de types de données est problématique en représentation des connaissances. En outre, la structure informatique qui accueille les descriptions (intensions) des objets et des classes doit, elle aussi, être manipulée comme une *donnée* structurée qui se décompose en agrégation d'autres structures. Les résultats de ces manipulations doivent alors être interprétés par le modèle de connaissances comme des résultats d'opérations de manipulation des intensions. Décharger le modèle de connaissances de la définition de telles opérations, en les confiant à un système annexe dédié à la définition de structures de données, a pour effet de clarifier la coopération entre intensions et extensions, notamment parce que les intensions seront interprétées dans un autre univers que le monde modélisé, à savoir un univers de valeurs. Cette coopération est en effet étroitement liée à l'existence des deux niveaux d'interprétation dont il a été fait état précédemment.

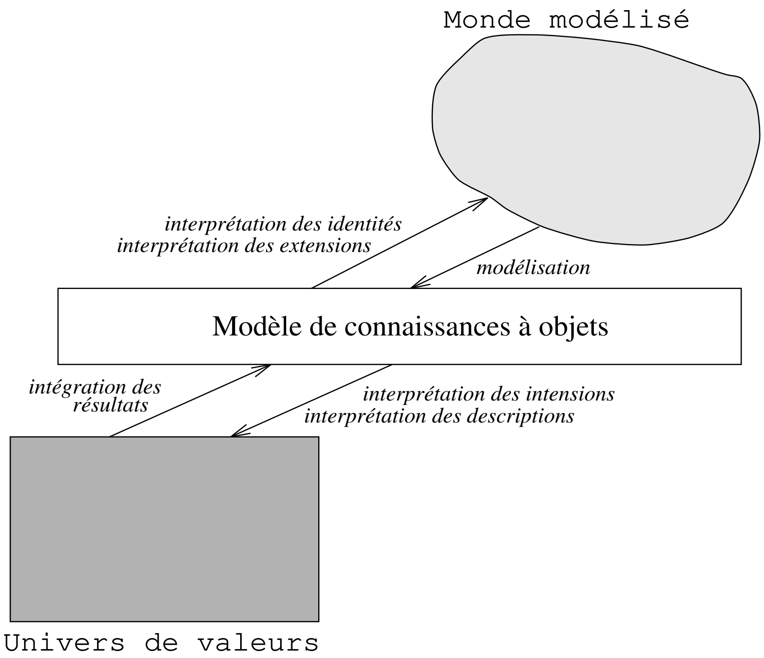


FIG. 0.4 - : Distinction des deux domaines d'interprétation. Les domaines d'interprétation des descriptions et des identificateurs sont distingués. Le modèle de connaissances, dans ce cadre, définit la coopération entre ces deux mondes : c'est par cette coopération que des valeurs se verront attribuer une signification dans le monde modélisé.

Solution proposée

Dans l'optique, d'une part de clarifier la coopération entre interprétations en intension et en extension, et, d'autre part, de cerner les différences entre structures de données calculatoires et descriptions des entités de représentation, nous proposons de **distinguer les domaines des interprétations en intension et en extension des objets et classes d'objets** (figure 0.4). Dans ce nouveau cadre sémantique, si les identificateurs sont toujours à dénotation dans le monde modélisé, aux descriptions d'objets sont associées des valeurs construites sur des données structurées, interprétables dans un univers de valeurs organisé, et manipulables par des opérations spécifiques à ces valeurs. Le résultat de ces opérations prendra alors une signification dans le monde à partir du moment où il sera attaché à un objet ou à une partie d'objet significative. Quant aux classes, leurs descriptions sont interprétées comme des ensembles de ces valeurs.

Toutefois, pour que la distinction des deux domaines d'interprétation soit valide, il s'agit d'établir entre eux une correspondance. Ainsi, le domaine de valeurs est muni d'un ordre partiel sur les ensembles de valeurs, tandis que le monde modélisé est muni de l'inclusion ensembliste (isomorphe à un ordre partiel), les ensembles considérés étant des ensembles de dénotations d'objets.

Dans l'optique de représenter dans un cadre formel l'interprétation en intension des entités et ordres, nous avons conçu et réalisé le système de types METÉO. METÉO (Module Extensible de Types Élaborés pour les Objets) définit deux niveaux de types.

- Les C-types sont l'implémentation de types abstraits de données (structure + opérations). METÉO définit une hiérarchie extensible de C-types, qui se propose d'accueillir de nouvelles définitions de structures de données exigées par une application particulière.
- Les δ-types sont des sous-ensembles arbitraires de C-types, et sont aussi ordonnés selon l'inclusion ensembliste. Contrairement aux C-types, les δ-types ne spécifient pas d'opération sur leurs valeurs. Les δ-types sont réservés à accueillir l'interprétation en intension des entités de représentation.

La relation d'ordre définie sur les descriptions des classes est alors naturellement représentée, dans METÉO, par le sous-typage.

Nous verrons que l'explicitation par METÉO du domaine d'interprétation en intension d'une base et la correspondance entre les deux domaines d'interprétation, clarifient la sémantique même du modèle de connaissances. En effet, à toute opération effectuée sur les descriptions des entités, sera associée une opération équivalente sur les (ensembles de) valeurs. Car, la distinction nette, effectuée entre intensions et extensions, se répercute au niveau des mécanismes d'exploitation et d'inférence : lorsqu'ils contiennent des opérations effectuées sur les descriptions des entités, ces opérations seront substituées par celles définies sur les structures de données du système de types.

Par ailleurs, nous montrerons que METÉO offre un cadre adéquat à l'intégration de nouvelles structures de données, qui seront traitées uniformément, comme le sont, dans la plupart des systèmes, les types de données primitifs importés du langage hôte. En ce sens, le système de représentation TROPES, auquel est couplé METÉO, est enrichi par cette possibilité, sans que le langage d'expression des connaissances ne voit sa sémantique modifiée. PLus généralement, et indépendamment de TROPES, nous montrerons que METÉO est un outil supplémentaire pour la conception de bases de connaissances, car il incite les utilisateurs de modèles de connaissances à objets à s'interroger sur la pertinence des structures qu'ils développent vis-à-vis du monde qu'ils modélisent. Notre étude se situe ainsi dans un cadre fondamental, et reste significative dans des modèles de connaissances ignorant la notion d'objet.

Plan du document

Le document est organisé comme suit :

Le **chapitre 1** définit la notion d'objet en programmation, et montre qu'en représentation des connaissances, même s'il n'est pas toujours pertinent de parler d'*objet*, il est toujours fait état de l'association identificateur/description. Nous analysons alors, dans divers systèmes, la coopération qui est faite entre ces deux aspects d'une unité de représentation, pour constater que ce problème est souvent écarté plus ou moins sciemment. Pourtant, et c'est ce qui fait une des différences d'avec les langages de programmation par objets, cette distinction est essentielle en représentation des connaissances.

Le **chapitre 2** présente le modèle de connaissances à objet TROPES, d'abord informellement, puis en considérant les deux interprétations, en intension et en extension. Il s'agit d'un modèle original du fait de la distinction qu'il réalise, syntaxiquement et sémantiquement, entre les notions de classes et d'objets. En outre, il permet le développement d'une base de connaissances selon différents points de vue coopératifs. Nous avons basé notre étude sur ce modèle, qui est intéressant du fait de la richesse des outils de modélisation qu'il offre, cette puissance ne simplifiant pourtant pas la tâche de la définition de sa sémantique. Nous montrons alors l'intérêt du développement d'un système de types pour TROPES, non seulement dans le but d'étendre le nombre de structures de données disponibles pour les applications, mais aussi pour offrir un cadre formel à l'interprétation des intensions des entités de représentation.

Le **chapitre 3** est réservé à la présentation du système de types METÉO, indépendamment de tout modèle de connaissances. Nous débutons ce chapitre par un très bref rappel des propriétés caractéristiques des systèmes de types, dans le but de dégager ensuite celles que possède METÉO. Les deux niveaux de types de METÉO sont successivement définis, ainsi que les deux niveaux de sous-typage qui tendent à parfaire l'interprétation en intension de la spécialisation entre classes d'objets.

Le **chapitre 4** définit formellement le schéma selon lequel les bases de connaissances de TROPES sont typées dans METÉO. Nous montrons alors l'équivalence entre les intensions des entités et leurs types, et de ce fait l'isomorphisme entre l'algèbre issue de l'interprétation en intension des entités, relations et opérations de TROPES, et l'algèbre issue de l'interprétation ensembliste des types, sous-typages et opérations de METÉO. Ce résultat illustre en particulier l'adéquation de METÉO en tant que support formel de l'interprétation en intension d'applications développées en TROPES.

Le **chapitre 5** s'attache à analyser, pour tous les mécanismes d'exploitation et d'inférence disponibles dans TROPES, la façon dont ils sont, en partie, délégués à METÉO. Il s'agit de l'illustration de la coopération entre interprétations en intension et en extension des entités et ordres disponibles dans TROPES.

Le **chapitre 6** est destiné à illustrer l'intérêt de METÉO, et, en particulier, de la hiérarchie extensible des C-types qu'il propose, en ce qui concerne la possibilité de définir de nouvelles structures de données. Nous verrons, en particulier, qu'associer un C-type à une classe de représentation est un moyen de spécifier le comportement des valeurs construites à partir de la structure de cette classe, ce qui est l'équivalent de la définition de méthodes attachées à une classe en programmation, tout au moins lorsque ces méthodes concernent la manipulation de la structure d'agrégation des attributs de la classe.

Ce document comporte en outre quatre annexes :

L'**annexe A** présente les systèmes de transition correspondant à la normalisation des δ-types.

Celle-ci permet de garantir que deux domaines de valeurs identiques sont représentés par la même expression syntaxique, autrement dit par le même δ-type. Les preuves de complétude, correction, confluence et terminaison de ces systèmes de transition sont aussi fournis.

L'**annexe B** contient toutes les preuves des propriétés et théorèmes énoncés au fil de ce document. En particulier, nous y présentons les preuves de correction et de complétude du typage et du sous-typage, au regard des interprétations en intension des entités et des ordres définis dans le modèle de connaissances.

L'**annexe C** contient quelques algorithmes de gestion des treillis de δ-types, ainsi qu'un algorithme de détection de cycles dans les descriptions TROPES.

L'**annexe D** donne quelques indications en ce qui concerne l'implémentation de METÉO et son couplage faible avec TROPES.

Chapitre 1

Objets, programmes, et connaissances

L'approche qualifiée d'*objet* s'est rapidement étendue et développée dans divers domaines de l'informatique. Ainsi, on retrouve les objets dans les langages de programmation, dans les gestionnaires de bases de données ; ils sont aussi présents en représentation des connaissances. Pourtant, si les domaines investis par la notion d'objet sont multiples, les objectifs servis par ces domaines sont aussi distincts, et en conséquence le sont les interprétations faites de cette notion d'objet et des mécanismes qui l'utilisent comme fondement de leur définition.

Un premier objectif de notre travail est de montrer qu'en représentation des connaissances par objets, l'ambition première étant de permettre l'expression et le traitement informatique de la modélisation d'un monde, il est essentiel de distinguer deux aspects d'un objet $O = \langle O_m ; O_d \rangle$: un objet, tel qu'il est considéré par un système de représentation des connaissances, dénote O_m, une unité du monde, et possède simultanément une représentation structurée O_d qui permet d'intégrer la description de O_m, selon un format exploitable par la machine. En pratique, l'objet est manipulé par des processus qui considèrent tantôt sa signification dans le monde réel, tantôt la structure de sa description. La perversion de cette pratique est qu'il n'est pas rare de voir que les applications décrivent des objets qui n'ont pas de correspondance dans le monde, et qui ne sont intéressants que parce qu'ils possèdent une structure manipulable : ces objets, qui ne sont donc pas des représentations du monde, sont pourtant manipulés comme tels, au même titre et au même niveau d'interprétation que d'autres qui, eux, représentent quelque chose.

Il existe, comme nous l'avons dit, des différences d'objectifs entre langages de programmation et représentation des connaissances. La programmation par objets insiste sur les propriétés des structures de données des objets. La représentation des connaissances considère les deux interprétations de l'objet mentionnées dans le paragraphe précédent, mais la coopération de ces sémantiques reste encore floue.

Dans ce chapitre, nous nous intéressons, d'une part, à la sémantique des objets en programmation, car elle permet d'interpréter la structure de données sous-jacente aux objets. D'autre part, nous abordons les objets dans le domaine de la représentation des connaissances, afin d'illustrer la façon dont différents courants de systèmes traitent la coopération, pour un objet O, entre O_m et O_d. À partir de l'étude de ces différentes sémantiques de l'objet, nous constaterons qu'en intégrant, dans les systèmes de représentation des connaissances, celles considérées en programmation, nous pouvons parvenir à atténuer les risques d'amalgames entre structures de données et objets pour la représentation.

Nous commençons ce chapitre par une présentation générale de la notion d'objet, ainsi que des techniques de regroupement et de mise en relation de ces objets. Nous nous attachons ensuite, dans la section 1.2, à présenter l'interprétation qui est faite des objets dans le cadre de la programmation, tandis que la section 1.3 concerne l'objet pour la représentation des connaissances. Nous accordons, dans la section 1.4, une attention particulière aux langages terminologiques, qui sont devenus une véritable référence en matière de représentation des connaissances.

1.1 Caractéristiques principales des modèles à objets

En informatique, l'épistémologie du terme “objet” est relativement claire, malgré le nombre toujours grandissant de domaines dans lesquels il est utilisé. D'un point de vue conceptuel, l'objet est apparu pour permettre l'abstraction de données, tout en apportant une nouvelle technique de résolution de l'équation de Niklaus Wirth : *Algorithmes + Structures de données = Programmes* [Wir87]. La différence fondamentale d'avec les techniques, jusqu'alors classiques, de programmation dirigée par les traitements, est que, dans l'approche considérant la notion d'objet, les algorithmes et données *ne sont pas séparés*, ils sont justement réunis au sein des objets. On parle dans ce cas de programmation dirigée par les données.

À cette riche notion d'objet sont associés quelques autres concepts de grande importance, désignés par les termes suivants : classe, encapsulation, modularité, comportement, méthode, héritage, réutilisation de code, etc. Quels que soient alors les domaines dans lesquels le concept d'objet est repris, qu'il s'agisse de conception, de programmation, de gestion de bases de données ou de représentation des connaissances, ces notions adjacentes surviennent également, tout au moins en partie. Nous proposons, dans cette section, de présenter brièvement l'*objet* et ses satellites. Une analyse complète et très didactique est disponible dans [MNC+89].

1.1.1 Qu'est-ce qu'un objet ?

Un objet est la description d'un élément ayant sa propre **identité** dans le contexte d'exploitation informatique dans lequel il a été créé. Il détient la donnée de son propre **état**, c'est-à-dire de l'ensemble des propriétés qui le décrivent dans son contexte d'exploitation, appelées **champs** ou **variables d'instances**.

L'objet définit lui-même son propre **comportement**, par le biais de **méthodes** qui lui sont associées. Ces méthodes sont des opérations ; ce sont les seules qui ont l'autorisation de modifier ou simplement consulter l'état de l'objet. Autrement dit, les méthodes permettent à l'objet de contrôler son évolution, même si leur activation peut être sollicitée extérieurement. L'état d'un objet et l'implémentation de ses méthodes sont cloisonnées dans cet objet, c'est-à-dire qu'ils ne sont pas visibles de l'extérieur. On parle alors d'**encapsulation**, cette propriété des objets qui en fait leur pouvoir d'abstraction, dans la mesure où les détails de l'implémentation de l'objet sont cachés au regard de l'extérieur, les informations qu'il contient ne sont accessibles que par l'intermédiaire d'opérations qui constituent justement l'interface de l'objet (figure 1.1, extraite de [MNC+89]).

Même si un objet peut être considéré comme une donnée regroupant un ensemble d'autres données, il ne peut être considéré comme une simple valeur car il est partagé. En conséquence, si un objet est partagé par plusieurs autres (par référence dans des champs), une seule modification de cet objet est considérée par tous ceux qui le partagent. À l'inverse, si deux objets partagent la même valeur, la modification de la valeur de l'un des objets n'entraîne pas la modification de la valeur de l'autre. Par ailleurs, même si deux objets possèdent les mêmes caractéristiques (le même état et les mêmes méthodes), ils ne sont pas égaux car leurs identités sont différentes.

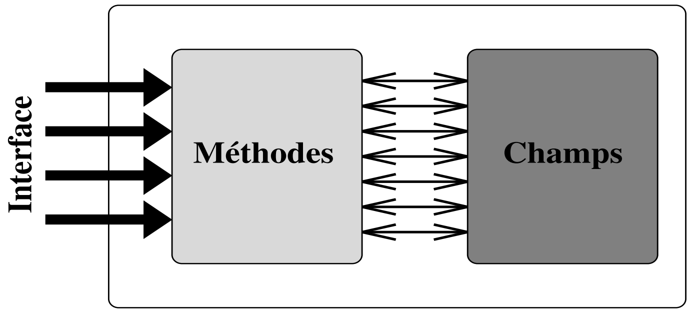


FIG. 1.1 - : Un objet regroupe une partie statique qui représente son état (des champs contenant des données), et une partie dynamique constituée des méthodes (procédures) qui manipulent ces données.

Les objets d'un système contribuent à son comportement en collaborant entre eux. Il existe principalement deux types de relations entre objets.

1. Plusieurs objets communiquent à l'aide d'**envois de messages**, c'est-à-dire que l'un d'eux sollicite l'activation d'une méthode d'un autre, soit parce que le premier désire que le second change d'état, soit parce qu'il attend une information que seul le second détient. L'envoi d'un message est en quelque sorte une requête.

2. Un objet peut faire référence à un autre objet au niveau de la description de son état. En effet, les champs décrivant l'état d'un objet prennent leurs valeurs dans des domaines particuliers ; la valeur d'un champ pour un objet peut être une référence à un autre objet. Il s'agit dans ce cas d'une coopération statique.

1.1.2 Classes d'objets

La *classe* est une composante commune à beaucoup de modèles à objets, et en particulier à ceux que nous considérons ici. Une classe regroupe un ensemble d'objets dont la caractérisation de l'état et le comportement sont similaires. Autrement dit, les objets d'une classe possèdent les mêmes champs et les mêmes méthodes (*Cf.* exemple de la figure 1.2).

La classe est un moule pour la création de ses **instances** (représentation des objets), c'est-à-dire qu'elle leur fournit une interface et une structure. Toute instance est créée à partir d'une classe, ce qui lui confère une identité et une existence propre dans son contexte d'exploitation. On appelle cette création l'**instanciation**.

Alors qu'un objet est une entité concrète dont l'évolution dans le temps et dans l'espace a une signification, la classe est presque toujours considérée comme une simple abstraction. Son rôle est pourtant essentiel, puisqu'elle représente une référence pour l'accès à ses objets. En effet, c'est la classe elle-même qui attribue à ses objets leur sémantique.

1.1.3 Héritage

L'héritage est aussi une notion fondamentale inhérente aux modèles à objets. Il peut avoir de multiples interprétations, et son intérêt se situe à plusieurs niveaux.

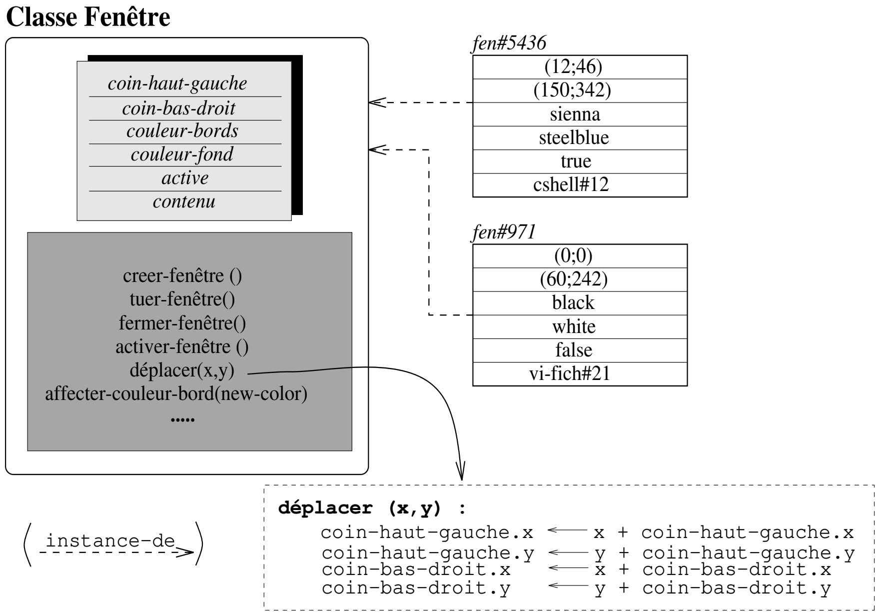


FIG. 1.2 - : La classe `fenêtre` possède deux instances particulières, deux fenêtres existantes. La description de ces instances est orientée par la classe, qui leur impose une position, deux couleurs, un contenu, etc. La classe définit un mode d'évolution de ses instances, par des opérations de manipulation de leur état. Par exemple, la méthode `déplacer-fenêtre(x,y)` a pour effet de modifier les deux champs de position de l'instance à laquelle s'applique cette méthode.

Héritage et spécialisation

L'héritage est avant tout un mécanisme de réutilisation de code, qui est issu de la hiérarchisation des classes d'objets. Dans la plupart des modèles à objets, en effet, les classes sont ordonnées par une relation d'ordre partiel que nous noterons par la suite $\preceq_H^*$ (la réduction transitive de $\preceq_H^*$ est notée $\preceq_H$). La relation d'ordre $\preceq_H$ est appelée *relation de spécialisation*, qui, intuitivement, traduit l'inclusion ensembliste des ensembles d'objets de classes, et se réalise par l'enrichissement des informations disponibles sur les objets.

La terminologie liée à cette relation d'ordre est la suivante : lorsque $C_1 \preceq_H C_2$, on dit que C_1 (resp. C_2) est une sous-classe (resp. sur-classe) directe de C_2 (resp. C_1). Le qualificatif "direct" disparaît si la fermeture transitive partielle de $\preceq_H$ doit être effectuée pour lier deux classes, c'est-à-dire si $C_1 \preceq_H^* C_2$ et $C_1 \not\preceq_H C_2$. Lorsque $C_1 \preceq_H C_2$, on dit alors que C_1 hérite de C_2, cela concerne l'héritage par C_1 des champs et méthodes définis dans C_2 (figure 1.3). La relation $\preceq_H$ se visualise sur un graphe d'héritage (ou graphe de spécialisation), son sommet est une classe dont la définition est donnée par le système indépendamment de toute application (il s'agit de l'unique élément maximum de l'ensemble de classes partiellement ordonné par $\preceq_H$). Cette classe maximum est souvent appelée `Object`, c'est elle qui décrit le comportement *informatique* de tous les objets des classes propres à l'application (ex. `Print-object`).

Le mécanisme d'héritage est ainsi un processus de parcours ascendant de la relation d'ordre $\preceq_H$, qui a pour effet de récupérer, pour une classe, tous les champs et méthodes disponibles dans ses sur-classes, qu'elle ne redéfinit pas. Tout au moins, l'héritage permet de récupérer une caractéristique précise. Dans l'exemple de la figure 1.3, l'activation du mécanisme d'héritage sur la classe `Fenêtre` peut avoir pour but de retrouver la méthode `créer-application-fille()`, afin de l'exécuter pour une instance de `Fenêtre`.

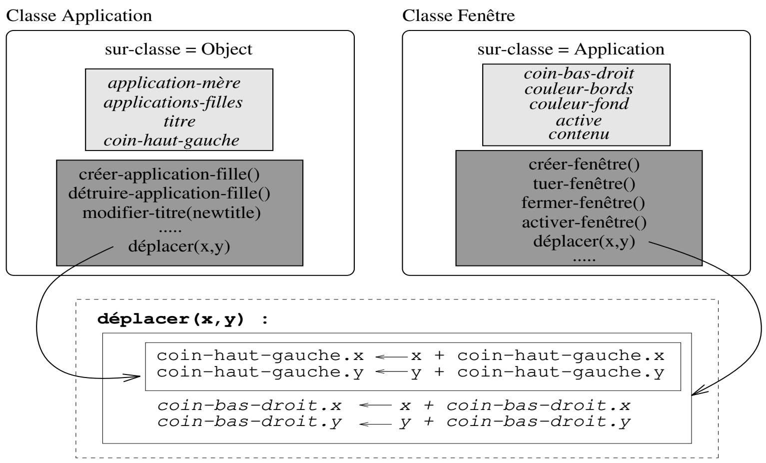


FIG. 1.3 - : L'héritage. Les deux classes de cette figure vérifient : `Fenêtre` $\preceq_H$ `Application`. Cela entraîne que `Fenêtre` hérite de `Application` tous ses champs et méthodes. Certaines des caractéristiques héritées peuvent être redéfinies dans la sous-classe (*substitution*). C'est le cas ici de la méthode `déplacer(x,y)`. En outre, la sous-classe peut naturellement introduire de nouvelles caractéristiques qui ne sont pas pertinentes dans la sur-classe mais qui le deviennent dans la sous-classe (*enrichissement*) ; c'est le cas, par exemple, des champs `contenu` ou `couleur-fond`, qui ne sont pas des caractéristiques propres à toutes les applications graphiques.

L'héritage est un concept fondamental de la programmation orientée-objet. Plus exactement, la programmation orientée-objet met à profit cette notion d'héritage, qui existe implicitement dans tout environnement de programmation, à objet ou non. En programmation par objets, on organise les programmes autour de l'héritage. Non seulement ce dernier concerne des méthodes de conception fondées sur la modularité et la répartition de l'information, mais cette propriété de modularité inhérente à l'héritage permet la réalisation de prouesses techniques amenant des facilités inégalées quant à la gestion de l'évolution des applications. Les quantités d'études menées autour de l'héritage, relatives aussi bien aux propriétés de la relation sous-jacente, qu'aux techniques pour représenter ces ordres et les parcourir, montrent effectivement l'importance et la complexité de cette notion.

Les héritages

Il existe un certain nombre de variations autour de l'héritage et de la relation sur laquelle il est établi.

D'une part, on parle d'héritage *statique* ou *dynamique* pour désigner deux techniques d'utilisation du mécanisme d'héritage. La première consiste à recopier dans une classe toutes les définitions héritées de sa sur-classe, lorsqu'elle ne les substitue pas (cette recopie, bien entendu, n'est pas nécessairement une réécriture physique, il s'agit généralement de tables de pointeurs). L'héritage dynamique, quant à lui, consiste à retrouver la définition d'une méthode (ou de champ) héritée au moment de l'exécution, lorsque son activation est sollicitée par un envoi de message. Les deux approches ont leurs avantages et inconvénients, l'héritage statique favorise l'efficacité des accès, tandis que l'héritage dynamique rend optimal tout processus de remise à jour des objets. En outre, si l'héritage dynamique offre une grande souplesse de conception, l'héritage statique impose, en quelque sorte, une conception systématiquement descendante.

D'autre part, sont distingués l'héritage *simple* (une classe possède une et une seule sur-classe directe) et l'héritage *multiple* (une classe possède une ou plusieurs sur-classes directes). Il s'agit plus précisément d'une propriété de la relation d'ordre $\preceq_H$. Selon l'option d'héritage choisie, la hiérarchie de classes est, dans le premier cas, un arbre, et dans le second cas, un graphe (les circuits sont toujours interdits). Dans le cas de l'héritage simple, la relation d'ordre $\preceq_H$ entre les sur-classes (directes et indirectes) d'une classe donnée est une relation d'ordre total ; dans le cas de l'héritage multiple, l'ordre n'est plus que partiel.

Si l'héritage simple est relativement facile à gérer, l'héritage multiple peut introduire des conflits, dont l'origine provient de la question fondamentale suivante : si une classe C possède deux sur-classes incomparables C_1 et C_2 qui possèdent toutes deux la méthode (et plus généralement la caractéristique) m, avec des définitions différentes, de quelle classe, parmi C_1 et C_2, C hérite-t-elle m ? Ce type de conflit peut être dû à une simple erreur de conception, qui a mené à attribuer le même nom à deux actions ayant une sémantique différente[1]. Il y a réel conflit lorsque deux méthodes, ayant une sémantique identique mais des résultats différents, peuvent être choisies. En ce sens, les conflits mettent en évidence une mauvaise adaptation de la sémantique de $\preceq_H$ aux exigences particulières du domaine d'application.

Il existe alors plusieurs traitements possibles des conflits dus à l'héritage multiple. La première technique, la plus simple et la plus répandue, mais aussi la plus radicale, est celle qui interdit les conflits en imposant un renommage des méthodes. C'est en réalité un moyen de ne pas reconnaître l'équivalence sémantique de deux méthodes ou champs. En effet, l'identificateur d'une caractéristique de classe est généralement porteur de sa sémantique.

Une autre approche consiste à établir des priorités quant aux diverses méthodes candidates, mais aucun ordre de priorité n'est universellement acceptable. Ainsi, outre l'ordre des priorités correspondant à l'ordre de déclaration des méthodes, la technique principale de mise en évidence de priorités consiste à élaborer un parcours de la hiérarchie des classes dont l'ordre est dicté par la linéarisation de la fermeture transitive de $\preceq_H$ [DH89]. Cette technique s'appuie sur des heuristiques concernant les propriétés de la relation $\preceq_H$ et à sa signification dans le domaine d'application.

Notons en outre les travaux de Luca Cardelli concernant l'héritage multiple [Car84], dont l'approche est différente. Il a attribué à l'héritage multiple une sémantique fixe basée sur l'intersection des définitions de classes qui entrent en conflit. Il ne s'agit donc pas ici de chercher quelle est la définition de méthode la plus appropriée, mais de considérer simultanément et conjointement les diverses définitions. Pour cela, L. Cardelli a établi une théorie de l'héritage multiple fondée sur la théorie des *records* et du sous-typage (section 1.2.2).

1.2 Objets et programmes

De nombreux objectifs sont servis par des systèmes issus de la technologie fondée sur les objets. En particulier, depuis la conception de Simula qui a mené à une première formalisation des notions de classe et d'objet [BDMN73], nombre de langages de programmation dits “orientés-objet” sont apparus, illustrant l'évolution de la notion d'objet et des concepts qui l'entourent. Parmi ces langages, nous pouvons citer Smalltalk [GR83], Ada [Uni83], C++ [Str86] ou encore Clos [BDG+88], chacun ayant introduit (ou synthétisé) des notions propres, inspirant la conception de beaucoup d'autres langages parfois plus spécialisés ou appliqués.

Un programme écrit dans un langage orienté-objet est une collection d'objets qui interragissent par envois de message. Les procédures et données sont contenues dans les objets. Outre les

[1] Par la suite, nous parlons des résolutions de l'héritage multiple dans le cas de l'héritage de méthodes, mais les mêmes techniques, éventuellement simplifiées, peuvent être appliquées pour résoudre l'héritage des champs.

différentes notions exposées dans la section 1.1 qu'ils interprètent en fonction de leurs objectifs, les langages à objets ont introduit plusieurs autres caractéristiques nécessaires, d'une part, à la définition de sémantiques adaptées, et d'autre part, à l'amélioration de l'efficacité et de la souplesse de ces langages. Notons que les recherches menées dans ce cadre sont à la base, bien souvent, de nombreuses études de systèmes ayant des objectifs différents, tels que les systèmes de gestion de bases de données à objets ou la conception de systèmes d'expoitation.

Nous présentons très brièvement dans cette section l'essence des langages de programmation orientés-objet, en nous appuyant sur les composantes qui permettent de définir leur sémantique.

1.2.1 Rôle d'une classe : méta-niveau et types

Il n'existe pas, à proprement parler, de théorie des langages orientés-objet. Pourtant, il semble qu'il y ait un consensus solide quant au rôle joué par toute classe d'objets [Weg87]. Ce rôle est en fait double, l'un se ramène à des considérations d'implémentation (indépendantes de toute application), tandis que l'autre se situe dans le cadre de la spécification de programmes.

- La classe est *un moule pour la création de ses instances*. D'une part, elle est à l'origine de leur attribution d'identité, et d'autre part, sa structure permet de déduire celle de ses instances (par copie).
- La classe joue le rôle d'un *prédicat d'appartenance*, ce prédicat étant issu des conditions portées sur les champs et méthodes qui la définissent. La classe est alors une abstraction, elle permet de spécifier la description et le comportement de ses objets. De ce fait, pour qu'une instance puisse appartenir à une classe, elle doit satisfaire le prédicat d'appartenance correspondant. En outre, ce prédicat influe au niveau de la relation d'héritage.

Ainsi, la classe relève, d'une part, de considérations d'implémentation, et d'autre part, d'un principe d'abstraction des données. S'il existe des langages orientés-objet qui ne distinguent pas cette mixité, la plupart d'entre eux l'ont cernée, et ont proposé en conséquence des techniques formelles de mise en évidence de la différence de ces deux rôles.

Parmi ces techniques, nous retenons les deux plus répandues, qui sont la notion de *méta-niveau* et le *typage* des classes.

Le méta-niveau d'un langage à objets

Une classe, telle que définie par l'utilisateur, est censée spécifier le comportement de ses objets. La notion de *méta-niveau* est une technique de spécification du comportement des classes d'objet, en les considérant elles-mêmes comme des instances de classes, appelées des méta-classes. On distingue alors trois niveaux de programmation : les *méta-classes* (indépendantes de l'application), qui servent à la spécification du comportement des *classes*, elles-mêmes génératrices d'*instances* [Bri85] [Coi87].

En particulier, dans certains systèmes, les méta-classes définissent les méthodes de création et de suppression d'instances de classes, qui sont activées par envois de message. En ce sens, le méta-niveau d'un langage à objets comporte la définition d'une classe vue comme un moule pour la génération d'instances. Autrement dit, les méta-classes définissent, entre autres, la nature du lien qui unit une classe à ses instances, indépendamment de la signification de ces objets dans le domaine d'application.

Tous les langages orientés-objet ne définissent pas un méta-niveau. Il n'en reste pas moins que l'aspect implémentation des classes reste défini par une sémantique opérationnelle, qui est en général indépendante de toute application.

Types de classes

Dans les langages de programmation orientés-objet, les classes sont parfois considérées comme des types abstraits de données, du fait qu'elles sont la spécification des objets qu'elles peuvent créer : elles leur donnent une structure (les caractéristiques d'un état propre aux objets) et le comportement de cette structure (des opérations de manipulation et d'évolution de cet état). En réalité, la classe est plus qu'une spécification, puisqu'elle contient aussi la réalisation (l'implémentation) de cette spécification. En ce sens, il est naturel d'affirmer que la définition d'une classe contient la définition d'un type abstrait de données. D'ailleurs, l'origine même des mots *classe* (latin) et *type* (grec) appuie la différence de leurs significations tout en renforçant le lien qui existe entre ces deux notions : la classe concerne un ensemble de personnes ou de choses partageant des propriétés, tandis que le type est justement l'ensemble des propriétés distinctives d'un ensemble de personnes ou de choses.

En conséquence, de nombreux langages orientés-objet ont introduit la notion de type de classe afin d'expliciter ce rôle de spécification abstraite de l'état et du comportement des objets. Le type d'une classe est généralement constitué de la spécification des champs de cette classe, et de la signature de ses méthodes. Les types se voient ensuite associer une sémantique algébrique. C'est ainsi par le type d'une classe que son rôle d'abstraction des données est spécifié, tandis que la classe est une réalisation de son type, c'est-à-dire qu'elle fournit le code des méthodes, en accord avec la spécification abstraite des signatures. Par conséquent, à une classe ne peut correspondre qu'un seul type, alors qu'à un même type correspond plusieurs classes : une classe est l'implémentation d'un type abstrait de données [PS91].

En poursuivant dans cette direction, nous pouvons dire que la sémantique des classes, vues comme des spécifications d'objets, est définie par leurs types. D'ailleurs, les types étant des spécifications, ils sont aussi à la base des vérifications de compatibilités.

- D'une part, c'est le type d'une classe qui est utilisé comme prédicat d'appartenance d'une instance à cette classe.

- D'autre part, les types sont utilisés pour tester la validité des envois de message. Par exemple, lors de l'activation d'une méthode paramétrée par une classe, le système de types vérifie que l'objet en paramètre appartient bien au type de la classe donnée dans la signature de la méthode[2].

Lorsque les classes sont interprétées comme des types, les instances de ces classes sont, quant à elles, traitées en valeur. En effet, la notion d'identité, propre aux objets mais non aux valeurs, est considérée au niveau implémentation.

[2] Notons ici que cette possibilité permet au système de procéder à de telles vérifications sans violer la propriété d'encapsulation, cette dernière portant sur les classes vues comme des implémentations de spécifications.

Conclusion

D'un point de vue implémentation, les classes et leur comportement sont spécifiés opérationnellement, éventuellement par le biais de l'existence d'un méta-niveau. D'un point de vue des objets qu'elles regroupent, la classe est sémantiquement définie par son type, et en conséquence, la sémantique des classes est celle donnée aux types ; elle peut être dénotationnelle, algébrique, opérationnelle ou encore logique.

Les classes, vues comme des structures de données, sont donc interprétées vers des types abstraits, les objets des classes correspondant alors à des valeurs de ces types. En ce qui concerne l'ordre sur les classes, matérialisé par l'héritage, il s'interprète naturellement par le sous-typage, lorsque les classes sont considérées comme étant des types abstraits.

1.2.2 Héritage et sous-typage

L'héritage est un mécanisme de parcours de la hiérarchie des classes, ordonnées par la relation $\preceq_H$. Dans tous les langages à objets, cette relation est qualifiée *d'héritage*, ou plus justement de spécialisation. Un très grand nombre d'études visant à construire une théorie de l'héritage ont été effectuées dans le cadre des langages de programmation orientés-objet [Coo88] [Tou86] [BW87] [Car84] [WZ88]. Cette profusion de formalisations de l'héritage, dont une des principales difficultés est liée à sa multiplicité, est rendue nécessaire de par son importance. En effet, à cette notion d'héritage sont associées des techniques de conception en génie logiciel qui requièrent une définition précise de ce mécanisme et de la relation sur laquelle il se base. De plus, l'héritage concerne une organisation particulière des programmes, qui permet une gestion de l'évolution définie plus naturellement et de façon plus satisfaisante à tous les points de vue. Outre quelques langages tels que Ada qui ne considèrent pas cette notion d'héritage, tous les langages à objets élèvent l'héritage au rang de composante essentielle.

Compte-tenu de l'existence de types de classes, et l'héritage étant une relation entre classes, ce dernier a été maintes fois assimilé, tout au moins sa relation sous-jacente, à la relation de sous-typage. Malheureusement, cette notion de sous-typage est, quant à elle, très vaste, et s'interprète de multiples manières. En effet, il existe, en programmation, diverses raisons qui conduisent à l'élaboration d'un ordre sur les types [DT88] :

1. organiser l'application de cœrcions[3] par les compilateurs,
2. offrir un support pour la résolution de la surcharge (*overloading*),
3. offrir un support à la réalisation de vérifications et d'inférences de types [Mit84],
4. offrir un support adapté à l'héritage.

Malgré cela, les objectifs du sous-typage dans le cadre des langages à objets sont plutôt bien cernés. Le sous-typage n'est pas l'héritage [CHC90], il n'en est qu'une interprétation utilisée à des fins de vérifications de compatibilités de spécifications, quand il n'est pas destiné à être une matérialisation de la relation $\preceq_H$.

Le sous-typage, tel qu'il est étudié dans le cadre de la programmation orientée-objet, a pour objectif de traduire la notion d'*affinement* des descriptions et comportements d'objets. Autant

[3] "Cœrcion" est un terme anglais, que nous utiliserons par la suite car son usage est devenu courant même en français, bien que sa traduction française exacte soit "Cœrcition".

une interprétation ensembliste (sémantique dénotationnelle) des types, et ainsi du sous-typage, est suffisante pour caractériser l'affinement de l'état des objets (en termes de champs), autant elle ne suffit pas à pallier le délicat problème de l'affinement du comportement de ces objets (en termes de méthodes).

L'affinement des champs, qui sont spécifiés par des types, revient à tester la compatibilité au niveau de l'enrichissement de la sous-classe et au niveau des substitutions. Lorsqu'un champ d'identificateur f de type t_1 dans une classe C de type $t(C)$ est substitué dans une sous-classe de C par un champ de même identificateur mais avec un type t_2, il y a compatibilité au regard du sous-typage si t_2 est un sous-type de t_1 (cette condition doit être vérifiée pour tous les champs de C qui sont substitués dans sa sous-classe).

Indépendamment des difficultés rencontrées dans le cas de l'héritage multiple, l'interprétation sémantique de l'affinement des méthodes d'une classe vers ses sous-classes reste un problème ouvert, qui est très largement discuté dans de nombreuses conférences (ECOOP, OOPSLA...). Les types de classes définissent la signature des méthodes. La notion de comportement d'une classe peut théoriquement être spécifiée, à partir de son type, par une algèbre dont :

1. la signature comporte l'ensemble des types S spécifiant l'état de la classe (spécifié par ses champs typés), ainsi que l'ensemble F des symboles d'opérations correspondant aux méthodes de la classe,

2. la sémantique est donnée par une fonction d'interprétation, qui à S associe un domaine de valeur, et à tout symbole $f \in F$ associe une fonction. Une autre possibilité pour la définition de la sémantique de l'algèbre est la donnée d'équations axiomatiques portant sur les signatures des méthodes.

Dans ce cadre, l'affinement des comportements est sémantiquement défini par l'introduction de sous-algèbres[4].

Luca Cardelli a proposé un cadre théorique particulièrement adapté à la définition d'une sémantique de l'héritage, fondé sur la définition du sous-typage entre types *records* [Car84]. Un type *record* est une collection non ordonnée d'associations étiquette/type, où un type peut être primitif (entier, booléen, etc.), construit (liste, produit cartésien) ou encore fonctionnel. Le type d'une classe est alors un *record* dont les étiquettes correspondent aux noms des champs et méthodes, et dont tous les types associés sont des types fonctionnels (d'arité éventuellement nulle lorsqu'associés aux champs), correspondant à la signature des méthodes. La notion de type *record* est augmentée d'un ensemble d'opérations de manipulation [CM91], telles que la conjonction, la disjonction, ou plus simplement la substitution. L'interprétation des *records* est ensembliste, son domaine est le domaine récursif de Scott [Sco76] qui offre un cadre idéal à la notion d'inclusion, et donc de sous-typage (voir chapitre 3). En ce sens, l'affinement des comportements, interprété par le sous-typage entre *records*, ne concerne que les signatures des méthodes, ce qui revient à considérer le sous-typage de types fonctionnels (appelés aussi applicatifs). Ce sous-typage est encore sujet de débats, puisque deux interprétations en sont faites, portant les noms de contravariance et de covariance[5]. Notons ici l'enrichissement de la théorie des *records* réalisée par Hassan Aït-Kaci [AK86], consistant à permettre la mise en relation de plusieurs étiquettes au sein d'un même *record*, ce qui a mené au développement

[4] La théorie sous-jacente est celle des algèbres multi-sortes ordonnées (ce sont des algèbres construites sur des ensembles de sortes ordonnées, induisant un ordre sur les algèbres).

[5] La première, souvent préférée, consiste à étendre, dans le sous-type, le nombre de paramètres en entrée d'une méthode, et à affiner le type de son résultat, tandis que la seconde prône la diminution du nombre de paramètres et la généralisation du type du résultat.

de langages fondés sur la notion de ψ-termes. Les mêmes opérations que celles effectuées sur les *records* sont applicables sur les ψ-termes, et ce grâce à une sémantique issue de la logique.

Il y aurait beaucoup à dire à propos des théories du sous-typage. Nous nous contenterons ici de retenir que les systèmes de types élaborés pour les langages de programmation orientés-objet considèrent *les types comme la spécification des classes du point de vue des propriétés des objets qu'elles regroupent.* Ces propriétés sont aussi bien descriptives que comportementales, et toute la difficulté des systèmes de types est de définir, non seulement l'affinement des comportements, mais, plus généralement, la compatibilité des comportements. Au regard de l'héritage, le sous-typage doit garantir que toute méthode définie dans une classe C doit pouvoir être associée à tout objet appartenant à une sous-classe de C. Par exemple, si l'on considère les classes `Carré` $\preceq_H$ `Losange`, la méthode `Losange.calcul-aire` qui calcule l'aire d'un losange doit pouvoir être utilisée pour calculer l'aire de n'importe quel carré, même si cette méthode est substituée dans `Carré`. Un sous-typage puissant permettrait de garantir l'équivalence des résultats de l'application de `Losange.calcul-aire` et de `Carré.calcul-aire` à un objet de la classe `Carré`, et cela exige une sémantique opérationnelle. Une autre variété de sous-typage se "contente" de vérifier que la signature de `Carré.calcul-aire` est compatible avec la signature de `Losange.calcul-aire`.

1.2.3 Conclusion

Pour les langages de programmation par objets, un programme est réparti en objets encapsulant des données et des opérations de manipulation de ces données. "*L'idée est de voir le fonctionnement du programme non pas comme le déroulement d'un algorithme, mais comme l'animation d'un modèle réduit*" [Per92b], l'objet étant justement un modèle réduit. En sus, les langages à objets représentent, par la relation d'héritage, une technique de conception descendante, par affinements successifs de la spécification des objets, ce qui mène à des techniques d'optimisation de l'organisation des informations et des accès à ces informations.

Les objets qui ont une description et un comportement similaires au regard de l'application considérée, sont regroupés au sein d'une classe, dont le rôle est, d'une part, de leur attribuer une identité et une structure, et, d'autre part, de définir justement la nature de la description et du comportement qui sont à l'origine de leur regroupement : on rejoint ici l'exacte raison d'être de la notion de type de données. Par conséquent, ce second rôle des classes est généralement délégué au *type* de la classe. Un type est ainsi l'expression de la spécification de classes (ces dernières en étant chacune une réalisation), et les vérifications, statiques ou dynamiques[6], portant sur des compatibilités de spécifications, sont effectuées sur les types. En conséquence, le sous-typage dans les langages à objets est une interprétation relativement naturelle de la relation d'héritage, fondée sur l'affinement des spécifications de classes (une spécification étant, rappelons le, la donnée des caractéristiques d'un état, doublée de la donnée des spécifications d'opérations traduisant la manipulation et l'évolution de cet état).

Cette caractérisation des objets, des classes et de l'héritage en programmation, par des concepts de la théorie des types, est très pertinente. Pourtant, elle ne s'est pas encore adaptée aux caractéristiques propres à la représentation des connaissances, peut-être parce qu'en représentation, la description d'un objet, représentée elle aussi grâce à des structures de données informatiques, entretient un lien étroit avec la signification de cet objet dans le monde modélisé. Ce manque de formalisation de la description des objets au regard de leur implémentation mène pourtant à des ambiguités en ce qui concerne la définition des opérations de manipulation des objets de représen-

[6] Notons que la plupart des systèmes de types dédiés à des langages orientés-objet sont statiques, c'est-à-dire que toutes les vérifications s'effectuent à la compilation.

tation. En effet, il est important de distinguer, au sein d'un système :

1. les opérations de manipulation de l'objet en tant que structure de données (ex. impression de certaines caractéristiques de l'objet, suppression de l'objet...),
2. les opérations qui modélisent une partie du comportement réel de ce que cet objet dénote (signifie) dans le monde modélisé (ex. si l'objet est un cheval, la course d'un cheval est une telle action).

Dans la mesure où les opérations de la seconde catégorie se représentent généralement en termes d'accès aux champs (attributs) de l'objet, de même que certaines opérations de la première catégorie, il est très facile de toutes les représenter au même niveau, comme des méthodes de l'objet. Mais si l'on se rappelle que l'objet est censé contenir la description de ce qu'il signifie dans le monde, on constate que les opérations de la première catégorie n'ont pas à faire partie de cette description. Que signifie dans le domaine des animaux, l'opération d'impression ? Ne pas distinguer ces deux catégories d'opérations peut ainsi provoquer des erreurs de conception et de modélisation. Afin de cerner la portée de ces remarques, nous proposons, dans la section suivante, d'analyser, dans le cadre de la représentation des connaissances, les multiples interprétations de notions qui s'apparentent à l'objet, afin de dégager le rôle exact que joue la description d'une unité de représentation au regard de sa dénotation dans le monde.

1.3 Objets et connaissances

Si l'objet est devenu un thème conceptuel central, son champ d'activité ne se limite pas au domaine des langages de programmation. Les recherches en représentation des connaissances ont été marquées par deux courants d'idées majeurs, le premier étant la notion de réseau sémantique [Qui68], le second étant les *frames* de Marvin Minsky [Min75]. La réunion conceptuelle des deux a mené à l'avènement d'une structure unitaire, interprétable de façon unique, et contenant parfois sa propre description, à l'instar des modèles à objets. Cette structure, en représentation des connaissances, a pour finalité d'accueillir la modélisation de faits et phénomènes réels. Si cette structure n'est pas toujours appelée "objet" (on retrouve les termes *concept*, *individus*, *nœud*, etc.), il n'en demeure pas moins qu'il existe de fortes similitudes entre ses caractéristiques intrinsèques et celles d'un objet de langage de programmation. Pour aller plus loin, les relations et mécanismes, considérés par les systèmes de représentation autour de cette structure de connaissance, sont étrangement proches de ceux habituellement rencontrés en programmation orientée-objet. Par exemple, la distinction classe/instance est proche de la distinction structures de connaissance générique/spécifique, et de ce fait, on retrouve aussi en représentation des ordres sur ces structures.

Pourtant, si l'objet "connaissance" et l'objet "programme" ont des composantes communes, ils diffèrent sur beaucoup de points, tant conceptuels qu'implémentatoires, ce qui amène d'ailleurs les systèmes de représentation à ignorer, de façon délibérée, certaines notions développées en programmation orientée-objet. Nous proposons dans cette section une rapide présentation des objectifs des systèmes développés pour la représentation des connaissances et des problèmes qu'ils soulèvent.

1.3.1 Objectifs de la représentation des connaissances

La représentation structurée des connaissances, que l'on qualifiera de représentation des connaissances, a pour objectif principal le développement de systèmes munis de langages réservés à l'expression des connaissances et à son exploitation. Nous qualifierons ces systèmes de *systèmes de représentation*, qui offrent un modèle de connaissances[7].

De façon générale, l'objectif des modèles de connaissances est donc de fournir un cadre formel dédié à l'expression des connaissances d'un domaine particulier, de telle façon que les représentations qui en sont faites par l'humain, traduites en termes d'un langage "informatique" puissent être exploitées par des mécanismes d'inférences généraux et indépendants d'un problème précis à résoudre, et ce dans le but d'élargir justement l'ensemble des connaissances portées sur le domaine considéré. Un modèle de connaissances est ainsi composé de deux parties principales :

- un **langage de représentation des connaissances**,
- des **mécanismes généraux d'exploitation** des expressions (termes) de ce langage, visant, entre autres, à compléter ces expressions et à en créer de nouvelles (on parle alors d'inférences).

Dans le cas de la représentation par objets, le langage sous-jacent impose une structuration des expressions de connaissance, fondée sur l'identification, dans le monde modélisé, d'entités propres, ou d'ensembles d'entités, puis sur la restitution des caractéristiques de ces (groupements d') entités. On est très proche de la notion d'objet en programmation, dans la mesure où l'on cherche à identifier des entités individuelles, possédant une identité (une existence).

Le langage de représentation est généralement muni d'une sémantique dénotationnelle dont le domaine d'interprétation est le monde modélisé. Les mécanismes d'exploitation s'interprètent, quant à eux, à partir de la même sémantique, ce qui confère à leur déroulement (et donc résultat) une signification dans le monde modélisé. Le modèle de connaissances intègre la définition sémantique du langage et des mécanismes d'exploitation.

La difficulté majeure de la représentation d'un domaine de connaissances dans un système informatique réside dans la *description* de cette connaissance. Il s'agit alors, pour les modèles de représentation, de fournir des outils informatiques qui :

- facilitent l'expression et la structuration des connaissances, en offrant un langage de représentation à la fois expressif sans pour autant que ses termes soient inexploitables,
- facilitent la vérification de la cohérence des modélisations représentées.

En outre, une composante forte des systèmes de représentation des connaissances est qu'ils doivent permettre l'explication de toute inférence de résultat.

Absolu de la représentation

Les systèmes de représentation par objets sont développés afin de permettre la représentation d'un monde, *indépendamment de l'utilisation qui pourrait être faite de la représentation obtenue.* C'est-à-dire que la modélisation effectuée doit l'être, dans la mesure du possible, dans l'absolu. Il s'agit ici d'un objectif certes idéaliste, non dépourvu d'ambition, mais son atteinte est la clé de la réutilisation et de la confrontation de bases de connaissances issues d'un même monde modélisé, mais développées dans des buts d'exploitation distincts, c'est-à-dire sous différents angles de modélisation.

[7] L'appellation "modèle de connaissances" (*Knowledge model*) a été choisie par souci d'uniformité avec l'expression "modèles de données" (*Data model*).

Bien entendu, la description de la connaissance dans un langage de représentation doit tenir compte de la sémantique le définissant. La prise en compte de cette sémantique lors des étapes d'identification et de construction est nécessaire à la cohérence des résultats des mécanismes d'inférence.

Cette affirmation de l'indépendance entre *représentation* et *utilisation* marque une différence de taille entre les systèmes de représentation par objets et les systèmes dits “experts”, ces derniers étant développés à des fins d'exploitations identifiées (cette identification étant d'ailleurs une étape nécessaire lors de la conception des bases) : ainsi, dans les systèmes experts, et contrairement aux systèmes de représentation par objets, la modélisation dépend d'un objectif de résolution à atteindre et des moyens mis en œuvre pour les atteindre.

Déclaratif/procédural

Pour des raisons d'explicabilité, de lisibilité, mais aussi et surtout pour permettre la maîtrise sémantique des faits représentés, les systèmes de représentation par objets prônent l'expression déclarative des connaissances, par opposition à toute expression procédurale telle que celle réalisée par des systèmes à base de règles. Autrement dit, ils cherchent à favoriser la *description* de faits et de phénomènes portant sur ces faits, en imposant leur décomposition en termes structurés. La différence entre un modèle procédural et un modèle déclaratif est que, dans le premier cas, seul le résultat d'une déduction peut être interprété par le système et, dans le second cas, toutes les étapes d'une déduction sont interprétables. En effet, une définition de procédure n'est finalement qu'un texte, indivisible, à moins qu'elle soit justement définie, explicitement et dans un langage interprétable par le système de représentation, comme une séquence d'actions élémentaires, elles aussi connues du système. L'intérêt d'une décomposition qui puisse être comprise par le système est qu'en conséquence il pourra expliquer l'effet de l'application de cette décomposition à des données (ou connaissances représentées) particulières, et ce en fonction du contexte d'activation.

Nous n'avons pas pour but de décider quel est le meilleur type de formalisme parmi ces deux possibilités, car cela relèverait de l'extrémisme. Terry Winograd, entre autres, l'a bien montré [Win85], en tentant de trouver un compromis entre les deux.

Programmation/représentation : différences d'objectifs

Il est bien évidemment faux d'affirmer qu'un programme (un enchaînement d'instructions) n'est pas une connaissance, il exprime en effet un savoir-faire. En ce sens, une collection de programmes, liant une collection de données qu'ils manipulent et s'échangent, peut être considérée comme une base de connaissances, un ensemble de savoir-faire. De ce point de vue, on peut dire qu'un langage de programmation peut avoir pour objectif la *représentation de connaissances exécutables*, et justement, elles sont toujours exécutées dans un contexte précis, en fonction d'un *but à atteindre*, ou à des fins de simulation. C'est ici que se situe une distinction majeure entre programmation et représentation : l'écriture d'un programme n'a de sens que parce qu'il est censé être exécuté et fournir un résultat dont on présuppose la teneur, tandis que la représentation d'une connaissance factuelle est brute, elle n'est pas exécutée, elle se contente d'être une modélisation d'un fait existant. La représentation des connaissances a ainsi pour objectif de fournir des moyens de description et d'identification de faits observés, voire d'ensembles de faits. Cette différence dans les objectifs est capitale car ce sont les objectifs d'un système informatique qui orientent sa sémantique.

Ainsi, d'une part, la définition de la sémantique d'un langage de programmation a pour finalité l'attribution d'une signification à un résultat, liée à la signification des données ayant fourni ce résultat. D'autre part, la définition de la sémantique d'un système de représentation concerne l'attribution d'une interprétation de chaque terme du langage vers un élément, un phénomène, ou un ensemble d'éléments du monde modélisé.

1.3.2 Réseaux sémantiques

Les réseaux sémantiques sont les premiers systèmes de représentation des connaissances à avoir utilisé et mis en œuvre le principe et les mécanismes de l'héritage appliqués à des unités de connaissances. D'abord Ross Quillian [Qui68], puis William Woods [Woo75] et Ronald Brachman [Bra77], se sont attachés à définir un cadre formel pour le développement et l'interprétation des réseaux sémantiques.

Structure d'un réseau sémantique

Un réseau sémantique est constitué d'un ensemble de nœuds étiquetés qui dénotent des concepts du domaine modélisé, et d'un ensemble d'arcs orientés étiquetés traduisant les relations sémantiques entre les concepts dénotés par les nœuds. L'intéressant ici est que chaque concept peut se décrire à partir d'autres concepts et au moyen d'arcs, et les relations n-aires entre nœuds peuvent aussi être décrites à l'aide de cette structure de nœud et au moyen d'arcs indiquant les acteurs de la relation. La description d'un nœud est l'ensemble des arcs qui en partent, que l'on appelle les caractéristiques de ce nœud.

Les réseaux sémantiques, tels qu'ils ont été introduits par Quillian, ont ensuite été étendus. Brachman a introduit la distinction générique/spécifique : les *nœuds génériques* sont des collections de *nœuds spécifiques*. Les relations impliquant un nœud générique lui confèrent un ensemble de caractéristiques génériques, qui seront particularisées par les nœuds spécifiques qui s'y attacheront [Bra77]. Il existe plusieurs niveaux de généricité dans les nœuds, le plus bas étant justement la spécificité, dénotant une individualité dans le domaine modélisé. Le passage d'un niveau à un autre est spécifié par le lien `sorte-de` (parfois appelé `is-a` [Bra83]). N_1 `sorte-de` N_2 signifie que N_1 possède les mêmes caractéristiques que N_2 qu'il peut affiner et compléter : on retrouve ici le principe d'héritage.

La figure 1.4 illustre un petit réseau sémantique où l'on découvre les deux catégories de concepts : `Daniel-Pennac` est un concept spécifique qui est un `Écrivain` (concept générique) particulier. On distingue ainsi deux façons de relier les concepts : par des caractéristiques propres au domaine modélisé (ex. `Édité-par` qui lie les concepts génériques `Livre` et `Éditeur`), ou par le lien de spécialisation `sorte-de` (ex. un `Roman` est une sorte de `Livre`).

Inférences

Le principe de la représentation des connaissances par réseaux sémantiques est ainsi de décrire ces connaissances au moyen des relations qui les lient. Un certain nombre de modes de raisonnement sont alors mis en œuvre, dont les mécanismes s'appuient sur la structure même des réseaux. On retiendra plus particulièrement les méthodes d'inférences par héritage, ainsi que le filtrage. Le filtrage est un mécanisme qui permet de résoudre des problèmes eux aussi décrits sous forme de réseau dans lequel les nœuds représentant des connaissances à inférer sont étiquetés par des variables. Le filtrage retrouve alors les informations recherchées par accès associatif : en mettant

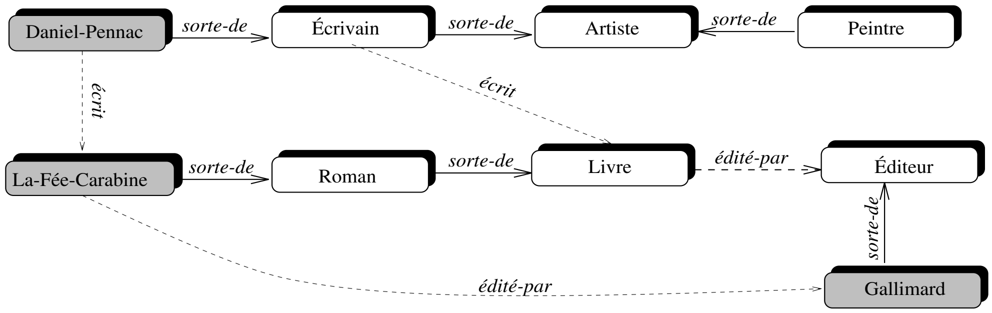


FIG. 1.4 - : Un réseau sémantique : nœuds spécifiques (fond gris) et nœuds génériques (fond blanc).

en correspondance un à un les nœuds du réseau correspondant au problème et ceux d'un sous-réseau dans lequel tout est connu, le système est capable de déterminer des valeurs de variables recherchées.

Conclusion

Les réseaux sémantiques ont été les premiers à illustrer l'intérêt de la notion de description et de l'héritage dans une modélisation et dans la mise en œuvre de mécanismes d'inférence.

Le lien avec la notion d'objet ne peut être nié : on manipule ici des entités ayant une identité (nœuds spécifiques), qui sont décrits, à un niveau de généralité plus élevé, au moyen de relations. Ces relations peuvent être assimilées aux champs qui décrivent les classes d'objets en programmation, et par là même, qui caractérisent les objets. On relève toutefois le fait que la notion de *méthode* n'existe pas dans les réseaux sémantiques. En fait, le "savoir-faire ", la connaissance procédurale, est censée elle aussi être décrite à partir de relations : on préfère indiquer explicitement quels sont les acteurs d'une action activable et son résultat pour chaque collection de données spécifiques, plutôt que dégager un processus opérationnel générique et instanciable. Autrement dit, l'activation d'une procédure, et en conséquence la connaissance de son résultat, est appelée à être décrite en termes de relations, l'exécution d'une telle procédure revenant alors à observer les nœuds impliqués dans les relations. En ce sens, les réseaux sémantiques correspondent à des modèles déclaratifs.

Les unités de connaissance considérées dans les réseaux sémantiques, et d'ailleurs par tous les autres formalismes qui s'en inspirent peu ou prou, sont des *prototypes*. La théorie des *prototypes*, instigatrice du développement de nombreux systèmes de représentation, est née de l'observation suivante : la plupart du temps, l'être humain porte des jugements et, plus généralement, raisonne, sur un élément *typique* d'une catégorie d'objet, c'est-à-dire un élément qui possède les caractéristiques généralement partagées par tous les autres éléments de la catégorie [Ros75]. Il s'agit en fait de l'élément qui, si l'on était capable d'établir une distance de similarité $d(X,Y)$ entre deux individus d'une catégorie C, serait tel que $\forall I_1, I_2 \in C, d(I_1, I) \leq d(I_1, I_2)$ et $d(I_2, I) \leq d(I_1, I_2)$ [RSS73][8]. Les nœuds et arcs d'un réseau sémantique représentent ainsi, de façon plus ou moins précise, la description de prototypes.

Les nœuds d'un réseau sémantique sont des éléments qui ont une signification dans le monde réel, mais ils ne sont pas décrits au même titre que les objets : la description d'un nœud est externe à ce nœud. Autrement dit, si l'on se souvient qu'un *objet* de représentation O est composé d'une dénotation O_m et d'une description O_d, alors on peut affirmer qu'un nœud de réseau sémantique

[8] On retrouve cette distance en catégorisation conceptuelle (classification symbolique) lorsqu'il s'agit d'identifier et de décrire des groupements d'individus similaires [SM86].

correspond à O_m, tandis que O_d n'est pas représenté au sein d'une unité, mais est constitué d'un ensemble de lien entre différents nœuds, O_d est donc distribué. En ce sens, et même si les réseaux sémantiques ont été les premiers à introduire un formalisme pour exprimer la description d'unités observées dans un monde, la description et la dénotation ne sont pas réunies au sein d'une même structure. Ceci rend obsolète, dans ce cadre, notre volonté de distinguer la structure d'un élément et sa dénotation, qui risquent d'être confondues lorsqu'elles sont toujours considérées simultanément.

Pourtant, la structure d'un réseau sémantique n'est pas adaptée à la mise en œuvre efficace de mécanismes d'exploitation, du fait de la complexité du parcours systématique de liens associatifs. On retrouve ainsi, dans d'autres formalismes de représentation, la volonté de (re)penser le langage d'expression des entités de représentation, pour améliorer la capacité des mécanismes de manipulation des descriptions. Comme nous allons le constater dans la section suivante, certains de ces formalismes conservent la séparation entre description et dénotation, tandis que d'autres cherchent une fusion au sein d'une même unité, et l'étude de ces deux catégories nous permettra de mettre en évidence la coopération des deux aspects d'un élément de représentation.

1.3.3 Autres formalismes de représentation

Mis à part les réseaux sémantiques, il existe quatre principaux formalismes de représentation des connaissances qui sont axés sur l'expression de la description d'éléments du monde réel, la dénotation de ces expressions demeurant essentielle au regard de la sémantique de ces formalismes.

Les graphes conceptuels

Les graphes conceptuels [Sow84] sont nés de la nécessité de représenter la langue naturelle et son interprétation. Il s'agit de systèmes logiques, les termes étant interprétés par des mécanismes de déduction proches de ceux rencontrés en logique du premier ordre. Les graphes conceptuels se rapprochent pourtant de la notion d'objet, et plus précisément du formalisme des réseaux sémantiques, puisque John Sowa a introduit un opérateur permettant la transformation de formules logiques en graphes et inversement, par interprétation des quantificateurs et des connecteurs des formules. Un graphe conceptuel est alors un ensemble de nœuds (des concepts et des relations conceptuelles), et des arcs étiquetés qui permettent de connecter les différents types de nœuds.

On retrouve, dans ce formalisme, le principe de représentation déclarative, qui consiste à décrire la connaissance en termes de relations entre entités. En outre, on retrouve les notions de concepts génériques et individuels (les premiers correspondent aux variables rencontrées en logique, les seconds sont des constantes de la logique et dénotent un individu particulier du monde modélisé). La relation de spécialisation, inhérente à l'héritage, est elle aussi présente dans le formalisme des graphes conceptuels. En effet, à chaque nœud représentant un concept est associé un type qui détermine une structure pour ce concept. Les types de concepts sont alors ordonnés en treillis par le sous-typage, défini sur les structures des types. Les concepts d'un graphe sont ainsi ordonnés par l'ordre sur leurs types.

Outre les inférences effectuées sur l'ordre des types de concepts, il existe deux mécanismes de raisonnement, sémantiquement fondés sur la logique sous-jacente. Le *raisonnement exact* réalise des inférences en traitant les graphes comme des définitions. Le *raisonnement plausible*, quant à lui, permet de réaliser des inférences en considérant les graphes comme des descriptions prototypiques de faits et phénomènes, ce qui revient à considérer que les graphes sont incomplets au regard de l'équivalence logique.

Tout comme les réseaux sémantiques, la description d'une entité est partagée avec les autres entités auxquelles elle est liée. On remarque cependant que lorsqu'une description est associée à une entité interprétable dans le monde modélisé, cette description est alors interprétée comme un type, ou plutôt comme un ensemble de valeurs, dans la mesure où aucun comportement spécifique n'est associé à ces valeurs, contrairement aux types de classes des langages de programmation, qui définissent des structures de données et des opérations sur ces données. Mais on relève une volonté de distinguer la structure de descriptions des entités et la dénotation de ces entités : au niveau d'un graphe conceptuel, tous les nœuds s'interprètent dans le monde modélisé, les données de base se situent au niveau des types de ces concepts.

Les langages de *frames*

L'appellation "langages de *frames*" désigne tous ces formalismes de représentation qui sont issus, en grande partie, de la théorie des *frames* de Marvin Minsky [Min75]. Cette théorie est directement issue de celle des prototypes. Dans ces langages, les prototypes sont représentés par des *frames*, ils correspondent à des objets typiques ou à des situations stéréotypes de référence.

Contrairement, d'une part aux réseaux sémantiques dont les concepts ne sont pas décrits mais reliés à d'autres, et d'autre part aux graphes conceptuels qui séparent la description d'un concept (son type) et les relations qu'il entretient avec d'autres concepts, les langages de *frames* considèrent que *les relations qu'un concept entretient avec d'autres font partie de la description de ce concept.* La description d'une entité et sa dénotation (son identité) font maintenant partie de la même unité : le *frame*.

Les systèmes à base de *frames* s'attachent à fournir des langages permettant la description des prototypes. Ainsi, un *frame* modélisant un objet typique est constitué d'un ensemble d'attributs (les caractéristiques du prototype) à chacun desquels est attachée une liste de facettes spécifiant les propriétés de la caractéristique (figure 1.5). Un assemblage de *frames* constitue un système de *frames* ; la structure obtenue n'est pas une hiérarchie à structure unique, et de ce fait la notion d'héritage n'existe pas à proprement parler : il s'agit plus exactement de propagations d'informations.

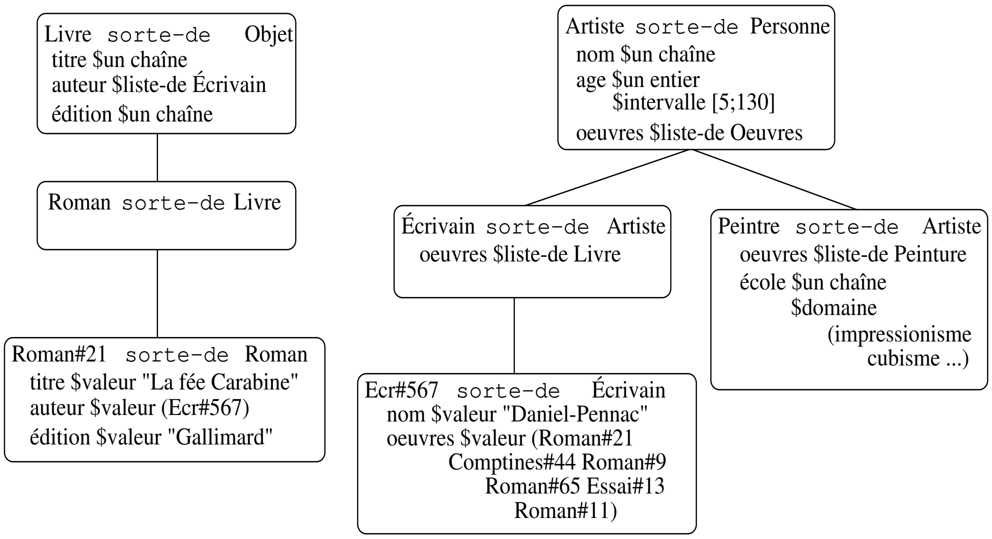


FIG. 1.5 - : Définitions et spécialisations de *frames*. Les relations entre objets font partie de la description interne de chaque *frame*.

Les langages de *frames* se sont enrichis depuis 1975, avec toujours comme constante cette notion de description prototypique. En particulier, la notion de spécialisation a été rapidement introduite, menant à la définition d'un ordre sur les *frames*: un sous-*frame* hérite les caractéristiques de ses pères, de la même façon que dans les langages à objets, une classe hérite les définitions de sa sur-classe. Il s'agit d'un héritage dynamique, car dans les systèmes de représentation, les bases de connaissances évoluent sans cesse, ne serait-ce que par l'apport de connaissances inférées: le système doit donc favoriser cette évolution. Contrairement aux réseaux sémantiques, il n'y a pas, dans la théorie des frames, de distinction entre concepts *individuel* et *générique* (ni syntaxique, ni même conceptuelle). Il y a essentiellement cette notion d'affinement des descriptions, qui peut aller jusqu'à la spécification de concepts ne pouvant plus être spécialisés (mais seulement complétés par la donnée de nouveaux attributs).

Les mécanismes d'exploitation des connaissances représentées sont particulièrement nombreux. Tout d'abord, il est intéressant de préciser que la spécification des attributs d'un *frame* à l'aide de facettes consiste non seulement à décrire la nature de la caractéristique représentée par l'attribut, mais aussi à préciser des moyens de calculs de sa valeur associée et des moyens d'utilisation dans certains contextes précis. On parle généralement de facettes *descriptives*, *inférentielles* et *comportementales* (ces dernières sont aussi appelées *réflexes*, car elles réagissent à certains évènements sur l'attribut, tels que des demandes de lecture ou d'écriture).

Les deux mécanismes classiques d'inférences proposés par tout langage de *frames* sont le filtrage et la classification.

- le filtrage est une opération qui consiste à retrouver un ensemble de *frames* d'un système qui vérifient certains critères portant sur les attributs. Il est réalisé par des tests d'appariemment[9] entre un *frame* factice représentant les critères de sélection, et les *frames* potentiellement candidats.
- la classification consiste à déterminer la position d'un *frame* dans une hiérarchie ordonnée par la spécialisation des frames. L'ordre induit par la spécialisation est généralement fondé sur les descriptions, de la même façon que le sous-typage considéré dans les graphes conceptuels.

Si le filtrage peut s'apparenter à la notion de requête dans les systèmes de gestion de bases de données, le fait qu'il soit attaché explicitement à un attribut par une facette particulière, comme dans SHIRKA [Rec89], lui confère un véritable statut inférentiel puisque le résultat d'un filtre est considéré comme la valeur d'un attribut. Pour le lecteur intéressé par les différents mécanismes de filtrages répertoriés en représentation par objets, une synthèse est disponible dans [Dek94].

La classification, quant à elle, est un véritable mécanisme d'inférence, sur lequel nous reviendrons plus précisément dans la section 1.4. Amedeo Napoli, dans sa thèse d'état, a étudié tous les tenants et aboutissants du raisonnement par classification [Nap92a].

Notons, parmi tous les langages de *frames* existants (KRL [BW77], FRL [RG77], ou encore KANDOR [PS84]) le statut particulier de SHIRKA [Rec85] [Rec88]. SHIRKA est en effet le seul langage apparenté aux *frames* qui réalise une distinction nette (syntaxique et sémantique) entre concepts *générique* et *individuel*. Réutilisant la terminologie des langages à objets, SHIRKA manipule différemment *classes* et *instances*. Les classes ont pour but de regrouper des instances (représentation d'individus du monde modélisé) à partir de la donnée des caractéristiques communes de ces instances. En outre, l'ensemble des attributs d'une classe a la prétention de constituer une *définition* des objets regroupés. Cette distinction entre classes et instances dans SHIRKA est comparable à la

[9] L'appariemment entre deux *frames* consiste à confronter, attribut par attribut, la compatibilité de leurs descriptions.

différence (tout relative) que l'on peut établir entre une valeur et son type, lorsque l'on considère une sémantique ensembliste. Ainsi, à l'instar des langages de programmation orientés-objet, si une classe peut être vue comme génératrice d'instances, une instance ne peut rien créer à partir de sa description. Le premier attachement d'une instance à une classe confère à l'instance son identité, sans que la classe ne demeure maître de cette identité, ainsi qu'un statut au sein de la collectivité des instances de classes d'une même famille[10]. Les instances en SHIRKA peuvent migrer, c'est-à-dire qu'elles peuvent changer de classe d'appartenance au gré de l'évolution de leurs caractéristiques, sans pour autant que leur identité soit modifiée. C'est une différence majeure d'avec les langages de programmation orientés-objet, pour lesquels une classe instigatrice de la création d'une instance, possède cette instance durant toute sa durée de vie. Cette restriction est tout simplement due au fait que l'interprétation d'une instance, et surtout son unicité, passe par l'interprétation de sa classe de création. Nous reviendrons plus en détail sur ce phénomène dans le dernier chapitre de ce mémoire.

Les langages de *frames* sont les premiers formalismes de représentation des connaissances à unifier, au sein d'une même structure, l'identité d'un élément du monde modélisé et sa description. C'est ce qui vaut à ces langages d'être parfois assimilés à des modèles à objets.

Les langages hybrides

Il existe des langages qui intègrent et font coopérer plusieurs formalismes de représentation des connaissances, dans l'objectif de permettre l'expression et l'exploitation coopérative de connaissances hétérogènes. Ces langages sont qualifiés, à juste titre, de *langages hybrides*. Chacun de ces langages a ses propres particularités, du fait de leur préférence marquée pour l'un ou l'autre formalisme. La qualité de ces langages se mesure à la pureté et à l'homogénéité de leur définition sémantique, et en particulier, à la clarté de la frontière, voire de la coopération, entre les différents formalismes considérés.

Il existe toute une panoplie de langages hybrides, parmi lesquels nous retiendrons les plus complets, à savoir LOOPS, MERING, Y3 (comprenant YAFOOL), OBJLOG ou encore LORE. Une composante qui se retrouve presque unanimement dans ces systèmes est la *reflexivité*. Un système reflexif est un système qui contient une structure d'auto-représentation, c'est-à-dire qui permet au système de représenter la connaissance qu'il a de lui-même [Mae87]. Cette propriété est un atout en ce qui concerne la définition sémantique du système qui se doit d'être homogène, ou tout au moins cohérente, entre les différents formalismes. La plupart du temps, les systèmes adoptent une auto-représentation par objets.

LOOPS [BS83] est l'intégration de la programmation par objets, de la programmation logique et de la représentation à base de *frames*. Une de ses particularités, qui a été ensuite reprise dans la plupart de ce genre de systèmes, est d'avoir appliqué le mode de spécification des attributs de *frames* à l'aide de facettes, aux variables d'instances et de classes. En particulier, le mécanisme de démons, par l'existence de reflexes attachés aux attributs, est repris et permet une programmation en partie dirigée par les accès. Les classes manipulées sont des classes telles que considérées en programmation, à savoir des moules de génération d'instances, qui attribuent à leurs instances une structure et une identité figées. La puissance et la fiabilité de ce langage sont toutes relatives, mais on ne peut lui enlever le mérite d'avoir inspiré le développement de nombreux autres. Le système KEE [FK85] a repris les principales fonctionnalités de LOOPS, complétées en particulier par un système de maintien de la cohérence des bases de connaissances de type ATMS.

[10] Une famille en SHIRKA est une hiérarchie multiple de classes, elle dénote une ensemble de faits du monde modélisé, comparables dans l'absolu. Une base de connaissances SHIRKA est la donnée d'un ensemble de familles instanciées.

Le système MERING [Fer84] est fondé sur les *frames*, auxquels (et à l'intérieur desquels) il permet d'associer un comportement, décrit par des méthodes, au même titre que le savoir-faire d'un ensemble d'objets est décrit dans la classe regroupant ces objets de programmation. Tout comme LOOPS, MERING est réflexif, toutes ses composantes (méthodes, attributs, règles...) sont sémantiquement et génériquement définies par des classes d'objets.

De même que les deux systèmes précédents dont il s'est inspiré, YAFOOL [Duc88], le moteur de Y3, est complètement uniforme et réflexif, et par là même extensible et personnalisable. Il reprend les principes généraux de LOOPS et MERING, qu'il complète par la notion importante de *relations*. YAFOOL permet la définition de liens particuliers et dépendants du domaine d'application, dont il détermine automatiquement la nature de l'inverse. Ces liens impliquent des prototypes par le biais de leurs attributs. Notons que c'est sur la base de YAFOOL que Roland Ducournau et Michel Habib ont effectué leurs recherches sur la résolution de l'héritage multiple fondée sur la linéarisation de l'ordre sous-jacent à la relation d'héritage [DH89]. Le système hybride Y3 comprend en outre une composante logique (YAFLOG) ainsi ainsi qu'un véritable environnement de programmation (YAFEN), définis en termes de la couche objet du noyau de YAFOOL.

OBJLOG [Dug88] est un système intégrant la représentation par les *frames*, la programmation par objets et la déduction logique. Il s'agit d'un modèle réalisant, tout comme SHIRKA, une distinction nette entre les statuts respectifs des classes et des instances. Les travaux menés dans le cadre d'OBJLOG, portant sur l'héritage multiple et la résolution de conflits [Dug89], ont abouti au développement de la notion d'*unification sémantique* : lorsqu'un attribut est hérité de deux classes différentes ayant des spécifications incompatibles pour cet attribut, la définition de cet attribut devient la réunion des deux héritées, chaque choix étant associé à un point de vue de pertinence (tout simplement la classe – ou le chemin – d'où provient l'héritage), et ponctuée de la donnée d'une préférence par défaut. Il s'agit en fait de la représentation, au sein même des objets, d'une préférence de parcours de la hiérarchie d'héritage, en fonction de l'environnement dans lequel a été activé ce parcours.

Le langage LORE [Cas87] permet de définir des ensembles d'objets décrits à l'aide d'attributs et de méthodes, ces ensembles étant ordonnés par l'inclusion ensembliste. La description des caractéristiques des objets se fait en termes d'ensembles d'objets reliés par des connecteurs. Couplé à l'environnement de programmation logique MARIE, LORE constitue un environnement pour la représentation des connaissances dont les fonctionnalités, d'un point de vue exploitation, sont très complètes. En particulier, le couplage des relations entre objets et de la programmation logique offre un cadre adéquat à la résolution de problèmes de satisfaction de contraintes.

Les systèmes hybrides favorisent la représentation, à un même niveau d'interprétation, et parfois en termes d'un même langage, de connaissances certes hétérogènes, mais ils prennent le risque que ces connaissances représentées n'appartiennent pas à un même niveau de modélisation, alors qu'elles sont manipulées selon une sémantique commune.

Les langages terminologiques

Le troisième formalisme de représentation des connaissances, manipulant des notions proches de celle d'objet, est celui initialement appelé *langage terminologique*, et désigné actuellement par le terme *logiques de description*[11]. Il s'agit sans conteste du plus répandu et, de ce fait, celui qui est à l'origine du plus grand nombre d'études, tant conceptuelles que techniques (même si ce formalisme, du point de vue de la popularité, est talonné par celui des graphes conceptuels). Il s'agit

[11] En anglais : terminological languages et description logics.

d'une catégorie de langages elle aussi issue de la théorie des *frames*, qu'elle enrichit de principes rencontrés dans les réseaux sémantiques.

Du fait de la complexité de ces systèmes, due essentiellement au nombre important de problèmes de tous bords qu'ils tentent de résoudre sans remettre en cause leur homogénéité dans la représentation et le traitement de la connaissance, nous réservons la section suivante (1.4) à la présentation de ces systèmes.

Conclusion

Les réseaux sémantiques et la théorie des *frames* ont montré plusieurs fonctionnalités intéressantes pour les systèmes de représentation des connaissances, qui sont considérées à divers degrés par tout système de ce type. On retiendra bien évidemment la notion d'unité de connaissance, désignée, d'un formalisme à l'autre, par les termes *concept, classe, prototype, objet, individuel, modèle*, etc., éventuellement disposés en deux niveaux de généralité (classes et instances). Ces termes sont généralement ordonnés par une relation apparentée à la spécialisation et sont, dans une certaine mesure, décrits en termes des relations qu'ils entretiennent avec d'autres, et/ou à partir de données primitives. Tous ces systèmes ont pour objectif de permettre la représentation des relations sémantiques existant entre des entités à dénotation dans le monde modélisé, et certains d'entre eux vont jusqu'à fournir un langage pour la description de ces entités.

Nous retenons alors deux perspectives de synthèse, en distinguant :

– les langages de frames (et comme nous le verrons les langages terminologiques) qui cherchent à représenter, *au sein de l'objet*, la description de l'objet, incluant les relations qui lient l'objet aux autres,

– et les autres formalismes présentés ici, qui séparent la description de l'objet (lorsqu'il est possible de la donner), et la spécification de relations annexes entre ces objets.

Nous avons tendance à préférer la première vague de systèmes, tout d'abord pour une question de lisibilité (tout est dans l'objet donc directement accessible), et de ce fait pour une meilleure explicabilité des mécanismes opérant sur les représentations.

Ce principe des langages de *frames* (le "tout dans l'unité") est commun au fondement des langages de programmation par objets. On retiendra cependant une différence essentielle, à savoir que :

- dans les langages de *frames*, la sémantique des objets est dénotationnelle, à valeur dans le monde modélisé, et la définition et la gestion de l'identité d'un objet sont confiées au modèle de représentation (l'objet peut donc exister individuellement, indépendamment de toute appartenance à un groupe),

- tandis que dans les langages de programmation orientés-objet, la sémantique des objets est définie par la classe qui les a créés, qui est aussi maître de leur identité. L'objet n'existe que parce qu'il appartient à une classe.

Autrement dit, dans les langages de programmation, l'identité d'un objet est indissociable de sa description, tandis que dans les langages de représentation des connaissances, l'objet et sa dénotation peuvent être considérés indépendamment de la description de l'objet, et plus généralement, indépendamment de toute structure de données. Cela signifie en particulier qu'en représentation des connaissances, il serait pertinent de distinguer explicitement les mécanismes de manipulation des structures des objets, de ceux qui traitent de la signification des objets dans le monde modélisé.

Les langages terminologiques, même s'ils ne sont pas des modèles à objets, ont développé nombre d'algorithmes de manipulation des descriptions d'entités. Par ailleurs, ils offrent, pour la plupart, un cadre théorique permettant d'interpréter les descriptions, et donc les résultats des inférences sur ces descriptions, dans le monde modélisé, et ce au moyen d'assertions explicites. Cette distinction a mené à la distinction entre deux sémantiques, dites *intensionnelle* et *extensionnelle*, la première concernant l'interprétation des structures de descriptions des termes, tandis que la seconde s'attache à l'interprétation des entités de représentation dans le monde modélisé. Cette distinction est fondamentale au regard de notre problématique, et nous verrons que cette double sémantique peut elle aussi s'adapter aux modèles de représentation des connaissances par objets.

1.4 Les langages terminologiques

Les langages terminologiques[12] qui se situent à la croisée des chemins des réseaux sémantiques et des langages de *frames*, font référence à un formalisme de représentation des connaissances des plus appréciés. L'ancêtre de cette grande famille est KL-ONE, conçu par Ronald Brachman en 1978 [BS85], qui a établi les fondations de tout un courant de recherches et développements. En particulier, depuis qu'il a été muni d'un algorithme de classification et d'une sémantique dénotationnelle [SL83], KL-ONE est devenu un modèle.

Le principe de base de tous ces systèmes est qu'ils offrent des langages de représentation des connaissances intégrant la terminologie du domaine, le système structurant les termes dans des réseaux d'héritage. Les termes peuvent être décrits et/ou définis à partir d'autres termes, au moyen de relations instanciables. L'ordre sur ces termes est appelé la relation de *subsomption*, son extension peut être “assertée” ou inférée.

Comme nous le verrons par la suite, il existe un véritable dilemme quant au bon dosage à appliquer entre l'expressivité de ces langages (leur capacité à fournir des connecteurs entre termes permettant d'exprimer un nombre important de types de relations), la complétude de leurs inférences et la complexité en temps de ces inférences [DP91].

1.4.1 Composantes de base

A-Box et T-Box : assertions et descriptions

Bien qu'introduite parallèlement à la définition de KL-ONE par KRYPTON [BFL83], la notion de A-Box est reprise par la plupart des langages terminologiques, et nous estimons que sa distinction d'avec la T-Box est fondamentale. Les différences et liens entretenus entre T-Box et A-Box ont été étudié et formalisés par Bernhard Nebel [Neb90].

A-Box et T-Box sont deux *langages de représentation* coopératifs, qui servent chacun des objectifs particuliers.

– La T-Box (*terminological box*) est un langage qui permet de décrire des termes en fonction

[12] Ou encore *logiques terminologiques, langages à subsomption de termes, langages de descriptions de termes, systèmes basés sur la classification, famille de* KL-ONE.

d'autres, à partir de relations (les rôles) sur lesquels portent des contraintes, exprimées par des *restricteurs* comparables aux facettes des langages de *frames*. La T-Box est réservée à l'expression de termes correspondant à des ensembles d'individus.

– La A-Box (*assertional box*) permet d'établir des formules de la logique du premier ordre dont les prédicats sont des expressions de la T-Box. Il s'agit là de véritables opérateurs de constructions de phrases, dont l'existence a pour but d'autoriser la représentation de connaissances incomplètes [BL82]. La A-Box est réservée à la représentation d'individus.

En d'autres termes, la T-Box est le support de descriptions structurées et ordonnées dans une taxonomie, tandis que la A-Box permet à ces descriptions d'être utilisées afin de caractériser, en quelque sorte, leur signification au regard du domaine modélisé. Cette notion de A-Box est donc fondamentale quand il s'agit de relativiser toutes les inférences qui peuvent être faites sur des descriptions, donc des structures de données.

Les systèmes terminologiques ayant repris ces deux langages de représentation[13] considèrent que la A-Box permet, entre autres, d'exprimer des liens d'appartenance entre des faits individuels et des regroupements d'individus décrits en termes de la T-Box. Cela est comparable, dans une certaine mesure, à l'instanciation, car lorsqu'un individu est lié à un *frame* générique de par la A-Box, il acquiert automatiquement une identité, et implicitement une description issue de celle du *frame*. Bien entendu, la A-Box n'est pas que cela : elle a la pouvoir aussi d'exprimer des correspondances logiques entre individus et *frames*, indépendamment de leurs descriptions.

Concepts, rôles et éléments individuels

L'unité de représentation dans les langages terminologiques est le *frame*, dénotant un ensemble d'individus, dont les descriptions dans la T-Box sont reliées, entre autres, à l'aide de relations. Les *concept*, *rôle* et *éléments individuels*[14] forment la trinité des langages terminologiques, il s'agit des trois composantes formelles de ce formalisme de représentation des connaissances.

- Un *concept* regroupe un ensemble de faits dont il donne une description (on parle alors de concept primitif), ou une définition (on parle dans ce cas de concept défini). Les concepts correspondent à des prédicats unaires en logique du premier ordre, et s'appliquent donc à un seul individu à la fois. La description d'un concept est donnée par la composition de contraintes posées sur d'autres concepts, interprétées comme des relations.

- Un *rôle* caractérise une relation entre deux concepts. Il s'agit d'un prédicat binaire, qui, lorsqu'il est instancié, permet de lier deux individus. Les rôles font partie intégrante de la description (ou définition) d'un concept. Notons la différence entre les notions de *rôle* et d'*attribut* : si le premier met en relation deux concepts, le second, quant à lui, est un prédicat unaire dans le sens où il met en relation un concept et une donnée primitive. La différence à faire entre les notions de *concept*, *attribut* et *rôle*, est relative aux domaines d'interprétations de leurs arguments [Gua92].

- Un *individuel* est une construction simple qui a pour seul objectif de dénoter un individu particulier dans le monde modélisé. Lorsqu'attaché à un concept générique au sein de la A-Box, un individuel se voit attribuer la description de ce concept, qu'il peut personnaliser, en s'impliquant dans des rôles dont il devient un argument, l'autre argument pouvant être un autre

[13] Les autres ne possédant qu'une composante terminologique.

[14] que nous désignerons par la suite comme des *individuels*, tout en étant conscients de l'abus de langage.

individuel. Les individuels peuvent ainsi être décrits au même titre que les concepts définis, on parle alors de *concepts individuels*, et ils sont traités dans la T-Box exactement comme des concepts génériques. Pour résumer, la connaissance portée sur un individu est répartie entre la A-Box et la T-Box, sous forme de descriptions exprimées en termes de deux langages différents mais coopératifs. Une description de la T-Box est pourtant toujours interprétée comme un ensemble, la notion de *concept individuel* ne relève alors que d'une technique qui consiste à permettre l'application de mécanismes d'inférences, définis sur les concepts, à des individuels pour lesquels il est possible de calculer une description. La description d'un individuel dans la T-Box est soit la description d'un concept défini qui dénote un singleton, soit l'intersection des descriptions des concepts définis auxquels l'individuel est dit appartenir dans la A-Box.

Notons que les langages terminologiques, contrairement à la plupart des langages de *frames*, mais à l'instar des réseaux sémantiques, affichent une distinction entre les notions d'individualité et de généricité des descriptions. Nous verrons cependant que, contrairement aux langages de programmation, un concept générique n'est pas maître de l'existence d'un individuel.

Syntaxe et sémantique de la T-Box

Un des atouts des langages terminologiques est de posséder une syntaxe simple et évolutive, munie d'une sémantique dénotationnelle fort intuitive. Le langage de représentation d'une T-Box est, à l'opposé des langages à objets, peu structuré, on n'y distingue pas de *niveau de description*, si ce n'est la distinction entre expressions de rôles et de concepts.

Nous donnons dans la table 1.1 la syntaxe des termes du langage de représentation d'une T-Box habituelle.

C	$\longrightarrow$	CP	(concept primitif)
		(and $C_1 \cdots C_n$) |	(conjonction)
		(all R C) |	(quantification universelle)
		(atleast $<$ *entier* $>$ R) |	(restriction de cardinalité)
		(atmost $<$ *entier* $>$ R) |	(restriction de cardinalité)
R	$\longrightarrow$	RP |	(rôle primitif)
		(in R C) |	(restriction de rôle)
		(androle $R_1 \cdots R_n$)	(conjonction de rôles)

TAB. 1.1 - : Syntaxe d'un langage terminologique.

Les concepts peuvent être nommés, les langages terminologiques font l'hypothèse de l'unicité des noms. Le nom (identificateur) d'un concept, pris dans un ensemble de symboles A, est lié à sa description par trois types de connecteurs :

- $A \Rightarrow C$ signifie que le concept de nom A est *décrit* par l'expression C, il s'agit dans ce cas d'un concept primitif,
- $A \Leftrightarrow C$ signifie que le concept de nom A est *défini* par l'expression C, A est alors un concept défini,
- $A \rightarrow C$ signifie que le concept de nom A est décrit par l'expression C, il s'agit d'un concept individuel.

De même, les rôles et attributs peuvent être nommés, on distingue alors $A \mapsto R$ qui indique que A est un rôle décrit par R, et $A \stackrel{\cdot}{\mapsto} R$ qui signifie que A est un attribut décrit par R.

La sémantique dénotationnelle est définie par une fonction $\|.\|^E$ à valeurs dans un domaine d'interprétation $\mathcal{D}$ lorsqu'il s'agit de concepts, et à valeurs dans $\mathcal{D} \times \mathcal{D}$ quand il s'agit de rôles. La sémantique dénotationnelle dont est munie la T-Box est donnée par la table 1.2. La fonction d'interprétation est telle que $\forall C, \|C\|^E \subseteq \mathcal{D}$ et $\forall R, \|R\|^E \subseteq \mathcal{D} \times \mathcal{D}$. On note $\|S\|$ la cardinalité de l'ensemble S.

$\|(\mathit{and}\ C_1 \cdots C_n)\|^E$	=	$\{x \in \mathcal{D} \mid x \in \|C_1\|^E \wedge \cdots \wedge x \in \|C_n\|^E\}$
$\|(\mathit{all}\ R\ C)\|^E$	=	$\{x \in \mathcal{D} \mid (\forall y) : (x \times y) \in \|R\|^E \Rightarrow y \in \|C\|^E\}$
$\|(\mathit{atleast}\ n\ R)\|^E$	=	$\{x \in \mathcal{D} \mid \ \|\{y \in \mathcal{D} \mid (x \times y) \in \|R\|^E\}\| \geq n\}$
$\|(\mathit{atmost}\ n\ R)\|^E$	=	$\{x \in \mathcal{D} \mid \ \|\{y \in \mathcal{D} \mid (x \times y) \in \|R\|^E\}\| \leq n\}$
$\|(\mathit{in}\ R\ C)\|^E$	=	$\{(x \times y) \in \mathcal{D} \times \mathcal{D} \mid (x \times y) \in \|R\|^E \wedge y \in \|C\|^E\}$
$\|(\mathit{androle}\ R_1 \cdots R_n)\|^E$	=	$\{(x \times y) \in \mathcal{D} \times \mathcal{D} \mid (x \times y) \in \|R_1\|^E \wedge \cdots \wedge (x \times y) \in \|R_n\|^E\}$

TAB. 1.2 - : Sémantique dénotationnelle d'un langage terminologique.

À vrai dire, aucun langage terminologique ne présente la même richesse, et cela pour des raisons d'efficacité et de complétude dans les inférences. Beaucoup de recherches menées autour de ces systèmes sont motivées par l'étude des conséquences de l'introduction de certains connecteurs que nous n'avons pas présentés ici, tels que la disjonction ou la négation de concepts.

Intension et extension, définition et description

L'extension d'un prédicat est l'explicitation des données satisfaisant ce prédicat. On dit alors qu'un prédicat est défini *en extension* s'il est exhaustivement défini par une énumération. Par opposition, on dit qu'un prédicat est défini *en intension* lorsque c'est une expression qui exprime les conditions d'appartenance à son extension. Par exemple, $P = \{(1\ 2)\ (1\ 3)\ (2\ 3)\}$ est ici défini en extension, sa définition en intension pouvant être $P = \{(x\ y) \mid x \in [1..3], y \in [2..3], x < y\}$. On va fréquemment jusqu'à dire que l'intension d'un concept est l'ensemble des individus potentiels qui satisfont la description du concept, tandis que l'extension d'un concept est l'ensemble des individus qu'il (son identificateur) dénote dans le monde modélisé. Il est toutefois important de lever ici l'incorrection d'une telle définition de l'intension : l'intension doit pouvoir s'interpréter dans tous les mondes possibles, tandis que l'extension ne s'interprète que dans le monde modélisé. Immédiatement, nous constatons que certains prédicats ne peuvent être spécifiés qu'en intension, notamment lorsque leur extension est infinie.

Ces définitions s'appliquent aussi au niveau des concepts et rôles, qui sont interprétés comme des prédicats en logique du premier ordre. Par analogie, l'intension d'un concept (d'un rôle) est sa description, tandis que l'extension d'un concept (d'un rôle) est l'ensemble des (couples d') individuels qui peuvent être attachés au concept (au rôle), et qui ont une signification. En ce sens, une correspondance peut être faite, d'une part entre la connaissance intensionnelle (connaissance portant sur les descriptions) et la T-Box, et d'autre part entre la connaissance extensionnelle (portant sur les individus) et la A-Box [OK89] [Woo91].

Établir une égalité (ou tout au moins l'équivalence dans un modèle) entre l'intension et l'extension d'un concept, reviendrait à montrer que la description du concept est en réalité une définition, c'est à dire qu'elle exprime une condition nécessaire et suffisante pour l'appartenance à son extension. Une simple description ne correspond qu'à une condition nécessaire d'appartenance.

Les langages terminologiques considèrent que les concepts *primitifs*, lorsqu'il est permis de leur associer des rôles, et donc une description, ne sont justement que des *descriptions*, c'est-à-dire que leur intension ne représente qu'une condition nécessaire d'appartenance à leur extension. les concepts *définis* sont, quant à eux, des *définitions*, c'est-à-dire que leur intension correspond exactement à leur extension. Pourtant, nous pouvons imaginer qu'il est très difficile, voire impossible, de donner une définition intensionnelle exacte d'un ensemble d'individus pris dans un univers modélisé. La capacité des langages terminologiques à offrir la possibilité d'écrire des définitions en termes de rôles provient du fait que les concepts définis le sont, d'une part à partir d'une description, mais aussi et surtout grâce à une référence à un ou plusieurs concepts primitifs. Par exemple, les langages terminologiques permettent d'exprimer le fait suivant : `Les` *`triangles`* `sont des` *`polygones`* `ayant exactement 3` *`côtés`*, où *`polygones`* est l'identificateur d'un concept primitif (on ne sait pas lui donner une définition, ou tout au moins on ne veut pas en fixer une puisqu'il existe plusieurs moyens de le faire), et *`triangles`* est un concept ici défini. Ainsi, pour être un triangle, il faut et il suffit d'être un polygone et de posséder exactement trois côtés [BS85].

Les langages terminologiques sont les seuls, parmi ceux assimilés aux langages de *frames*, à considérer les descriptions de concepts comme des définitions. Considérer l'équivalence entre intension et extension des concepts est un avantage non négligeable, car cela permet aux inférences d'un système de représentation de n'opérer que sur des structures de données, à savoir les définitions des concepts. Cela permet, de ce fait, de faire l'hypothèse de monde ouvert au niveau de la A-Box (tout ce qui n'y est pas dit n'est pas forcément faux, et en particulier, si un fait non établi dans la A-Box l'est dans la T-Box, alors il est considéré comme vrai). Nous verrons que ce lien d'un concept défini vers un concept primitif est un lien de subsomption asserté.

La subsomption

Nous n'avons pas encore parlé de la relation de *subsomption*, qui est une caractéristique incontournable des langages terminologiques. La subsomption est un ordre sur les concepts et les rôles, qui, d'une part, relève d'une technique d'identification, de représentation et d'organisation de la connaissance, et d'autre part, qui est à base du principal mécanisme d'inférence des langages terminologiques, à savoir la classification.

La relation de subsomption, de par son importance, mérite que nous nous y attardions un peu plus, c'est pourquoi la section suivante y est entièrement consacrée.

1.4.2 Subsomption

La subsomption est une relation entre deux concepts, ou deux rôles, ou entre un individu et un rôle, qui, d'une certaine façon, est similaire à la relation de spécialisation. Il s'agit d'une relation d'ordre partiel, utilisée pour organiser concepts, rôles et individuels. Par la suite, nous noterons la relation de subsomption $C_1 \preceq_S C_2$, et nous dirons que C_2 (le subsumant) subsume C_1 (le subsumé)[15]. William Woods a dégagé cinq interprétations de la subsomption [Woo91], mais nous

[15] Par la suite, nous développons la subsomption entre concepts, mais les mêmes remarques et définitions s'appliquent à la subsomption entre rôles.

ne retiendrons que les deux principales :

- la subsomption *extensionnelle*, s'établit, comme son nom l'indique, entre les extensions des concepts ; $C_1 \preceq_S C_2$ si et seulement si l'extension de C_1 est incluse dans celle de C_2,

- la subsomption *intensionnelle* s'établit entre les descriptions des concepts ; $C_1 \preceq_S C_2$ si et seulement si la description de C_2 contient celle de C_1. Lorsque l'on veut pouvoir comparer directement ces deux interprétations de la subsomption, la subsomption intensionnelle considère aussi des ensembles d'individus : $C_1 \preceq_S C_2$ si et seulement si tous les individus décrits par C_1 le sont aussi par C_2.

La définition de la subsomption en logique du premier ordre fournit un bon critère de correction de la subsomption. En logique classique, une clause C subsume une clause D si et seulement si il existe une substitution θ telle que $C\theta \subseteq D$. Par ailleurs, si C subsume D, alors C implique logiquement D [Got87].

Les liens de subsomption peuvent être inférés (à partir des descriptions en termes de rôles) ou exister explicitement, soit par le lien `and` présent dans la description du concept subsumé, soit par une assertion dans la A-Box (*axiomatic subsumption).* L'inférence de liens de subsomption, fondée sur les intensions des concepts, et donc sur leurs structures, est à l'origine de nombreuses études qui visent, d'une part à rendre ce calcul décidable, et d'autre part, à réduire le coût de la complexité du calcul, et ce, tout en préservant la complétude des inférences.

Algorithmes de subsomption

Un point commun entre tous les systèmes terminologiques est l'algorithme d'inférence de lien de subsomption, plus communément appelé *algorithme de subsomption*, qu'ils cherchent tous à élaborer. Cet algorithme a pour but de déterminer un éventuel lien de subsomption entre deux concepts donnés. Bien évidemment, cet algorithme est généralement mis en œuvre sur les structures des concepts, c'est-à-dire par confrontation des restrictions portées sur les rôles, au regard de la sémantique dénotationelle.

Le premier algorithme de subsomption a été proposé par Thomas Lipkis pour KL-ONE [Lip82] [SL83]. Il consistait simplement à comparer un à un les composants d'un concept avec ceux de l'autre, en incluant les comparaisons de composants hérités. Ces comparaisons étaient guidées par la sémantique accordée à chacun des restricteurs de rôles et d'attributs. La plupart des autres algorithmes qui ont suivi celui de Lipkis sont du même type, avec quelques variantes, mis à part l'algorithme développé pour le langage KRIS [BH91] qui réalise un test de subsomption fondé sur les extensions des concepts, en s'appuyant sur la théorie des modèles.

Citons tout de même l'algorithme complètement décrit par Alexander Borgida et Ronald Brachman pour le système PROTODL, qui réalise, avant toute comparaison, la normalisation des concepts, à savoir la réduction de leur description vers une forme normale [BB92b]. Cette phase de normalisation avait été préalablement introduite par Nebel [Neb90], mais sa particularité dans PROTODL est d'être répartie au niveau de la définition sémantique des restricteurs de rôles du langage, et ce dans le souci de faire de PROTODL un langage extensible au regard de l'adjonction de restricteurs.

Efficacité et complétude de la subsomption

Les problèmes liés à l'efficacité, la calculabilité, la correction et la complétude de la subsomption, ont fait couler beaucoup d'encre.

L'algorithme de subsomption qu'ont conçu Lipkis et Brachman pour KL-ONE était certes très efficace (polynomial), mais, malgré les espérances de ses auteurs, s'est révélé être incomplet, suite à la formalisation dénotationnelle de KL-ONE [SI83]. Plus tard, Manfred Schmidt-Schauß a d'ailleurs prouvé l'indécidabilité de l'inférence de subsomption dans KL-ONE [SS89].

Un grand nombre d'études ont permis de cerner les propriétés de l'algorithme de subsomption en fonction de l'expressivité des langages[16]. Tout d'abord, Hector Levesque et Ronald Brachman ont montré que dans un langage terminologique plutôt simple, le test de subsomption est en réalité un problème co-NP-complet [LB85]. La logique du langage étudié était pourtant un sous-ensemble des logiques considérées dans KL-ONE, KL-TWO [Vil85] et dans NIKL [Sch85], ce qui amène la subsomption dans ces logiques à être NP-difficile. En outre, quelques temps après, Bernhard Nebel a prouvé que la subsomption dans un sous-langage de ceux des systèmes KANDOR [PS84], BACK [Pel91], KL-ONE, KL-TWO et NIKL, est NP-difficile. La forte complexité de l'inférence de liens de subsomption semble donc être le prix à payer pour sa complétude, et ce dans un langage moyennement expressif. Notons en outre que les origines des incomplétudes ne sont toujours pas encore toutes connues.

Pour pallier efficacement les problèmes d'incomplétude liés à l'expressivité des logiques de ces langages, certains d'entre eux ont mis au point des techniques annexes d'inférences, telles que des règles de déduction, propres aux domaines d'application, qui doivent en conséquence être écrites par l'utilisateur. C'est une solution adoptée dans BACK, LOOM et CLASSIC, qui n'est pourtant pas des plus satisfaisantes.

Au delà de ces études menées autour de la complexité de la subsomption, la *décidabilité* de ce type d'inférence a été remise en cause dans un certain nombre de langages tels que NIKL [PS89b] ou $\mathcal{U}$ [Sch88], ce dernier ayant montré que l'introduction simultanée de la conjonction de rôles et de la négation est une des raisons de l'indécidabilité. Dernièrement, la tendance est d'étudier des langages terminologiques dans lesquels la subsomption est (au moins) décidable, ce qui permet petit-à-petit de cerner les causes de l'indécidabilité [BDS93].

Tous les travaux qui ont porté sur l'étude de la subsomption convergent sur un point : plus un langage (une logique) terminologique est expressif, plus l'inférence de liens de subsomption est coûteuse, jusqu'à devenir indécidable [DLNN91a] [DLNN91b]. On notera en particulier le travail de synthèse qu'ont effectué Jochen Heinsohn et ses co-auteurs, qui concerne une analyse expérimentale de la puissance et de la capacité de tous les algorithmes de subsomption rencontrés dans différents systèmes (et donc en présence de divers degrés d'expressivité), et ce pour plusieurs classes de problèmes connus [HKNP92]. Face à cet épineux, mais incontournable, problème qu'est le calcul de la subsomption dans un langage terminologique, on relève deux prises de position principales, relevant de l'équilibre à instaurer entre efficacité, complétude et expressivité.

- Une première catégorie de systèmes préfère conserver le pouvoir expressif de leur logique, acceptant en substance l'incomplétude des inférences de subsomption (NIKL, LOOM [MB92] et BACK). En contrepartie, ils autorisent (et prennent en compte dans leurs algorithmes) des règles d'inférences annexes, écrites par l'utilisateur, qui portent sur le domaine d'application. Ce sont ces systèmes dont les concepteurs affirment qu'un langage peu expressif ne peut être

[16] On parle ici d'expressivité pour désigner le nombre et la qualité des restricteurs de rôles.

convenable lors du développement d'une application réelle.

– La seconde catégorie de systèmes regroupe ceux qui prônent complétude des inférences et efficacité, au dépens de l'expressivité. Cette approche est souvent désignée sous l'expression *Small can be beautiful* provenant du titre d'un article de Peter Patel-Schneider en 1984, qui étudiait les avantages d'un système certes réduit, mais dont la complexité des inférences reste acceptable. Les systèmes KRYPTON et CLASSIC ont adopté ce point de vue, CLASSIC étant d'ailleurs une tentative de compromis entre efficacité, complétude et expressivité : il s'agit, de l'avis même de ses concepteurs, du langage le plus expressif possible compte-tenu des contraintes liées à la qualité des inférences [PSMB+91] [RDP+91] [BPS94], et qui a tendance maintenant à s'adapter complètement aux exigences des utilisateurs [Bra92].

Nous retiendrons que le calcul de liens de subsomption s'effectue généralement sur les descriptions (définitions) des concepts[17], et la conduite de ce type d'inférences se conforme à la sémantique généralement dénotationnelle dont sont munis les concepts et restricteurs de rôles.

Applications de la subsomption

L'algorithme d'inférence (ou test) de liens de subsomption est à la base de nombreux mécanismes plus ou moins classiques. Lorsqu'un système définit la notion de concept incohérent ($\bot$) et de concept maximum ($\top$), la subsomption permet de répondre à trois fonctionnalités de base des langages terminologiques, à savoir :

1. satisfaction de concept, autrement dit, un concept possède-t-il ou non une extension vide, compte-tenu de sa description (ce qui revient à tester si le concept en question est subsumé par $\bot$),

2. vérification de la disjonction entre deux concepts, c'est-à-dire tester si l'intersection des extensions de deux concepts est vide ou non (ce qui revient à calculer l'intersection des deux concepts, et à tester la description de concept obtenue est subsumée ou non par $\bot$),

3. vérification de l'appartenance d'un individuel à un concept générique [Sch93].

Lorsque ces trois fonctionnalités sont présentes, trois autres plus générales peuvent être imaginées, qui sont :

1. la validation d'une base de connaissances (chercher d'éventuelles incohérences entre la TBox et la ABox),

2. le filtrage d'un ensemble d'individuels (retrouver un ensemble d'individuels satisfaisant certains critères),

3. la classification d'un concept C (ou d'un individuel), qui a pour but de déterminer dans la hiérarchie de subsomption, les concepts les plus spécifiques subsumants C et les concepts les plus généraux subsumés par C (en d'autres termes, déterminer la position de C au regard de la subsomption).

[17] On parle alors de subsomption *pure*, celle sur laquelle porte les études de complexité, de décidabilité, de correction et de complétude. La subsomption non pure est celle qui prend en compte les assertions de la A-Box et les éventuels moyens d'inférences annexes.

La première de ces fonctionnalités fait partie des processus de *vérification*, la seconde est le plus souvent assimilée à la notion de *requête* rencontrée dans les systèmes de gestion de bases de données, tandis que la dernière de ces fonctionnalités est un véritable moyen d'inférence, comme nous allons le voir dans la section suivante. Certains systèmes apparentent pourtant la classification au filtrage, par classification d'un concept virtuel dont la description est constituée des critères de filtrage [BGN89].

1.4.3 Classification

La classification est un processus qui a pour objectif de déterminer, pour un concept donné, la position de ce concept dans une hiérarchie de subsomption. Il s'agit là du mécanisme d'inférence privilégié des systèmes terminologiques. Il est utilisé dans plusieurs cas, et pour des objectifs parfois très différents :

- pour inférer l'extension d'un concept C dont on connaît la description, on détermine, d'après cette description, la position de C dans la hiérarchie : tous les individuels connus pour faire partie de l'extension des subsumés de C appartiennent aussi à l'extension de C, et ce d'après la sémantique dénotationnelle des descriptions,
- lors de la phase de construction, incrémentale ou non, d'une taxonomie de concepts[18],
- lorsque l'on veut connaître le concept générique de rattachement d'un individuel, ce dernier est classé dans la taxonomie des concepts.

Globalement, la classification est un mécanisme qui permet d'acquérir des connaissances sur le domaine modélisé, sur la base d'informations disponibles, et ce tout en préservant la cohérence de la base de connaissances [MB91]. Plus exactement, le mécanisme de classification (qui s'apparente à un raisonnement) cherche à mettre en évidence des dépendances et relations qui existent ou sont susceptibles d'exister entre divers objets d'une base [Nap92a], au regard de l'ordre partiel que constitue la subsomption.

La classification étant fondée sur la recherche de liens de subsomption, et cette dernière étant généralement réalisée sur la base des descriptions des concepts, intensionnellement, le positionnement d'un concept C se fait donc par recherche de compatibilités entre la description de C et celles des autres concepts de la hiérarchie. L'hypothèse de monde ouvert implique que les algorithmes de classification (*classifiers*) utilisent un test de subsomption assimilé à une fonction à trois valeurs[19]. Ainsi, la fonction `SubsumeP(`C_1`,`C_2`)` qui teste si C_1 est subsumé par C_2 a pour résultat :

- `vrai` si la description de C_1 affine la description de C_2,
- `faux` si la description de C_1 n'affine pas la description de C_2,
- `possible` si la relation n'a pu être ni établie ni réfutée : ce résultat est dû à un manque de connaissances représentées.

[18] Une construction manuelle s'avère dangereuse lorsque le nombre de concepts dépasse un certain seuil qui dépend entre autres de la complexité des descriptions, mais aussi et surtout lorsque les bases de connaissances sont construites par plusieurs personnes, ce qui promet d'être le cas dans très peu de temps, à l'heure où le partage des informations devient un véritable leit-motiv.

[19] Peter Patel-Schneider a introduit, dans ce cadre, une logique à quatre valeurs : faux, vrai, vrai *et* faux, *ni* vrai *ni* faux [PS89a], pour résoudre les problèmes dues aux définitions cycliques.

Qu'il s'agisse de la classification d'un concept générique ou d'un individuel, l'algorithme sous-jacent est un parcours de graphe, chaque étape étant marquée par une phase d'appariemment (de comparaison).

Classification de concepts

L'algorithme de base de la classification d'un concept dans une taxonomie consiste à déterminer les subsumants les plus spécifiques (SPS) de ce concept d'une part, et les subsumés les plus généraux (SPG) de ce concept d'autre part. Cet algorithme n'est pas à confondre avec celui de *formation incrémentielle* de concepts, dont le but est de déterminer, à partir d'un ensemble d'individuels, la description d'un (ou plusieurs) concept regroupant ces individuels. Il s'agit dans ce cas de la classification dite *symbolique* en analyse de données. Seuls les concepts définis (et individuels) sont concernés par la classification, car tout algorithme de classification considéré dans les systèmes terminologiques n'est correct que lorsqu'il opère sur des *définitions*: la raison à cela est que, dans le cas de simples descriptions, il n'y a pas équivalence entre extensions et intensions de concepts, ce qui rend incomplet (voire incohérent au regard des extensions) l'algorithme de classification sur les intensions.

L'algorithme de base, celui imaginé initialement pour Kl-one [SL83], a une approche duale en ce qui concerne la détermination des ensembles SPS et SPG, réalisée par un parcours en largeur d'abord, durant lequel chaque concept correspondant à un nœud de la taxonomie est comparé au concept à classer ; la construction de SPS et SPG se fait ainsi au fur et à mesure du parcours, qui s'est amélioré, par la suite, par le développement de techniques de limitation de l'espace de recherche. Il s'agit en réalité d'un parcours de graphe somme toute classique. En ce qui concerne la comparaison du concept courant avec le concept à classer, elle revient à réaliser, dans le pire des cas, trois tests. Soit le concept courant C ne peut être qu'un subsumant du concept à classer C_c (compte-tenu des relations qu'entretient C_c avec subsumants et/ou subsumés de C déjà déterminées, figure 1.6), dans ce cas, l'algorithme teste le résultat de `SubsumeP(`C_c`,`C`)`. Dans le cas inverse, le test `SubsumeP(`C`,`C_c`)` est activé. Lorsqu'aucune information quant au lien potentiel entre C et C_c n'existe, les deux tests sont activés, et en cas de résultat négatif, certains algorithmes testent une éventuelle disjonction entre les deux descriptions de concepts. Une fois qu'un lien entre C et C_c est établi (ce qui n'est pas forcément le cas), l'algorithme procède à des propagations qui relèvent de la transitivité de la subsomption. Ainsi, lorsqu'un concept est prouvé être un subsumant de C_c, tous les subsumants de C_c sont marqués. De la même façon, lorsqu'un concept est prouvé être subsumé par C_c, tous les subsumés de C_c sont marqués. Enfin, selon le même principe, lorsqu'un concept C est disjoint à C_c, tous les subsumés de C sont marqués. Timothy Finin a proposé un algorithme interactif de classification de concept [Fin86], qui propose la classification d'un concept simultanée à la donnée de sa description, par l'utilisateur, en termes de rôles et d'attributs, en partant de l'hypothèse que la classification peut justement aider à l'identification du concept à classer, notamment lorsqu'il s'agit d'un individuel. Finin a ainsi proposé un algorithme de classification par *profil d'attributs*, c'est-à-dire que la classification progresse avec l'identification et la description du concept.

Classification d'individuels

La classification d'un individuel a pour objectif de déterminer, dans la T-Box, quels sont les concepts auxquels l'individuel peut se rattacher, autrement dit quels sont les groupements auxquels l'individuel peut appartenir. Rappelons que la description d'un individuel est répartie entre la T-Box et la A-Box, et c'est justement parce qu'ils peuvent être décrits dans la A-Box qu'ils se distinguent

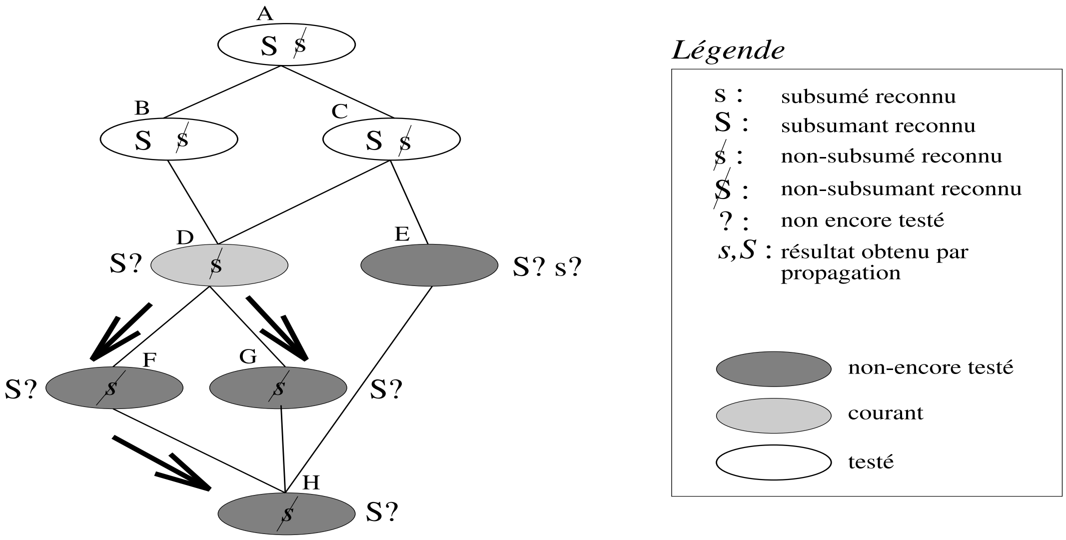


FIG. 1.6 - : Propagations / éliminations lors de la classification d'un concept. Sont représentés dans cette figure, d'une part, la taxonomie de concepts, et d'autre part, l'état courant de la classification, qui élabore au fur et à mesure les ensembles SPS et SPG. L'algorithme parcourt les concepts étiquetés ici par un caractère, dans l'ordre alphabétique de ces étiquettes. Le concept courant D est comparé avec le concept à classer C_c. Le test `SubsumeP(`D`,`C_c`)` s'étant avéré négatif, tous les sous-concepts de D ne peuvent non plus être des subsumés de C_c, la propagation est donc réalisée (elle revient à éliminer des SPG possibles les concepts D, E G H, ils n'auront donc pas à être testés en tant que subsumés potentiels). L'algorithme doit tout de même visiter ces concepts afin de les tester en tant que subsumants possibles.

des concepts génériques. Pourtant, la majorité des systèmes terminologiques ne traitent pas cette différence, et classe un individuel avec le même algorithme que s'il s'agissait d'un concept défini. Toutefois, au préalable, la description propre à l'individuel (en termes de la T-Box) est déterminée par la prise en compte simultanée de la description de l'individuel dans la A-Box et dans la T-Box : la réunion des deux constitue la description finale de l'individuel.

Pourtant, l'algorithme de classification de concepts appliqué aux individuels ne profite pas justement de ce statut particulier pour améliorer ses performances. Seuls les systèmes LOOM et BACK ont étudié la question, et ont proposé une variante de la classification de concepts appliquée à la classification d'individuels [Mac91b] [KR91].

La classification de concepts appliquée à la classification d'individuels est particulièrement coûteuse en espace mémoire dans la mesure où tous les concepts virtuels correspondants aux descriptions unifiées de chacun des individuels demeurent dans la hiérarchie. En conséquence, LOOM dispose d'un *recognizer*[20], qui effectue des comparaisons efficaces entre un individuel et un concept, tout en évitant l'encombrement dû au calcul de la description intermédiaire pour l'instance.

Lors du parcours du graphe, l'instance est représentée d'une part par l'ensemble des concepts qu'elle satisfait, et d'autre part par l'ensemble des caractéristiques qu'elle satisfait. À chaque concept rencontré lors de ce parcours, l'appartenance de l'individuel au concept est testée, caractéristique par caractéristique[21]. La satisfaction, par l'individuel, d'une caractéristique du concept testé[22] entraîne la remise à jour de l'ensemble des conditions satisfaites par cet individuel. Si toutes les ca-

[20] La traduction française n'étant pas satisfaisante (reconnaisseur, réaliseur... à la rigueur identifieur), nous conserverons l'appellation anglophone de ce mécanisme.

[21] Une caractéristique étant un rôle ou un attribut valué par l'instance.

[22] Cette validation peut être établie par application d'une règle de déduction, ou par test de la satisfaction par l'individuel des contraintes posées sur la caractéristique dans la description du concept testé.

ractéristiques d'un concept sont satisfaites par l'individuel, alors l'individuel appartient au concept en question, et l'algorithme, après une phase de propagations, se poursuit, après avoir remis à jour l'ensemble des concepts que satisfait l'individuel. La classification se termine lorsque tous les concepts ont été marqués.

Dans la dernière version du *recognizer* de LOOM, la description finale de l'instance, qui s'est enrichie au fil de sa classification, est conservée, de façon à pouvoir être réutilisée ultérieurement [MB92]. Les travaux portant sur le *recognizer* de LOOM sont comparables à ceux effectués par Carsten Kindermann quant à l'élaboration d'un algorithme similaire dédié au langage BACK.

Notons la similarité entre l'algorithme de LOOM pour la classification d'individuel et celui développé par le langage de *frames* SHIRKA qui, du fait de la distinction entre classes et instances qu'il considère, définit lui aussi un algorithme dédié à la classification d'instances [PP87]. Ce dernier réalise un parcours en profondeur d'abord de la hiérarchie de spécialisation, lors duquel, chaque classe est testée au regard de l'appartenance. Pour cela, l'appartenance d'une valeur donnée par l'instance pour un attribut de la classe est testée relativement aux contraintes posées, par l'intermédiaire de facettes, sur le domaine de valeurs accepté par cet attribut. Lorsque les contraintes sur tous les attributs d'une classe sont satisfaites par l'instance, l'appartenance de l'instance à la classe est `sûre` et l'algorithme se poursuit dans les sous-classes. Lorsque l'instance viole les contraintes posées sur au moins un attribut de la classe, l'appartenance est déclarée `impossible`, et dans les autres cas (il manque des valeurs d'attributs pour l'instance), l'appartenance n'est que `possible`[23].

1.4.4 Conclusion

À la différence des langages de *frames*, les langages terminologiques considèrent que les descriptions des unités structurées de connaissances sont en fait des définitions. Cette hypothèse, acceptable du fait de l'existence de concepts *primitifs* (qui sont des définitions de par leur dénotations dans le monde modélisé), permet à ces systèmes le développement de mécanismes de vérifications et d'inférence n'opérant que sur les définitions en intension de ces unités, présentes au sein de la T-Box.

À l'instar des réseaux sémantiques, les langages terminologiques réalisent une distinction explicite entre concepts génériques et individuels, grâce à l'existence de la A-Box. Cependant, trop peu encore de systèmes terminologiques exploitent, de par leurs inférences, cette distinction [Mac91a]. La déclaration d'un individuel dans la A-Box, par référence à d'autres individuels ou à des concepts génériques, constitue l'attribution d'une identité propre à cet individuel. Ainsi, similairement à SHIRKA et contrairement aux langages à objets, l'existence de la représentation d'un individu n'est pas dépendante de l'appartenance de cet individu à un groupe : l'existence propre d'un individu est considérée comme un fait en logique des propositions.

Notons que la migration d'instances est justement autorisée du fait que l'interprétation d'un individuel dans le monde modélisé ne soit pas définie par sa classe ou son concept d'appartenance, ce qui n'est pas le cas dans les langages de programmation par objets.

En outre, on note une correspondance entre les relations d'héritage et de subsomption, tout au moins lorsque cette dernière s'établit entre des entités génériques : la sémantique ensembliste est la même, qui se traduit dans les deux cas par l'affinement des descriptions. On remarque toutefois que, de la même façon que dans les réseaux sémantiques, la relation de subsomption est munie de la même sémantique,qu'elle lie deux concepts génériques ou un concept individuel et un générique. Dans les langages à objets, on distingue, syntaxiquement et sémantiquement, la spécialisation de

[23] Il serait d'ailleurs plus correct d'appeler ces marques respectivement `possible`, `inconnu` et `impossible`.

classes d'un côté, et l'appartenance d'une instance à une classe de l'autre. Cette distinction, dans les langages terminologiques, n'est faite qu'au niveau de la A-Box (qui ne permet, au regard de la spécialisation, que l'écriture de la relation d'un individuel vers un concept), mais en aucun cas au niveau des descriptions en intension de la T-Box.

Nous retiendrons enfin que la relation de subsomption est à l'origine de nombreux mécanismes et inférences. C'est le propre des systèmes basés sur la classification. Par ailleurs, la subsomption est généralement calculée sur la base des intensions des concepts, autrement dit sur des structures. On retient alors l'importance des descriptions de concepts au niveau des inférences, et donc l'intérêt de ne pas exprimer, en termes du même langage que le sont ces concepts interprétables dans le monde modélisé, des structures sur lesquelles ces inférences n'auraient aucun sens.

1.5 Conclusion

La notion d'objet se décline en de multiples définitions selon le contexte informatique dans lequel elle est considérée. En revanche, une constante demeure, celle qui consiste à définir l'objet comme *la représentation d'un individu ayant sa propre identité dans le domaine d'application considéré*. On retrouve d'ailleurs cette définition de l'objet dans le contexte des systèmes de gestion de bases de données à objets, dont nous n'avons pas parlé ici [NWW91] [Bar91].

Un objet peut appartenir à un ou plusieurs groupes, et les relations de dépendances existantes entre ces groupes et l'objet, mènent à distinguer diverses appellations des notions d'objet et de groupe, qui ne sont pas toujours antinomiques. Ainsi, le terme *instance* désigne généralement ces objets qui récupèrent la structure de description de la *classe* à laquelle ils ont été attachés, si ce n'est la classe qui les a explicitement générés. La notion de classe est, par ailleurs souvent confondue avec celle de *type*, en particulier lorsqu'elle est assimilée à un prédicat d'appartenance. Le terme *individuel*, quant à lui, évoque l'indépendance entre l'existence de l'objet et son appartenance à un ou plusieurs groupes, ces derniers étant généralement désignés par les termes *concept* ou *modèle*.

D'une certain façon, un groupe d'objets est caractérisé par une collection de propriétés. Ces propriétés jouent bien entendu un rôle important vis-à-vis de l'appartenance d'un objet au groupe. On distingue trois interprétations principales.

- La collection de propriétés d'un groupe représente une condition nécessaire et suffisante pour l'appartenance d'un objet au groupe. Il s'agit de la définition exacte des objets que le groupe cherche à posséder. La définition en intension du groupe (qui peut être vue comme un prédicat) caractérise exactement son extension. Cette interprétation est celle faite par les langages terminologiques quand le groupe est un concept défini.

- La collection de propriétés d'un groupe représente une condition seulement nécessaire pour l'appartenance d'un objet au groupe. Il ne suffit donc pas que l'objet satisfasse les propriétés du groupe pour y appartenir. En particulier, lorsqu'un objet appartient à un groupe, les propriétés de ce groupe ne peuvent en aucun cas correspondre à une définition de l'objet, elles ne sont interprétées que comme la donnée d'un certain nombre de caractéristiques que doit posséder l'objet. Cette interprétation des propriétés d'un groupe est essentiellement présente en représentation des connaissances, car un objet y est considéré comme étant la représentation d'une identité observée dans le monde modélisé, et en conséquence il est très difficile, voire impossible, de le définir parfaitement. Qui pourrait donner une définition exhaustive et complète de l'ensemble des mammifères, définition qui, de surcroît, doit permettre de distinguer les différents éléments de l'ensemble ? Cette interprétation est celle faite par les langages terminologiques pour les concepts primitifs, lorsqu'il est possible de leur attribuer des caractéristiques, ainsi que par les langages de programmation par objets au regard desquels les conditions portées sur les champs des objets ne sont que des conditions nécessaires.

– La collection des propriétés d'un groupe représente la description d'un prototype du groupe, d'un élément exemplaire. Cette interprétation, qui est faite par les réseaux sémantiques et les langages de *frames*, rejoint en fait la précédente.

Les multiples interprétations de l'objet ont bien évidemment une influence sur le comportement des mécanismes les manipulant [PS90]. Ainsi, par exemple, dans les langages de programmation par objets, le fait que l'existence d'un objet dépende en grande partie de l'existence de sa classe de création, interdit la migration d'instances, et en particulier l'affinement de la caractérisation d'une instance. D'ailleurs, dans ces langages, toute instance doit être complète pour être créée. À l'opposé, les langages de représentation sont conçus pour favoriser le développement de mécanismes de classification, qui, lorsqu'appliqués aux individuels, reviennent à chercher les groupes d'appartenance les plus précis pour cet individuel.

Ces différences de sémantiques apportées à la notion d'objet, et de ce fait à la notion de regroupement d'objets, modifient aussi la nature de l'ordre partiel qui les organise. On retrouve toutefois une constante autour de la relation d'héritage, à savoir qu'il permet de représenter l'affinement de la caractérisation des objets. En représentation des connaissances, elle est désignée sous les termes *subsomption* (lorsqu'établie indifféremment entre concept individuel et concept générique, ou entre deux concepts génériques), et *spécialisation* (lorsque considérée entre concepts génériques uniquement). La subsomption et la spécialisation se voient toutes deux munies d'une sémantique ensembliste, elles correspondent alors à l'inclusion. En programmation par objets, cette relation est appelée *relation d'héritage*, mais une sémantique purement ensembliste n'est pas suffisante pour la caractériser, dans la mesure où l'héritage entre deux classes introduit la notion d'affinement des comportements, ce qui exige la plupart du temps une formalisation algébrique, voire axiomatique.

Lorsque l'on utilise le terme d'objet pour caractériser un système ou un langage, il est donc essentiel de préciser l'interprétation que l'on en fait. Dans les systèmes de représentation, le langage s'adapte à l'expression de la représentation d'un monde modélisé. La sémantique dont est muni le langage, celle considérée par les mécanismes de manipulation des objets, doit, d'une part, interpréter ces objets comme des représentations du monde modélisé, et d'autre part, interpréter les descriptions de ces objets au regard de ce qu'elles cherchent à représenter, c'est-à-dire à partir de leur structure.

Nous avons pu observer qu'en représentation des connaissances, la dénotation d'un objet et sa description obéissent à des sémantiques différentes, et sont parfois manipulées indépendamment l'une de l'autre. Nous proposons alors, dans le cadre du modèle de connaissances à objets TROPES, d'expliciter ces deux sémantiques empruntées aux langages terminologiques, afin de concevoir un langage annexe pour l'expression des structures des descriptions des connaissances, dont les termes pourront être plus justement exploités par des mécanismes adaptés à la manipulation de structures de données.

Pour ce faire, nous étudierons dans ce modèle, présenté dans le chapitre suivant, le rôle des structures de données, et nous chercherons à montrer que la notion de *type* est elle aussi d'une importance capitale dans cette catégorie de systèmes, poursuivant de ce fait une étude amorcée par Alexander Borgida [Bor88] [Bor91] [Bor92], concernant les liens existants entre types de données structurés, classes d'objets et concepts.

Chapitre 2

Le modèle Tropes

TROPES [Mar93][1] est un modèle de connaissances à objets qui peut être considéré comme le petit frère de SHIRKA [Rec89], dont il reprend et étend les principales caractéristiques, que ce soit au niveau de la nature des entités de représentation, des relations maintenues entre ces entités, ou encore des mécanismes d'inférence et d'exploitation. TROPES est issu de la technologie à objets, et fait partie de la lignée des systèmes de représentation des connaissances dits *de classification*, tels que ceux rencontrés dans les logiques descriptives, même si plusieurs différences notoires peuvent être observées, que nous étudierons au fil de l'exposé de TROPES auquel est consacré ce chapitre.

Notre étude est basée sur le modèle TROPES. Ce chapitre est donc réservé à la présentation de ce modèle. Nous interprèterons TROPES, tel qu'il a été initialement conçu, afin de dégager les différences entre la description des objets et leur dénotation dans le monde modélisé. Nous montrerons alors que ces descriptions peuvent être interprétées comme des *ensembles de valeurs* structurés, et manipulables à partir de primitives extérieures au système, la dualité *structure+opération* correspondant à la notion de type de données.

Après une description volontairement informelle des objets de représentation de TROPES et des relations sémantiques dont ils sont les acteurs (section 2.1), la présentation de TROPES se poursuit par une définition plus formelle de son modèle et de sa sémantique (section 2.2), nécessaire à une description précise des processus d'inférence et d'exploitation mis en œuvre sur les objets (section 2.3). La dernière section de ce chapitre est consacrée à la mise en évidence des ambiguités inhérentes à TROPES, qui sont dues à la double sémantique qui définit le modèle, correspondant aux sémantiques intensionnelle et extensionnelle que nous avons introduites lors de la présentation des langages terminologiques. Un premier pas vers le traitement de ces ambiguités est alors proposé, qui consiste à développer pour TROPES un système de types qui se verra déléguer l'interprétation des descriptions des entités de représentation.

2.1 Représentation des connaissances dans Tropes

Les modèles de connaissances à objets ont pour but de représenter des catégories qui réunissent des individus observés dans le domaine de l'application considéré, ces individus étant eux aussi représentés dans le modèle. Les catégories et les individus sont représentés dans TROPES par des concepts, des classes et des instances, qui sont appelés des *entités (ou termes) de représenta-*

[1] La version actuelle de TROPES, ainsi que son manuel d'utilisation, sont disponibles à l'URL `http://everest.imag.fr`.

tion. Le modèle TROPES s'attache ainsi à définir un langage de représentation structuré, dont les termes peuvent s'interpréter naturellement dans le domaine de l'application, quelle que soit cette application.

Dans tout domaine d'application, les connaissances observées ne sont pas indépendantes les unes des autres. Le modèle de connaissances s'intéresse à identifier certaines relations naturelles entre ces connaissances, de façon à établir un mode de représentation de ces relations ramenées au niveau des termes représentant les connaissances. En ce sens, un modèle de connaissances est entièrement défini par son langage de représentation (syntaxe et sémantique des entités de représentation) et par les relations qu'il permet de définir entre les termes de ce langage, indépendamment de l'exploitation qui peut ultérieurement en être faite.

Les entités de représentation dans TROPES sont à leur tour définies au sein du modèle (informatiquement) comme des *schémas*, selon le même principe que SHIRKA. Un schéma est une entité informatique qui contient tout ce qui est nécessaire à la gestion automatique et calculatoire de l'entité de représentation à laquelle est elle associée. Il existe une version de TROPES qui dispose d'un méta-niveau [Sch95], ce dernier définissant la structure et le comportement d'un schéma en fonction de la catégorie de l'entité que représente ce schéma. Ces catégories d'entités correspondent à différents degrés d'abstraction dans la représentation du monde modélisé.

TROPES distingue trois niveaux de représentation des connaissances, correspondant à trois degrés de généralité de modélisation.

2.1.1 Concepts

Le plus haut degré de généralité est traduit par la notion de *concept*, un concept dénotant un ensemble de connaissances particulières (les individus) observées dans le monde modélisé, de par une première identification des propriétés qui caractérisent ces connaissances de façon unique. Par exemple, le concept des *oiseaux*, dont le but est de regrouper tous les oiseaux de la Terre et de les classer, peut être dans cette optique informé par diverses propriétés comme le fait qu'ils volent, qu'ils ont un bec et des plumes, qu'ils pondent, qu'ils migrent, ou encore par leurs lieux d'habitat, leur nourriture ou leur durée de vie. Ces propriétés très diverses sont représentées dans le modèle par le biais d'*attributs* nommés.

Une base de connaissances TROPES est la spécification d'un ensemble de concepts, c'est-à-dire la description de la représentation d'un ensemble d'individus, selon plusieurs niveaux d'abstraction, ainsi que la représentation la plus précise possible de ces individus.

2.1.2 Classes et points de vue

Le second degré de généralité est représenté par un découpage du concept en *classes*. Une classe regroupe un sous-ensemble des connaissances que réunit le concept duquel est issu la classe. Ce sous-ensemble est caractérisé par un affinement des propriétés du concept, c'est-à-dire par un apport de précision sur la nature de ces propriétés, qui naturellement restreint le nombre de connaissances spécifiques candidates. Les classes d'un concept sont regroupées en *points de vue*, un point de vue reflétant un axe particulier d'observation des individus du concept. Les classes d'un point de vue sont donc orientées vers l'affinement d'un sous-ensemble des propriétés identifiées par le concept, ce sous-ensemble étant propre au point de vue. En ce sens, une classe reflète en réalité une vue plus spécifique des connaissances qu'elle regroupe, car elle est plus ciblée sur certaines propriétés[2]. Pour suivre l'exemple du concept des *oiseaux*, il est possible de dégager plusieurs perspectives d'observation, comme le point de vue de l'habitat, de la physiologie ou encore de la reproduction. Dans chaque point de vue, un sous-ensemble précis de propriétés est privilégié, qui est alors considéré par les classes pour une caractérisation plus fine des individus représentés. Par exemple, les propriétés portant sur le bec des oiseaux ou sur la forme de leurs ailes, sont pertinentes du point de vue physiologique, mais n'ont a priori pas de raison d'être examinées dans l'observation des oiseaux du point de vue de leur habitation (ou tout au moins pas directement...).

Spécialisation de classes

Les classes d'un point de vue sont organisées en arbre, par la relation de *spécialisation* qui s'apparente, dans une certaine mesure, à la relation de subsomption rencontrée dans les systèmes à logique descriptive [PS90] [ND92]. Dans un point de vue, une classe C est plus spécialisée qu'une classe C' si l'ensemble d'individus dénoté par C est inclus dans l'ensemble des individus dénoté par C'. Cette condition sur la spécialisation peut être interprétée en termes des propriétés décrivant les classes dans le modèle : C est plus spécialisée que C' (on dit que C est une sous-classe de C') si leurs ensembles de propriétés sont sémantiquement comparables et si les informations concernant les propriétés de C sont globalement plus précises que celles portant sur les propriétés correspondantes de C' (section 2.1.6). La spécialisation concerne ainsi un gain en précision en ce qui concerne la caractérisation des propriétés des individus. Dans l'exemple des oiseaux, du point de vue de leur habitation, il est possible de spécifier que la classe des migrateurs-vers-le-sud est une spécialisation des oiseaux migrateurs. Les sommets de l'arbre de chacun des points de vue d'un concept sont des classes structurellement différentes mais qui représentent le même ensemble d'individus : cet ensemble est décrit de différentes façons, selon les propriétés propres à chaque point de vue considéré. Le sommet d'un point de vue est appelé la classe *racine* de ce point de vue.

Passerelles entre points de vue

Enfin, la relation *inter-points de vue* permet l'établissement de liens d'un ensemble de points de vue différents vers un autre point de vue, par le biais des classes qui sont les sources ou la destination de *passerelles*. Il existe une passerelle des classes $C_1 \cdots C_n$ (des points de vue $P_1 \cdots P_n$) vers la classe C du point de vue P pour signifier que l'intersection des ensembles d'individus dénotés par les classes $C_1 \cdots C_n$ fait partie de l'ensemble des individus dénotés par la classe C. Cette relation est essentielle quand il s'agit d'établir des correspondances sémantiques entre différents axes d'observation des individus d'un concept, autrement dit lorsque le concepteur tente de représenter des liens entre ensembles de propriétés qui, à première vue, n'ont rien en commun. En particulier, les passerelles permettent de décrire une même classe selon différents points de vue, c'est-à-dire que la classe possède plusieurs descriptions en termes de propriétés, pour une même dénotation : cette équivalence de dénotation est réalisée en TROPES par la pose d'une passerelle bidirectionnelle entre les deux schémas de classes des deux points de vue. Par exemple, la classe des guépiers peut être décrite différemment du point de vue de l'habitat et du point de vue physiologique des oiseaux. Notons que pour tout concept d'une base de connaissances, il existe une passerelle bidirectionnelle systématique entre les racines des points de vue.

[2] La notion de point de vue est comparable à celle de *vue* existante dans les systèmes de gestion des bases de données [Ull88], une vue étant une vision partielle d'une base unique. Pourtant, les points de vue sont plus puissants que les vues, dans la mesure où, à chacun d'entre eux, correspond une taxonomie.

2.1.3 Instances

Le dernier degré de généralité est représenté par les *instances*. Une instance dénote une connaissance particulière, c'est-à-dire un individu dans le monde modélisé. Une instance appartient à un concept : le rattachement d'une instance à un concept, autrement dit la création d'une instance, revient à représenter explicitement, dans le modèle, l'appartenance d'un individu du monde modélisé (l'instance) à un ensemble d'individus de ce monde (le concept). La reconnaissance de cette appartenance provient du fait que l'individu est reconnu pour posséder les propriétés caractéristiques des autres individus de l'ensemble. C'est en ce sens que l'instance qui représente l'individu doit satisfaire les conditions sur les attributs qui représentent les propriétés du concept.

Si une instance est initialement attachée à un concept, ce qui lui confère une identité et scelle son interprétation, elle peut aussi être observée selon les différents points de vue. Ainsi, une instance peut être membre d'une classe de chaque point de vue, ce qui reflète une caractérisation plus fine de l'instance, dans la mesure où les classes d'un point de vue s'attachent à exploiter un certain nombre de propriétés particulières du concept. Plus la classe de rattachement d'une instance dans un point de vue est spécifique (basse dans la hiérarchie de spécialisation), plus les propriétés sur cette instance sont canalisées et meilleure est l'identification de cette instance. Par exemple, lorsque l'on observe aux jumelles un animal que l'on sait être un oiseau, nous pouvons le représenter dans le modèle par une instance : l'observation de cet oiseau selon certains points de vue, c'est-à-dire relativement à certaines propriétés, permet de le rattacher à des classes qui nous en offrent une caractérisation plus fine, comme le fait de savoir qu'en plus d'être un oiseau, il s'agit d'un guépier et qu'il possède les caractéristiques d'un oiseau migrateur.

Il est donc possible de parler d'instanciation lorsqu'une instance est liée à une classe d'un point de vue du concept, ce qui signifie simplement l'appartenance de l'individu dénoté par l'instance au sous-ensemble dénoté par la classe. Il s'agit toutefois d'un abus de langage dans la mesure où l'instanciation reflète la *création* de l'instance, à savoir sa prise d'identité, qui ne lui est pas accordée par une classe mais par le concept [Euz93]. On parlera alors plutôt d'attachement d'une instance à une classe, et d'instanciation d'une instance à un concept, sauf lorsqu'aucune confusion ne pourra être faite.

2.1.4 Que représentent les entités de représentation ?

La figure 2.1 montre la perspective dénotationnelle des entités de représentation, grâce à laquelle la confrontation des trois niveaux de généralité s'interprète naturellement, dans le monde modélisé, par l'inclusion ensembliste et l'appartenance d'une valeur à un ensemble.

Le rôle du modèle à objets est de permettre une représentation des connaissances fidèle à la dénotation, sans expliciter complètement cette dénotation car cela ramènerait les termes à de simples identificateurs, sur lesquels les inférences sont extrêmement limitées. Les modèles à objets choisissent ainsi de décrire les entités de représentation à l'aide d'un nombre fini de propriétés structurées, les attributs, en établissant un lien sémantique entre ces propriétés et la dénotation de l'entité. Il s'agit d'un choix de représentation intéressant dans la mesure où la caractérisation des propriétés pertinentes contribue à l'enrichissement de la connaissance que l'on a sur le monde modélisé.

Ainsi, la perception ensembliste des concepts, des classes et des instances se traduit, dans le modèle de connaissances, par le biais des propriétés représentées par les attributs. De ce fait, le respect de la dénotation est très largement tributaire de la justesse de la caractérisation et de

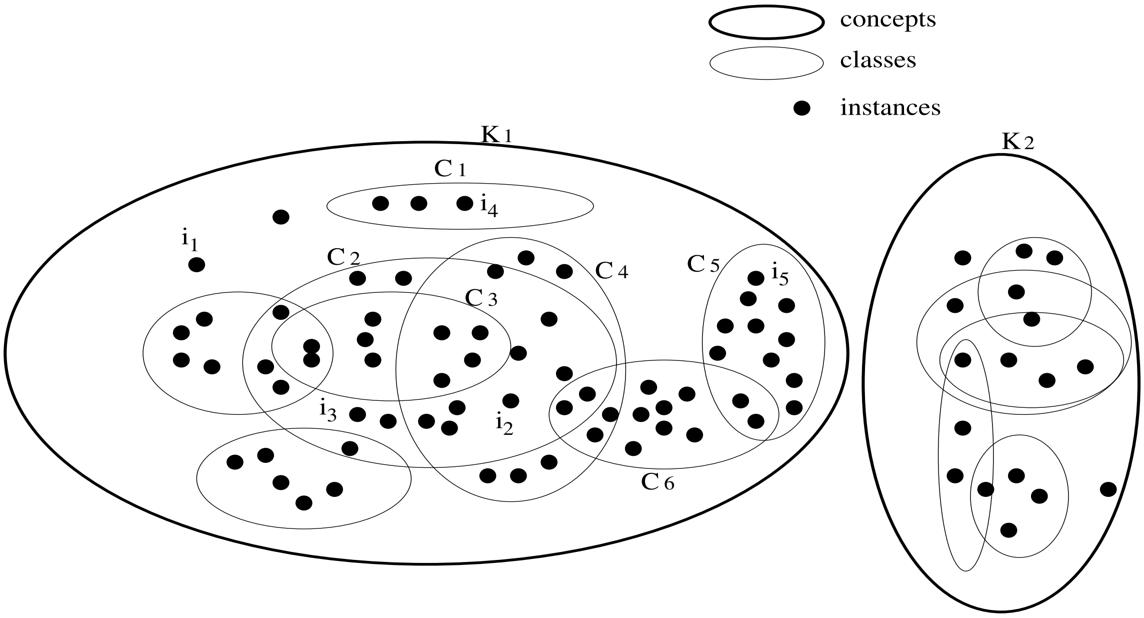


FIG. 2.1 - : Cette figure représente une vision ensembliste des notions de concept, classes et instances. Les concepts qui constituent une base de connaissances sont, du point de vue ensembliste, indépendants, dans la mesure où leurs ensembles d'instances sont disjoints. Chaque concept est partagé en classes non forcément disjointes. Une instance appartient à une ou plusieurs classes identifiées du concept, quitte à ce qu'elle soit attachée aux racines des points de vue. Les classes racines ne sont pas figurées ici pour ne pas charger le dessin : d'un point de vue ensembliste, elles se confondent avec le concept. Ce schéma peut être perçu comme une interprétation dénotationnelle d'une base de connaissances, où le domaine d'interprétation est le monde modélisé.

la spécification des propriétés. Il est sans conteste nécessaire, pour un modèle de connaissances à objets, de définir exactement l'influence des propriétés sur la dénotation des termes dans le monde modélisé.

La figure 2.1 n'illustre pas la notion de point de vue car celle-ci est directement liée à la description des entités en termes de propriétés. Si l'on considère l'aspect ensembliste, un point de vue représente la projection du concept selon l'ensemble des propriétés pertinentes sous ce point de vue (figure 2.2), autrement dit un point de vue reflète un partage du concept relatif à certaines propriétés identifiées.

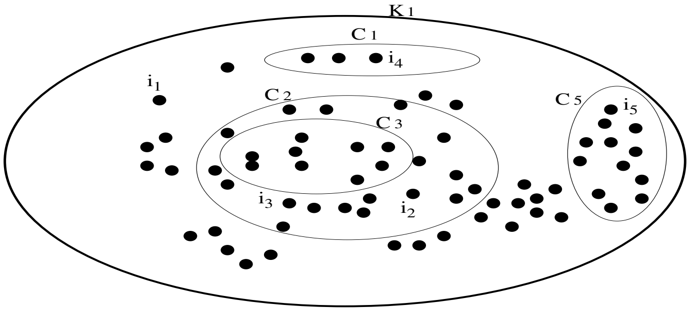


FIG. 2.2 - : Cette figure illustre la vision ensembliste de la notion de point de vue. Le concept K_1 de la figure précédente a été projeté selon les propriétés d'un point de vue auquel appartient les classes C_1, C_2, C_3, C_5. L'affinement du concept réalisé par les classes C_4 et C_6 n'est plus représenté puisqu'il n'est pas pertinent pour le point de vue considéré. La représentation dénotationnelle de ce point de vue montre par ailleurs que C_3 est une spécialisation de C_2.

La figure 2.3 est un exemple de structuration d'un concept TROPES. La description des entités de représentation en termes de propriétés n'y est pas illustrée, il s'agit du thème de la section suivante.

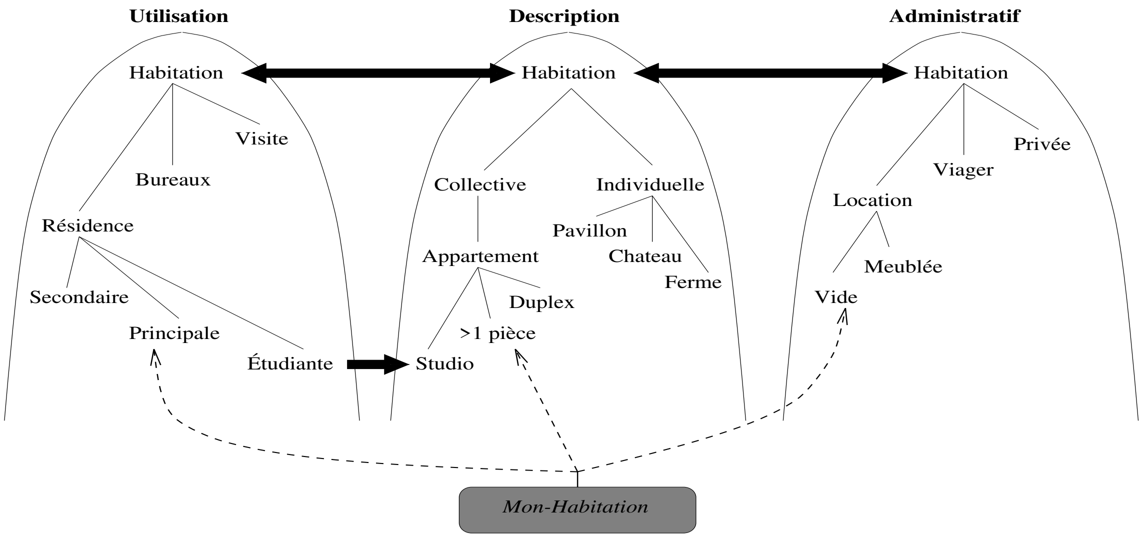


FIG. 2.3 - : Le concept d'`habitation` peut être observé sous au moins trois points de vue (`Utilisation`, `Description` et `Administratif`). L'instance `Mon-habitation` se décompose sous chacun des points de vue en se rattachant à une classe de chacun : ici, du point de vue administratif, `Mon-habitation` est une `Location Vide`. Outre la passerelle bidirectionnelle entre les racines des points de vue, il existe une passerelle unidirectionnelle de `Résidence Étudiante` vers `Studio`, stipulant que toute habitation qui est une résidence étudiante est un studio.

2.1.5 Attributs (ou comment sont représentées les entités de représentation?)

Les concepts et les classes se décrivent en termes d'attributs structurés, qui représentent des propriétés, et qui sont nécessairement spécifiés par un couple (identificateur / domaine de valeurs). Le domaine de valeurs est en fait soit un type de données, soit un autre concept de la base de connaissances. Selon le même principe que dans SHIRKA, les concepts, les classes, les instances et les attributs sont, dans TROPES, décrits par des *schémas* : un attribut de même nom (identificateur) dans deux classes différentes est représenté par deux schémas d'attributs.

Un concept déclare toutes les propriétés et caractéristiques pertinentes pour ses instances, selon tous les points de vue d'observation. La déclaration d'un attribut au niveau du concept équivaut à en donner son *type principal*, à savoir l'ensemble des valeurs que peut prendre cet attribut ou le concept dont les instances caractérisent cette propriété. Au niveau des classes, le type principal peut être affiné au moyen de *descripteurs* (l'équivalent des facettes dans les langages de *frames*) (restriction de l'ensemble de valuation de l'attribut). Les descripteurs de restriction de domaine sont appelées *descripteurs de typage*. À la différence des langages terminologiques, TROPES n'assimile pas les descriptions d'entités à des définitions.

TROPES définit onze descripteurs de typage, les trois premiers permettant de spécifier le type principal d'un attribut à partir d'un type initial, les suivants permettant de restreindre le domaine de ce type principal.

Descripteurs de typage principal Tout attribut doit être obligatoirement et initialement spécifié par un de ces descripteurs.

– `$un`, suivi d'un type de données, d'un concept ou d'une liste de classes (type initial de l'attribut), signifie que les valeurs de l'attribut sont des valeurs à prendre dans le domaine spécifié.
– `$liste-de`, suivi d'un type de données, d'un concept ou d'une liste de classes (type initial de l'attribut), signifie que les valeurs de l'attribut sont des valeurs à prendre parmi des listes de valeurs issues du type initial.
– `$ens-de`, suivi d'un type de données, d'un concept ou d'une liste de classes (type initial de l'attribut), signifie que les valeurs de l'attribut sont des valeurs à prendre parmi des ensembles de valeurs issues du type initial.

La spécification d'un (et d'un seul) de ces trois descripteurs fixe le **type principal** de l'attribut dans le concept. Il peut ainsi s'agir d'un type *monovalué* (application du descripteur `$un` au type initial), ou d'un type *multivalué* (application des descripteurs `$liste-de` ou `$ens-de` au type initial).

Descripteurs de restriction de domaines La déclaration de tels descripteurs pour un attribut n'est jamais obligatoire, elle permet simplement de restreindre le domaine de valeurs de l'attribut défini par le type principal.

– `$domaine`, suivi d'une liste de valeurs appartenant au type principal, représente l'énumération des valeurs admises pour cet attribut.
– `$intervalle`, suivi d'une liste d'intervalles de valeurs issues du type principal (obligatoirement un type ordonné), représente le domaine de valeurs admises pour l'attribut, à savoir les valeurs comprises entre les bornes d'un des intervalles.
– `$sauf`, suivi d'une liste de valeurs appartenant au type principal, représente l'énumération des valeurs interdites pour cet attribut.
– `$sauf-intervalle`, suivi d'une liste d'intervalles de valeurs issues du type principal (obligatoirement un type ordonné), représente le domaine de valeurs interdites pour l'attribut, à savoir les valeurs comprises entre les bornes d'un des intervalles.
– `$valeur`, suivi d'une valeur issue du type principal, représente la seule valeur possible pour cet attribut.
– `$parmi`, suivi d'une énumération de valeurs issues du type initial, représente une restriction du domaine sur lequel les valeurs multivaluées peuvent être construites (restriction du type initial).
– `$interdit`, suivi d'une énumération de valeurs issues du type initial, représente les valeurs interdites du domaine sur lequel les valeurs multivaluées peuvent être construites (restriction du type initial).
– `$cardinalité`, suivi d'un intervalle de valeurs entières positives, représente la cardinalité possible dans le cas de valeurs dont le type principal est multi-valué.

Les trois derniers descripteurs ne sont applicables qu'aux attributs dont le type principal est multivalué. De la même façon, les descripteurs relatifs à des intervalles ne sont applicables qu'aux attributs dont le type principal est ordonné. Ces descripteurs peuvent être combinés (par conjonctions) pour un même attribut, afin de lui associer un domaine de valeurs particulier (exemple figure 2.4).

TROPES distingue clairement les attributs de concept et les attributs de classe, même si les deux manipulent un ensemble d'identificateurs commun. Un attribut de concept associe à un identificateur son type principal, cet identificateur étant le porteur de la sémantique de cet attribut. Par référence à l'identificateur d'un attribut de concept, et donc à sa sémantique, un attribut de classe en affine le type principal, à l'aide des descripteurs de restriction de domaines. Autrement dit, l'attribut de concept associe à un identificateur une propriété particulière et unique, et l'attribut de classe affine les connaissances sur cette propriété.

nombre-de-pattes	***nombre-de-pattes***	***nombre-de-pattes***
$un Entier	$un Entier	$un Entier
$intervalle [0 8]	$domaine (0 2 4 6 8)	$sauf-intervalle [-inf;0[ ]8;+inf[
$sauf (1 3 5 7)		$sauf (1 3 5 7)

FIG. 2.4 - : Les descripteurs de typage peuvent être combinés pour établir un domaine de valeurs. Sur cet exemple, les trois expressions sont issues de combinaisons différentes de descripteurs valués, mais elles sont toutes trois équivalentes par leur dénotation : le domaine de valeurs signifié est une des valeurs entières de l'ensemble {0 2 4 6 8}.

Les attributs dont le type initial est un concept sont appelés des *attributs complexes*, par opposition à tous les autres attributs dont le type principal est un type de données prédéfini dans le langage hôte (comme Entier ou Date), qui sont des *attributs élémentaires*. Nous retrouvons ici la distinction entre les notions d'attribut et de rôle dans les langages terminologiques (section 1.4.1). TROPES interdit les définitions récursives, comme la plupart des systèmes à typage statique. Cela signifie qu'un attribut complexe ne peut faire référence au concept qui le contient.

Les descripteurs de typage sont spécifiés par des informations *statiques*, c'est-à-dire dont l'évaluation n'est pas dynamique et peut être effectuée à la compilation d'une base de connaissances. TROPES possède d'autres catégories de descripteurs (à évaluation dynamique) qui seront présentés dans la section 2.3.2.

Une instance dans TROPES est tout d'abord créée, c'est-à-dire attachée explicitement à un concept, puis à un ensemble de classes (une par point de vue). Ensuite, l'instance est progressivement caractérisée, par identification d'une valeur spécifique pour un sous-ensemble d'attribut de son concept, valeur qui se doit d'appartenir au domaine de l'attribut spécifié dans le concept ou dans les classes d'appartenance.

Un concept fournit, parmi l'ensemble des attributs représentant les propriétés potentielles de ses instances, une *clé* pour ces instances. Une clé est une collection d'identificateurs d'attributs qui, à l'instar des langages de bases de données, assure l'unicité de l'identité d'une instance. Une instance, à sa création, se voit associer une identité à partir du moment où elle associe une valeur à chaque attribut de la clé.

2.1.6 Relations

Les entités qui peuvent être définies en termes du langage de représentation de TROPES ne sont pas indépendantes les unes des autres, trois relations conceptuellement fortes permettent d'établir des liens intra et inter degrés de généralité. Il s'agit des relations d'instanciation, de spécialisation, qui sont classiques dans les modèles à objets, ainsi que de la relation entre points de vues propre à TROPES.

Instanciation

La section précédente aborde le lien sémantique qu'il existe entre toute instance et son concept d'appartenance (une instance n'existe pas indépendamment d'un concept). Le rattachement d'une instance à un concept implique l'attribution d'une identité à l'instance ainsi que la reconnaissance de sa dénotation dans le monde modélisé. Pour cela, parmi les propriétés (attributs) qui constituent la description du concept, ce dernier doit identifier la *clé* de ses instances, c'est-à-dire un n-uplet d'attributs pour lesquels chaque instance associe un unique n-uplet de valeurs, différent de celui des autres instances.

La création d'une instance à partir d'un concept est nommée *instanciation*. Elle est assertionnelle (une instance est déclarée appartenir à un concept). En ce sens, l'instanciation correspond à la A-Box rencontrée dans les systèmes à logiques descriptives. Mais pour qu'une telle assertion soit acceptée par le modèle, une valeur associée par l'instance à un attribut doit nécessairement appartenir au domaine de cet attribut tel qu'il est spécifié dans le concept ou la classe de rattachement. Il s'agit d'une condition certes rigide, mais qui permet de garantir la correction de l'interprétation dénotationnelle de la base de connaissances (section 2.2).

L'instanciation en TROPES est "assertée" par le lien `est-un`. Une instance est créée à partir d'un concept grâce à la donnée de ce lien de l'instance vers le concept, puis l'instance peut être attachée à au plus une classe par point de vue (par le même lien `est-un`), ce qui traduit une meilleure caractérisation des propriétés de l'individu dénoté par rapport à l'ensemble auquel il appartient, et qui se représente dans le modèle par de plus fortes contraintes sur certains attributs (figure 2.5). L'instanciation d'une instance vers un concept (ou une classe) K est notée $I \in K$.

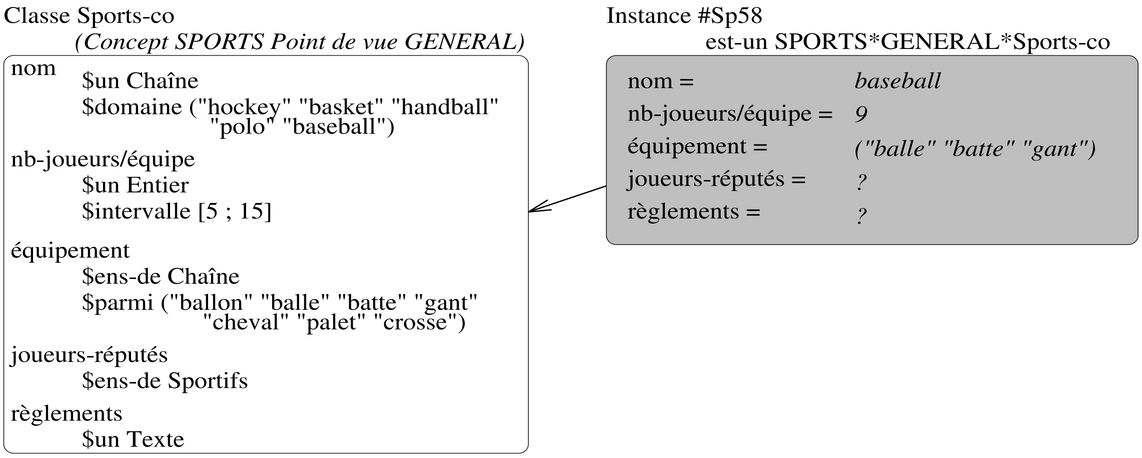


FIG. 2.5 - : L'instance `#Sp58` a été créée dans la base de connaissances grâce à son rattachement au concept `SPORT`. `GENERAL` est un des points de vue selon lequel la notion de sport peut être observée, qui correspond à la présentation générale de tous les sports. L'appartenance de l'instance à la classe `Sports-co` de ce point de vue est elle aussi représentée par le lien `est-un`. La figure montre la description en termes d'attributs de la classe `Sports-co`, ainsi que la valuation de ces attributs par l'instance `#Sp58` (l'instance possède d'autres propriétés, représentées par d'autres attributs, qui ne sont pas figurés ici car ils ne sont pas pertinents sous ce point de vue d'observation). Toutes les valeurs d'attributs ne sont pas connues, ce qui ne remet pourtant pas en cause l'appartenance dans la mesure où il s'agit d'une assertion.

Une instance incomplète est une instance qui n'associe pas de valeurs à tous les attributs spécifiés dans sa ou ses classes d'appartenance. Cela signifie simplement que l'instance n'est pas entièrement identifiée dans le modèle vis-à-vis des propriétés qu'elle est susceptible de posséder. L'incomplétude d'une instance ne remet toutefois pas en cause l'appartenance de l'instance à une classe dans la mesure où cette appartenance est assertée et s'interprète dans le monde modélisé de façon unique.

Spécialisation

La spécialisation est la relation entre classes d'un même point de vue qui est à la base de l'organisation taxonomique de ce point de vue. Elle relève bien sûr d'une technique de structuration propre aux modèles à objets, ce qui lui vaut d'être souvent, mais abusivement, assimilée au mécanisme d'héritage. Toutefois, la spécialisation est avant tout la représentation dans le modèle des multiples niveaux de précisions qui régissent l'observation du monde réel. La spécialisation est en ce sens une technique de modélisation qui a fait ses preuves dans de nombreux domaines d'application, fondée sur l'affinement progressif de la caractérisation des individus à représenter. La spécialisation esr la représentation dans le modèle de l'inclusion ensembliste considérée dans le monde modélisé ; cela se traduit par le fait que toute instance d'une classe C est instance de la super-classe de C.

Dans TROPES, la spécialisation est validée au niveau des propriétés décrivant les classes. Une classe C peut être sous-classe d'une classe C' à condition que C possède toutes les propriétés de C' (et éventuellement de nouvelles qui n'introduisent pas d'incohérence), dont elle a la possibilité de restreindre les conditions. Cette restriction sur les propriétés (attributs) a pour effet de restreindre naturellement l'ensemble des individus dénoté par la sous-classe, puisque certains de la super-classe ne vérifient plus les nouvelles conditions apportées dans la classe la plus spécifique.

De même que l'instanciation, la spécialisation dans TROPES est assertionnelle, son extension est d'ailleurs issue de la construction incrémentale de la base de connaissances : une classe est définie à partir d'une autre, dont elle récupère les propriétés, les affine éventuellement et en précise d'autres. La spécialisation d'une classe C_1 par une autre classe C_2 est ainsi assertée par le lien `sorte-de`, elle est notée, dans TROPES, $C_2 \leq_\sigma C_2$.

La restriction portée par une classe sur les propriétés de sa super-classe concerne les domaines de valeurs associés aux attributs : le domaine de valeurs d'un attribut d'une classe doit être inclus dans le domaine de valeurs de l'attribut correspondant (de même identificateur) dans la super-classe (figure 2.6). La vérification porte ainsi sur l'interprétation qui est faite des descripteurs de typage et de la combinaison de ces descripteurs. Lorsqu'il s'agit d'un attribut complexe, la vérification est ramenée à la vérification de la spécialisation entre les classes données comme domaine de l'attribut.

Tous les points de vue d'un concept sont ordonnés en arbre par la relation de spécialisation. Deux classes sœurs (dont la super-classe immédiate est commune aux deux) sont disjointes, c'est-à-dire que les ensembles d'individus qu'elles dénotent chacune ont une intersection vide.

Sur un arbre de spécialisation, le mécanisme d'héritage est activé dynamiquement lors de l'accès à la spécification d'un attribut qu'une classe récupère d'une super-classe. Il est important de faire la distinction entre la spécialisation et l'héritage : la spécialisation est interprétée par le modèle, il s'agit de la représentation d'une relation sémantique observée dans le monde modélisé, alors que l'héritage est un mécanisme directement issu de la technologie “objet”, qui profite, pour sa mise en œuvre, de l'organisation hiérarchique des attributs et des classes issue de la spécialisation.

La spécialisation de classes dans TROPES est tout à fait comparable à la subsomption dont il est fait état dans les systèmes à descriptions logiques, le principe dans les deux cas étant un affinement dans l'identification des individus dénotés. Spécialisation et subsomption définissent chacune un ordre partiel sur des éléments, la différence étant la nature même de ces éléments. En effet, même si les systèmes terminologiques établissent une différence conceptuelle entre individuels et concepts définis [RDP+91], la subsomption ne fait pas la différence : un lien de subsomption entre deux concepts définis a la même sémantique qu'un lien de subsomption entre un individuel et un concept défini, mis à part le fait qu'un concept individuel est une feuille dans le graphe de subsomption (section 1.4.2). À l'opposé, TROPES distingue la représentation des instances et la représentation des

Classe Habitation
(Concept HABITATION Point de vue DESCRIPTION)

dénomination $un Chaîne $domaine ("Cours" "Boulevard" "Rue"
"Chemin" "Passage" "Route" "Impasse" "Avenue" "Place")
numéro $un Entier $intervalle [0 ; +inf]
nom $un Chaîne
code-postal $liste-de Entier $cardinalité [5 ; 5] $parmi [0 ; 9]
commune $un Commune
exposition $liste-de Chaîne $parmi ("Sud" "Est" "Ouest" "Nord")
$cardinalité [1 ; 4]
chauffage $un Chaîne $domaine ("Collectif" "Mazout" "Gaz" "Electrique"
"sans" "solaire" "bois")
surface $un Entier $intervalle [9 ; +inf]
nb-pièces $un Entier $sauf-intervalle [250 ; +inf]

Classe Collective sorte-de Habitation
(Concept HABITATION Point de vue DESCRIPTION)

nb-pièces $intervalle [1 ; 20]
chauffage $sauf ("solaire")
surface $un Entier $intervalle [9 ; 400]
attributs affinés

étage $un Entier $intervalle [0 ; 78]
ascenseur $un Booléen
voisins-immédiats $ens-de Personne
attributs ajoutés

attributs hérités

FIG. 2.6 - : La spécialisation impose la restriction des contraintes statiques portées sur les attributs de classes. La sous-classe récupère la spécification de certains attributs de sa super-classe par héritage dynamique, redéfinit les autres (attributs affinés) et en introduit de nouveaux qui n'avaient pas lieu d'être dans les classes supérieures (attributs ajoutés). Notons qu'en réalité, le type principal d'un attribut est fourni au niveau du concept : ce n'est pas le cas sur cette figure, de façon à éviter l'écriture non pertinente ici de la définition du concept.

classes, ce qui amène une distinction nette entre instanciation et spécialisation[3]. La subsomption dans les systèmes terminologiques est la généralisation des relations de spécialisation et d'instanciation rencontrées dans les modèles à objets, et c'est une des raisons qui a fait dire à Amedeo Napoli que la subsomption subsume la spécialisation [Nap92b].

Passerelles

Même si la notion de points de vue est partagée par quelques autres systèmes de représentation des connaissances, tels que KRL [BW77], ROME [CG90], LOOPS [SB85] ou encore VIEWS [Dav87], la notion de coopération entre ces points de vue est assez originale.

Établir une passerelle d'un ensemble de points de vue vers un autre point de vue correspond à la représentation, dans le modèle, d'une correspondance dénotationnelle entre plusieurs ensembles de propriétés a priori sans rapport puisque relatifs à des points de vue d'observation différents. Une passerelle des classes $C_1 \cdots C_n$ vers la classe C signifie que tous les individus qui appartiennent à chacun des ensembles dénotés par les classes $C_1 \cdots C_n$ appartiennent aussi à la dénotation de C. Une telle passerelle est notée $\{C_1, \cdots, C_n\} \gg C$, elle permet de déduire que toute instance des classes $C_1 \cdots C_n$ est aussi une instance de la classe C ; c'est d'ailleurs pour ce type de déduction que permet d'établir une passerelle qu'il est possible de comparer les passerelles aux règles rencontrées dans les

[3] À l'origine, cette différentiation provient de la distinction faite entre type et valeur, ou plus généralement entre ensemble et élément, qui n'est pas universelle.

logiques de description [PSMB+91]. En effet, dans les deux cas, il s'agit de permettre l'expression d'un lien d'inclusion dénotationelle entre deux ensembles, indépendamment de la description de ces ensembles.

Les points de vue de TROPES ont été créés pour résoudre les conflits dus à l'héritage multiple, tout en permettant à une instance d'appartenir à plusieurs classes, une par point de vue : il s'agit de permettre la représentation des instances selon différentes perspectives, donc selon divers ensembles de propriétés de ces instances, autrement dit selon de multiples descriptions en termes d'attributs. De même que dans le cas de la spécialisation multiple, plusieurs descriptions pour une même instance peuvent provoquer des conflits de noms, et par là même des incohérences dues aux conditions portées sur les domaines des attributs. En réalité, ce problème dans TROPES est évité dans la mesure où un attribut est déclaré, et donc sémantiquement défini, au niveau du concept. Cette déclaration d'un attribut au plus haut niveau de généralité impose la signification de l'attribut, *quel que soit le point de vue considéré par la suite*. En conséquence, une incompatibilité entre deux domaines d'attributs de même nom, situés dans deux classes partageant une instance, n'est pas interprétée comme un conflit de noms. Cette incompatibilité n'est donc pas levée par le choix d'un des deux domaines ; elle est traitée et corrigée comme une incohérence dans la description de l'instance, ou dans la description des classes.

Cette considération est importante à partir du moment où l'on autorise la déclaration de passerelles entre des points de vue, qui amènent naturellement, au niveau du modèle, à confronter sémantiquement plusieurs descriptions différentes. Le comportement du modèle, face à des incompatibilités de domaines d'attributs entre des classes liées par une passerelle, compte-tenu de ce qui a été dit dans le paragraphe précédent, consiste à considérer ces incompatibilités comme des incohérences, menant ainsi à la remise en cause de la passerelle (ou de la description des classes en relation). De ce fait, pour qu'une passerelle soit valide, le modèle doit s'assurer de la compatibilité des domaines d'attributs lorsque ces attributs sont différemment définis dans plusieurs classes liées par une passerelle.

Il existe dans TROPES deux types de passerelles :

- une passerelle *unidirectionnelle* possède une source (un ensemble de classes de points de vue différents) et une destination (une classe dans un autre point de vue) : toute instance de la source est instance de la destination. Une passerelle s'interprète dans le monde modélisé comme l'inclusion de la dénotation de la source[4] dans la dénotation de la destination.
- une passerelle *bidirectionnelle* relie deux classes exactement, de deux points de vue différents : toute instance de l'une est instance de l'autre. Une passerelle s'interprète, dans le monde modélisé, comme l'équivalence des dénotations des deux classes.

Tout comme l'instanciation et la spécialisation, les passerelles sont assertionnelles, le concepteur d'une base de connaissances pose une passerelle en la déclarant au niveau du concept, par l'identification d'un couple (`Source` liste de couples (classe/point de vue) `destination` classe/point de vue).

Les notions de passerelle et de spécialisation sont proches dans la mesure où les deux s'interprètent, dans le monde modélisé, par l'inclusion ensembliste. Pourtant, les phénomènes qu'elles représentent sont de natures très différentes :

- la spécialisation modélise un affinement progressif de la caractérisation des individus, à partir de propriétés ciblées qui sont précisées,

[4] lorsque la source est constituée de plusieurs classes, sa dénotation est l'intersection des dénotations de ces classes.

– une passerelle modélise la coopération entre différents groupes de propriétés caractéristiques.

Un phénomène d'inclusion ensembliste obervé dans le monde modélisé peut être représenté de deux façons différentes : par la spécialisation ou par la pose d'une passerelle. Le choix entre deux méthodes de représentation est alors guidé par la connaissance des propriétés que l'on a sur les catégories d'individus et par la nature de ces propriétés (figure 2.7).

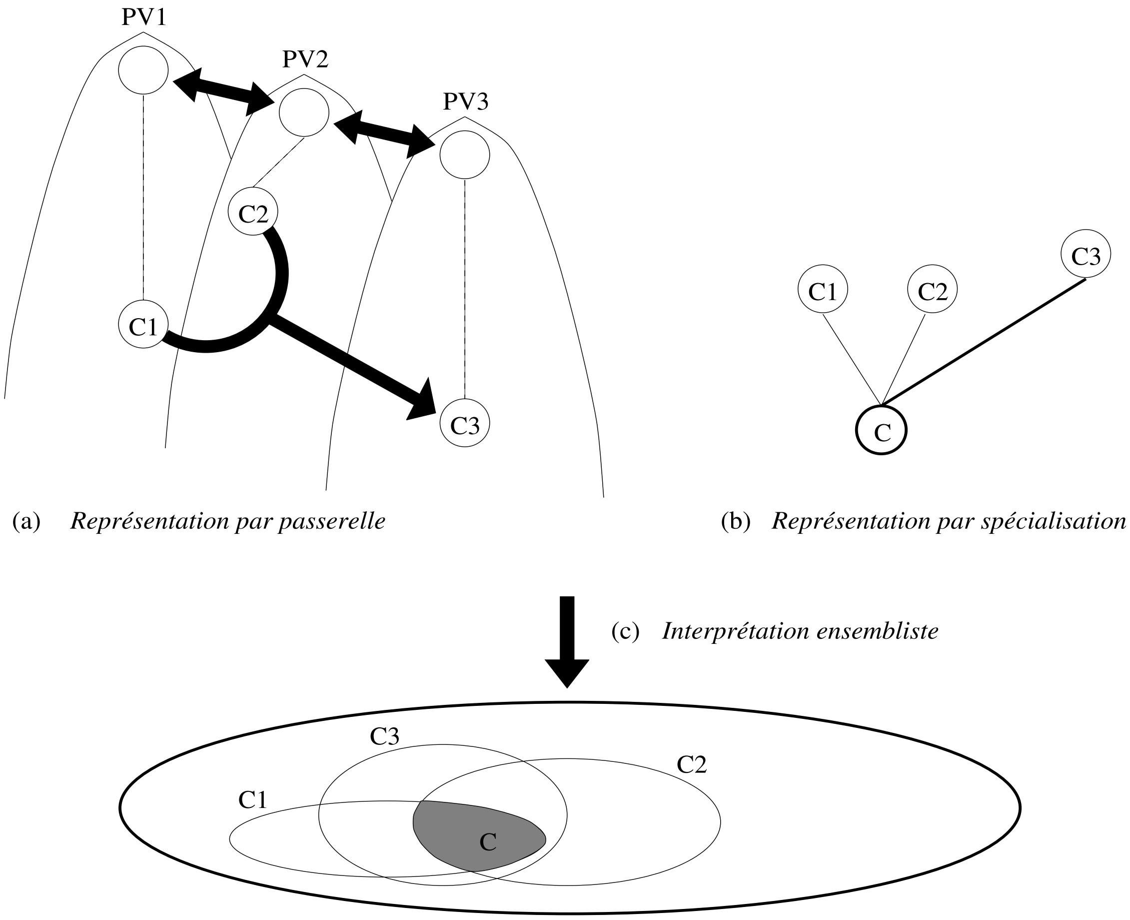


FIG. 2.7 - : La spécialisation multiple et la pose de passerelles peuvent avoir une interprétation ensembliste commune. Dans cette figure, les représentations (a) et (b) dénotent la même chose : tous les individus dénotés à la fois par C_1 et C_2 sont aussi dénotés par C_3. Au niveau du modèle, cela signifie que toute instance de C_1 et de C_2 (donc instance de C dans la représentation (b)) est aussi une instance de C_3.

En particulier, si l'on considère que la spécialisation peut se définir complètement (et donc être inférée) à partir des attributs définissant les classes, cela n'est pas vrai de l'existence et de la validité des passerelles, dont l'intérêt est justement de mettre en rapport des groupes d'attributs différents.

2.2 Sémantiques intensionnelle et extensionnelle

Toute tentative de description informelle des entités de représentation d'un modèle mène à un discours malheureusement ambigu, dans lequel deux niveaux de caractérisation sont constamment mêlés. Par exemple, à la question *comment décide-t-on qu'une instance appartient à une classe ?*, il y a deux réponses possibles :

– car l'instance vérifie les propriétés de la classe,

– car l'instance dénote un individu qui appartient à l'ensemble dénoté par la classe.

Cette ambiguité est due au fait que les entités de représentation et les relations entre ces entités sont présentées simultanément en fonction de leur signification vis-à-vis du monde modélisé et en fonction des propriétés qui sont à la base de leur identification. Nous proposons d'établir clairement la différence entre ces deux interprétations, commmunément appelées *extensionnelle* et *intensionnelle*.

Extension L'extension d'un concept (ou d'une classe) désigne l'ensemble des instances qui existent et lui appartiennent, autrement dit l'extension concerne ce qui peut être interprété dans le monde modélisé.

Intension L'intension d'un concept (ou d'une classe) fait référence à la description de ce concept (de cette classe). Il existe une interprétation partielle de cette description vers l'ensemble des individus de l'extension du concept (de la classe), ce qui nous amène souvent à confondre l'intension d'un concept (sa description) et l'ensemble des individus qui vérifient cette intension, c'est-à-dire l'interprétation de l'intension dans le monde. Ainsi, dire que l'intension d'un concept est l'ensemble des instances qui pourraient exister à partir de la description de ce concept est un abus de langage, mais il est commode car permet de cerner immédiatement la différence conceptuelle entre l'intension et l'extension d'un concept.

Dans le modèle TROPES, les concepts et les classes sont des *descriptions* et non pas des *définitions*. Ceci est la principale différence conceptuelle entre l'approche des langages terminologiques et celle d'un modèle tel que TROPES. Certaines instances peuvent être créées d'après la description d'un concept, sans que cette instance ne corresponde à quelque chose dans le monde modélisé (figure 2.8).

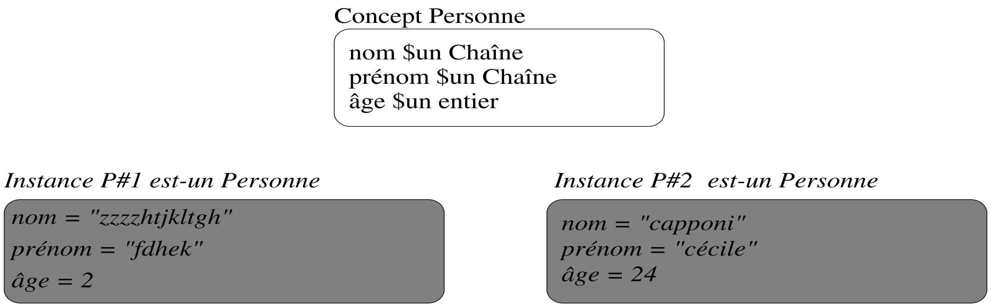


FIG. 2.8 - : Toute instance dont les valeurs d'attributs satisfont la description du concept `Personne`, comme les instances `P#1` et `P#2`, peuvent être créées, qu'elles correspondent ou non à quelqu'un existant ; elles valident l'intension du concept `Personne`. Les instances de `Personne` qui peuvent être créées et qui correspondent effectivement à quelqu'un, comme l'instance `P#2` mais certainement pas `P#1`, constituent l'extension du concept `Personne`.

Il est certain que l'adéquation entre l'intension et l'extension d'un concept est un critère de validité de la base de connaissances : plus l'intension et l'extension sont proches, meilleure est la caractérisation des propriétés qui définissent ce que l'on cherche à modéliser. L'équivalence entre l'intension et l'extension d'une entité signifie que l'ensemble des propriétés attribuées à l'entité établit une définition de ce que cette entité dénote.

La définition sémantique d'un modèle de connaissances nécessite de refléter ces deux interprétations des entités, d'étudier leurs incidences aux niveaux des relations et par là même au niveau des processus d'exploitation et d'inférence du système. C'est pour cette raison que nous insistons, dans cette section, sur les définitions, en intension et en extension, à la fois des entités de représentation, mais aussi des relations entre ces entités.

2.2.1 Interprétations des entités de représentation

Définition 1 (Extension d'un concept) *L'extension d'un concept K, notée $E(K)$, désigne l'ensemble des instances qui peuvent être créées à partir de K et qui ont une interprétation dans le monde modélisé. L'extension d'une classe désigne toutes les instances qui appartiennent à l'extension du concept et qui sont membres de la classe.*

Concepts et classes peuvent ainsi être interprétés *extensionnellement* par une fonction dont le domaine est le monde modélisé. La figure 2.1 illustre en fait l'interprétation extensionnelle d'une base de connaissances décrite par deux concepts, assimilée à une sémantique dénotationnelle.

Définition 2 (Intension d'un concept) *L'intension d'un concept K, notée $I(K)$, désigne la description en attributs de K. Par interprétation ensembliste de cette structure, on utilise parfois le raccourci consistant à dire que l'intension d'un concept est l'ensemble des structures d'instances qui peuvent être appariées à celle du concept.*

$$I(K) = \{a_1 : d_1, \cdots, a_n : d_n\}$$

($\{a_1 : d_1, \cdots, a_n : d_n, A\}$ est une abbréviation de la notation de l'intension d'un concept, où A désigne une agrégation d'attributs que l'on ne cherche pas à préciser). De la même façon, l'intension d'une classe C désigne l'ensemble des instances qui appartiennent à l'intension du concept et qui vérifient les conditions sur les attributs apportées dans la description de C.

Tout concept et toute classe peuvent ainsi être interprétés *intensionnellement* par une fonction dont le domaine est l'ensemble des données (valeurs) issues des structures d'agrégation décrivant concepts et classes. L'intension d'une entité peut effectivement être considérée indépendamment de son extension, ce qui revient à ne pas considérer la raison d'être de ces structures.

L'extension et l'intension sont par définition des ensembles d'instances, l'un s'attachant à considérer la dénotation de ces instances, l'autre reflétant l'intérêt porté sur leur structure.

Une instance i d'un concept K doit appartenir à l'intension et à l'extension de ce concept. Aucun moyen "informatique" ne permet pourtant de vérifier la validité de l'appartenance d'une instance à l'extension de son concept, c'est-à-dire de vérifier que cette instance est interprétable dans le monde modélisé. Par contre, l'appartenance à l'intension est calculable sur la base des propriétés qui définissent cette intension. Cela démontre l'importance des intensions car elles seules permettent de vérifier non pas la validité d'une instanciation mais sa possibilité (un des deux critères d'appartenance est vérifié ou réfuté). Le comportement des modèles de connaissances consiste alors à tester la possibilité d'une instanciation en vérifiant l'adéquation des structures de l'instance et de son concept. En cas de non-compatibilité de ces structures (l'instance ne vérifie pas au moins une des propriétés du concept), l'instanciation est automatiquement refusée puisque l'instance n'appartient pas à l'intension du concept. Dans le cas contraire, le modèle ne pouvant décider de l'appartenance à l'extension, il ne peut qu'en préciser sa possibilité intensionnelle, et donner alors les pleins pouvoirs à l'utilisateur (concepteur) de la base de connaissances, qui lui seul apprécie la signification de cette instance dans le monde modélisé. C'est en ce sens que l'instanciation est assertionnelle dans les modèles de connaissance, *une instanciation "assertée" signifiant l'appartenance de l'instance à* *l'extension du concept.* Autrement dit, lorsque l'utilisateur instancie un concept et que les conditions sur les attributs sont vérifiées, le modèle considère que cette instance fait partie de l'extension du concept.

Il est important de remarquer que les attributs complexes (dont le domaine est celui d'un concept) peuvent se faire appliquer cette double sémantique : un tel attribut fait-il référence à l'intension ou à l'extension de l'entité qui constitue son domaine ? La signification est différente, et c'est pour cette raison qu'il est pertinent de parler de l'intension et de l'extension d'un attribut complexe. En ce qui concerne les attributs élémentaires, parler de leur extension n'a pas de sens puisque leur domaine n'admet pas de dénotation dans le monde modélisé, il ne peut s'interpréter que dans un espace de valeurs.

Nous autorisons par la suite les abus de langage qui consistent à parler d'intension et d'extension d'une instance (les termes exacts seraient respectivement *valeur intensionnelle* et *valeur extensionnelle*), pour désigner respectivement la structure et la dénotation de l'instance. L'intension d'une instance est notée $I(i) = \{a_1 = v_1, \cdots, a_n = v_n\}$ pour désigner la structure de cette instance en attributs (propriétés) valués.

Par la suite, $\|O\|^E$ désignera l'interprétation extensionnelle d'une entité O, tandis que $\|O\|^I$ correspondra à son interprétation intensionnelle.

L'ambiguité qui porte sur tout discours impliquant une entité de représentation peut maintenant être levée en précisant laquelle, de l'intension ou de l'extension de l'entité, est considérée dans ce discours. Ceci peut être étendu pour éviter les ambiguités portant sur les relations d'instanciation, de spécialisation et sur les passerelles.

2.2.2 Définitions sémantiques des relations

À partir des deux interprétations possibles des termes de représentation, la définition sémantique des relations impliquant ces termes devient plus précise car elle profite elle aussi de la levée des ambiguités. Nous proposons dans cette section d'énoncer les interprétations intensionnelles et extensionnelles des divers liens entre entités.

L'interprétation extensionnelle est attachée à refléter ce qu'un lien entre entités de représentation signifie dans le monde modélisé, tandis que l'interprétation intensionnelle permet de considérer ces relations du point de vue des propriétés représentées. De ce fait, les définitions ci-dessous sont une précision de la définition qu'il a été donnée informellement dans la section 2.1.

Définition 3 (Instanciation extensionnelle) *L'interprétation extensionnelle de l'instanciation est notée $\in_E$. Elle est définie entre une instance I et un concept K, et entre une instance et une classe C.*

$$I \in_E K \iff \|I\|^E \in \|K\|^E$$

$$I \in_E C \iff \|I\|^E \in \|C\|^E$$

Comme cela a été précisé dans la section précédente, le modèle ne peut vérifier l'instanciation extensionnelle, et s'en remet donc à l'utilisateur (concepteur) de la base de connaissances.

Partant de l'hypothèse que, dans TROPES, les classes et les concepts sont des *descriptions*, cela signifie que pour appartenir à un concept (à une classe), une instance doit *nécessairement* vérifier les propriétés qui décrivent le concept (la classe). Ces propriétés constituant l'intension des entités de représentation, l'interprétation intensionnelle de l'instanciation concerne le respect par l'instance des conditions portées sur les propriétés du concept (de la classe), c'est-à-dire que les valeurs qu'associe une instance à ses attributs doivent appartenir aux domaines de valeurs spécifiés pour les attributs correspondant dans la classe.

Définition 4 (Instanciation intensionnelle) *L'interprétation intensionnelle de l'instanciation est notée* $\in_I$*. Elle est définie entre une instance* $I = \{a_1 = v_1, \cdots, a_k = v_k\}$ *et un concept* $K = \{a_1 : d_1, \cdots, a_n : d_n\}$*, et entre une instance et une classe* $C = \{a_1 : d'_1, \cdots, a_p : d'_p\}$*.*

$$\bullet I \in_I K \iff \|I\|^I \in \|K\|^I$$
$$\iff k \leq n, \forall i \in [1..k], v_i \textit{ est inconnue ou} \begin{cases} v_i \in_I d_i & \textit{si } a_i \textit{ est complexe} \\ v_i \in d_i & \textit{sinon} \end{cases}$$

$$\bullet I \in_I C \iff \|I\|^I \in \|C\|^I$$
$$\iff p \leq k \leq n \textit{ et } \forall i \in [1..p], v_i \textit{ est inconnue ou} \begin{cases} v_i \in_I d'_i & \textit{si } a_i \textit{ est complexe} \\ v_i \in d'_i & \textit{sinon} \end{cases}$$

Cette définition de l'instanciation intensionnelle est une définition d'appartenance large, dans la mesure où les valeurs inconnues ne sont pas pénalisantes. Lorsque l'on parlera d'instanciation intensionnelle *stricte*, cela signifiera qu'elle ne peut être admise dès qu'une valeur de l'instance est inconnue.

Contrairement à l'instanciation extensionnelle, le modèle est capable de vérifier automatiquement l'instanciation intensionnelle, large ou stricte, en procédant aux vérifications des conditions imposées sur les attributs.

Définition 5 (Spécialisation extensionnelle) *L'interprétation extensionnelle de la spécialisation est notée* $\leq_{\sigma,E}$*. Elle est définie entre deux classes* C_1 *et* C_2*, d'un même point de vue d'un même concept.*

$$\begin{aligned} C_1 \leq_{\sigma,E} C_2 &\iff \|C_1\|^E \subseteq \|C_2\|^E \\ &\iff \forall I \in_E C_1, I \in_E C_2 \end{aligned}$$

La deuxième partie de la condition portant sur la spécialisation extensionnelle peut être vérifiée ponctuellement par le modèle, s'il considère qu'au moment t de la vérification, l'extension des classes correspond aux instanciations constatées à l'instant t. En ce qui concerne la vérification de la spécialisation intensionnelle, elle peut être vérifiée sur la base de vérifications portant sur les domaines des attributs, selon le même principe que la vérification de l'instanciation intensionnelle.

Définition 6 (Spécialisation intensionnelle) *L'interprétation intensionnelle de la spécialisation est notée* $\leq_{\sigma,I}$*. Elle est définie entre deux classes* $C_1 = \{a_1 : d_1^1, \cdots, a_p : d_p^1\}$ *et* $C_2 = \{a_1 : d_1^2, \cdots, a_n : d_n^2\}$*, d'un même point de vue d'un même concept.*

$$C_1 \leq_{\sigma,I} C_2 \iff p \leq n \textit{ et } \forall i \in [1..p] \begin{cases} d_i^1 \leq_{\sigma,I} d_i^2 & \textit{si } a_i \textit{ est un attribut complexe} \\ d_i^1 \subseteq d_i^2 & \textit{sinon} \end{cases}$$

La relation de spécialisation est transitive, $C \leq_\sigma C', C' \leq_\sigma C'' \Longrightarrow C \leq_\sigma C''$. Par convention, la notation $C_1 \leq_\sigma C_2$ signifiera que $\nexists C$ tel que $C_1 \leq_\sigma C \leq_\sigma C_2$. De plus, la spécialisation est reflexive ($C_1 \leq_\sigma C_2$ et $C_2 \leq_\sigma C_1 \Rightarrow C_1 = C_2$), et antisymétrique ($C_1 \neq C_2, C_1 \leq_\sigma C_2 \Rightarrow C_2 \not\leq_\sigma C_1$). Par conséquent, la spécialisation dans TROPES est un ordre partiel, comparable en ce sens à la relation de subsomption.

Ces interprétations extensionnelle et intensionnelle de la spécialisation sont issues de celles de l'instanciation. Elles sont définies pour que les deux conditions suivantes soient respectées : $C_1 \leq_{\sigma,E} C_2 \Longleftrightarrow \forall I \in_E C_1, I \in_E C_2$ et $C_1 \leq_{\sigma,I} C_2 \Longleftrightarrow \forall I \in_I C_1, I \in_I C_2$. Selon le même critère, nous définissons les deux interprétations possibles d'une passerelle.

Définition 7 (Passerelle extensionnelle) *L'interprétation extensionnelle d'une passerelle est notée $\gg_E$. Elle est définie entre un ensemble de couples $\langle C_1, P_1\rangle \cdots \langle C_n, P_n\rangle$ précisant des classes toutes issues de points de vue différents, et un couple $\langle C, P\rangle$ qui correspond à une classe dans un point de vue autre que $P_1 \cdots P_n$. Une telle passerelle est formellement notée $\{\langle C_1, P_1\rangle \cdots \langle C_n, P_n\rangle\} \gg_E \langle C, P\rangle$, ou plus simplement $\{C_1, \cdots, C_n\} \gg_E C$, en confondant implicitement les descriptions des classes et leurs identificateurs.*

$$\{C_1, \cdots, C_n\} \gg_E C \Longleftrightarrow \bigcap_{i=1..n} \|C_i\|^E \subseteq \|C\|^E$$

Une passerelle est un lien entre différentes structures de classes. De ce fait, son interprétatiom intensionnelle ne peut porter que sur les parties de ces structures qui sont comparables. En ce sens, l'interprétation intensionnelle d'une passerelle s'attache à fixer des contraintes sur les attributs qui sont déclarés et définis sous plusieurs points de vue, de façon à s'assurer que les domaines associés aux attributs de la source ne sont pas incompatibles avec les domaines des attributs correspondants dans la destination. Ainsi, la définition en intension d'une passerelle correcte pose des conditions sur les domaines des attributs, et ce à deux niveaux :

- tout d'abord, pour s'assurer que cette passerelle est pertinente, il convient de vérifier qu'il existe des instances qui peuvent appartenir simultanément à toutes les classes sources : pour cela, on vérifie que l'intersection des domaines d'attributs communs à plusieurs classes sources n'est pas vide,
- ensuite, il est fait état des attributs partagés par la destination et au moins une des classes sources : l'intersection des domaines d'un tel attribut pris dans chacune des classes sources qui le contiennent doit être inclus dans le domaine spécifié pour cet attribut dans la classe destination.

Notations 1 *Par la suite, nous considérons les notations suivantes :*

- *Pour toute classe C (resp. concept K), $A(C)$ (resp. $A(K)$) désigne l'ensemble des identificateurs d'attributs que la classe C (resp. concept K) possède ou hérite. Le symbole a désigne un identificateur d'attribut ; ainsi, $\forall C, A(C) = \{a_1, \cdots, a_n\}$.*
- *Pour toute classe C (resp. concept K), $\widetilde{A(C)}$ (resp. $\widetilde{A(K)}$) désigne l'ensemble des descriptions d'attributs que la classe C (resp. concept K) possède ou hérite. Le symbole $\tilde{a}$ désigne une description d'attribut, $\tilde{a} = a : d$; ainsi, $\forall C, \widetilde{A(C)} = \{\tilde{a}_1, \cdots, \tilde{a}_n\}$.*

– *Pour toute classe C, nous rappelons que son agrégation de descriptions d'attributs peut être notée $\{a_1 : d_1, \cdots, a_n : d_n, \tilde{A}\}$ où $\tilde{A}$ est une agrégation de descriptions d'attributs qu'il n'est pas utile de décrire.*

Définition 8 (Passerelle intensionnelle) *L'interprétation intensionnelle d'une passerelle est notée $\gg_I$. Elle est définie entre un ensemble de couples $\langle C_1, P_1\rangle \cdots \langle C_n\ , P_n\rangle$ précisant des classes toutes issues de points de vue différents, et un couple $\langle C, P\rangle$ qui correspond à une classe dans un point de vue autre que $P_1 \cdots P_n$. Une telle passerelle est formellement notée $\{\langle C_1, P_1\rangle \cdots \langle C_n, P_n\rangle\} \gg_I \langle C, P\rangle$, ou plus simplement $\{C_1, \cdots, C_n\} \gg_I C$, en confondant implicitement les descriptions des classes et leurs identificateurs. Soient $C_1, \cdots, C_n$ et C définies par :*

$$C_1 = \{a_1 : d_1^1, \cdots, a_m : d_m^1, \tilde{A}^1\}$$
$$\cdots$$
$$C_n = \{a_1 : d_1^n, \cdots, a_p : d_p^n, \tilde{A}^n\}$$
$$C = \{a_1 : d_1, \cdots, a_q : d_q, \tilde{A}\}$$

Alors :

$$\begin{aligned}
&\{C_1, \cdots, C_n\} \gg_I C \\
&\qquad \Longleftrightarrow \textstyle\bigcap_{i\in[1..n], a\in A(C_i)} \tilde{a}^i \neq \emptyset \\
&\qquad et \\
&\qquad\qquad \forall i \in [1..q], \forall j \in [1..p] \ \textit{tel que } C_i \textit{ possède } a_j \\
&\qquad\qquad \begin{cases} \forall I \in_I d_j^i, I \in_I d_j & \textit{si } a_j \textit{ est un attribut complexe} \\ \bigcap_{i, C_i \textit{ possède } a_j} d_j^i \subseteq d_j & \textit{sinon} \end{cases}
\end{aligned}$$

La première de ces conditions assure que les intensions des classes sources ne sont pas disjointes, et la seconde garantit l'inclusion des domaines d'attributs, considérés conjointement dans les classes sources, dans les domaines des attributs correspondant dans la classe destination.

2.2.3 Coopération des interprétations

Tropes a une sémantique fondée sur la coopération entre les deux interprétations, intensionnelle et extensionnelle. Pour qu'une assertion en Tropes soit acceptée, c'est-à-dire la représentation d'une connaissance intégrée aux autres, elle doit être cohérente extensionnellement et intensionnellement. Cela signifie que le modèle Tropes doit être capable de procéder à ces vérifications de cohérence. Pourtant, la seule vérification dont est capable le système est celle fondée sur le respect des propriétés, à savoir la vérification de l'interprétation intensionnelle.

Définition 9 (Instanciation, spécialisation et passerelles dans Tropes)

– *Instanciation*

$$\forall I, \forall K, I \in K \Longleftrightarrow (I \in_I K) \ et \ (I \in_E K)$$
$$\forall I, \forall C, I \in C \Longleftrightarrow (I \in_I C) \ et \ (I \in_E C)$$

– *Spécialisation*

$$\forall C_1, C_2, C_1 \leq_\sigma C_2 \Longleftrightarrow (C_1 \leq_{\sigma,I} C_2) \ et \ (C_1 \leq_{\sigma,E} C_2)$$

– *Passerelle*

$$\forall C_1, \cdots, C_n, C, \{C_1, \cdots, C_n\} \gg C \Longleftrightarrow (\{C_1, \cdots, C_n\} \gg_I C) \; et \; (\{C_1, \cdots, C_n\} \gg_E C)$$

Partant de l'hypothèse que la description des propriétés caractéristiques d'une entité est correcte vis-à-vis de la signification de cette entité dans le monde modélisé, TROPES établit ses vérifications sur la structure des entités. Lors d'une instanciation, à partir du moment où les propriétés sont vérifiées, l'appartenance à l'intension est acceptée, et l'appartenance à l'extension est alors considérée comme possible par le système, elle est décidée par le créateur de l'instance, seul médiateur entre le monde et la représentation qui en est faite dans le modèle. Lorsqu'une instanciation est ainsi validée par l'utilisateur, le système considère que l'instance appartient désormais à l'extension du concept de rattachement, autrement dit le système reconnaît que cette instance a une signification. La conséquence immédiate de ce comportement du système est qu'il construit, au fur et à mesure des créations d'instances, l'extension des classes et concepts.

En ce qui concerne la spécialisation et les passerelles, la vérification qu'effectue TROPES n'est qu'intensionnelle. De la même façon que dans le cas de l'instanciation, c'est la décision de l'utilisateur, quant à la création effective d'un lien qui établit l'acceptation extensionnelle du modèle. Cependant, TROPES utilise l'interprétation extensionnelle de ces relations pour compléter justement sa connaissance de l'extension des classes, de la façon suivante :

– d'après la définition 5 de la spécialisation, si C_1 est une sous-classe de C_2, alors toute instance de C_1 est aussi instance de C_2.
– d'après la définition 7 d'une passerelle, s'il existe une passerelle de $C_1, \cdots, C_n$ vers C, alors toute instance de $C_1, \cdots, C_n$ est aussi une instance de C.

Le statut de l'interprétation intensionnelle vis-à-vis de l'interprétation extensionnelle tel que TROPES le considère est caractérisé par la définition suivante :

Définition 10 *Pour toute entité de représentation O, pour toute relation R entre entités,*

$$\|O\|^E \subseteq \|O\|^I \qquad et \qquad \|R\|^E \subseteq \|R\|^I$$

La définition précédente établit que si un objet ou une relation ne peut être interprété intensionnellement, alors il ne peut pas l'être extensionnellement. Le fait que les structures des entités ne soient pas des définitions, mais seulement des descriptions, explique la non-égalité des deux interprétations.

La définition 10 guide le comportement du modèle, elle pose les bases de la coopération intension/extension qui devra être respectée par tous les mécanismes d'exploitation des représentations de la connaissance. En résumé, les vérifications intensionnelles sont une étape à l'acceptation des assertions, elles constituent un processus de vérification de correction. Ainsi nous supposons dans TROPES que **la validation intensionnelle est nécessaire à la validation extensionnelle**. Cette hypothèse est acceptable dans un tel modèle multi-points de vue du fait de l'existence de passerelles qui permettent de créer des équivalences entre différentes descriptions, intensionnellement incomparables. C'est un moyen de rejeter l'objection énoncée par William Woods [Woo91] qui refusait que l'interprétation intensionnelle soit une condition nécessaire à l'interprétation extensionnelle, en affirmant que la subsomption intensionnelle ne permettait pas d'inférer certains liens déductifs[5]. Dans le cas de TROPES, la spécialisation ne permet pas non plus de lier deux structures différentes même si leurs extensions correspondent, mais la pose d'une passerelle le permet.

Par ailleurs, TROPES ne considère pas la validation intensionnelle comme suffisante à la validation extensionnelle, car les descriptions des classes et concepts sont certes correctes mais rien ne garantit leur complétude. La validation extensionnelle ne peut que résulter d'assertions de l'utilisateur.

2.2.4 Propriétés des taxonomies

Les hiérarchies des points de vue sont des arbres qui possèdent des propriétés organisationnelles issues d'hypothèses faites sur la modélisation du monde et sa représentation. Nous retenons dans TROPES deux propriétés majeures et réalistes qui ont été identifiées lors de l'élaboration de l'algorithme de classification d'instance (section 2.3.3). Il s'agit des propriétés d'exhaustivité et d'exclusivité, la première stipulant que tout individu pouvant être attaché à une classe peut être attaché à l'une de ses sous-classes, et la seconde établissant que les sous-classes d'une même classe (appelées communément sœurs) sont disjointes. Ces propriétés s'interprètent extensionnellement car elles ont une signification dans le processus de modélisation, et intensionnellement car elles ont des répercussions sur les descriptions de classes.

Définition 11 (Exhaustivité) *Toute taxonomie d'un point de vue est exhaustive.*

- *Du point de vue extensionnel, cela se traduit par*

$$\forall C \text{ telle que } \exists C' \leq_{\sigma,E} C, \forall I \in_E C, \exists C'' \leq_{\sigma,E} C \text{ et } I \in_E C''$$

- *Intensionnellement, cette propriété se traduit par des conditions sur les spécifications des attributs*
  Soit $C = \{a_1 : d_1, \cdots, a_n : d_n\}$, *avec* $\exists p \geq 1$ *tel que* $\forall i \in [1..p], \exists C_i \leq_{\sigma,I} C$,
  avec $\forall i, C_i = \{a_1 : d_1^i, \cdots, a_n : d_n^i, \tilde{A}^i\}$

$$\forall k \in [1..n], d_k = \bigcup_{i=1..p} d_k^i$$

Tout comme l'exhaustivité, l'exclusivité est une propriété de toutes les taxonomies d'un concept, qui a une interprétation double. L'exclusivité correspond à la disjonction exclusive.

Définition 12 (Exclusivité) *Toute taxonomie d'un point de vue est exclusive.*

- *Du point de vue extensionnel, cela se traduit par*

$$\forall C, C', C'' \text{ telles que } C' \leq_{\sigma,E} C, C'' \leq_{\sigma,E} C, \text{ et } \|C'\|^E \neq \|C''\|^E$$

$$\forall I \in_E C, I \in_E C' \Longrightarrow I \notin_E C''$$

[5] "For example, [polygon with three sides] would extensionnaly subsume [polygon with three angles], but there is no direct structural correspondence between the two descriptions that would imply this" [Woo91].

- *Intensionnellement, cette propriété se traduit par des conditions sur les spécifications des attributs*

  Soit $C = \{a_1 : d_1, \cdots, a_n : d_n\}, \forall C_1$ *et* $C_2 \leq_{\sigma,I} C$,
  $C_1 = \{a_1 : d_1^1, \cdots, a_n : d_n^1, \tilde{A}^1\}, C_2 = \{a_1 : d_1^2, \cdots, a_n : d_n^2, \tilde{A}^2\}$

$$\exists i \in [1..n], d_i^1 \cap d_i^2 = \emptyset$$

Le comportement du modèle face à la vérification de ces propriétés est identique à celui établi pour la vérification de la spécialisation et des passerelles : le respect de l'interprétation intensionnelle est nécessaire pour l'acceptation extensionnelle, cette dernière étant encore une fois validée par l'utilisateur.

Toute taxonomie exclusive et exhaustive est telle que l'ensemble des sous-classes directes d'une classe C représente une partition de C (figure 2.9), c'est-à-dire

$$\forall C, \text{ soient } C_1, \cdots, C_n \leq_{\sigma,E} C, \forall i,j \in [1..n], i \neq j, \|C_i\|^E \cap \|C_j\|^E = \emptyset \text{ et } \bigoplus_{k=1..n} \|C_k\|^E = \|C\|^E$$

($\oplus$ représente l'union disjointe).

Il s'agit d'une propriété relativement rigide car elle exige une représentation complète à chaque niveau de spécialisation. Pourtant, cela relève d'un principe de modélisation rigoureux, qui oblige le concepteur d'une base à caractériser extensionnellement toute tentative de spécialisation d'une catégorie d'individus. La rigidité de cette propriété, et les difficultés de représentation qu'elle engendre, dépendent de la caractérisation intensionnelle (en termes de propriétés) de chaque étape de spécialisation.

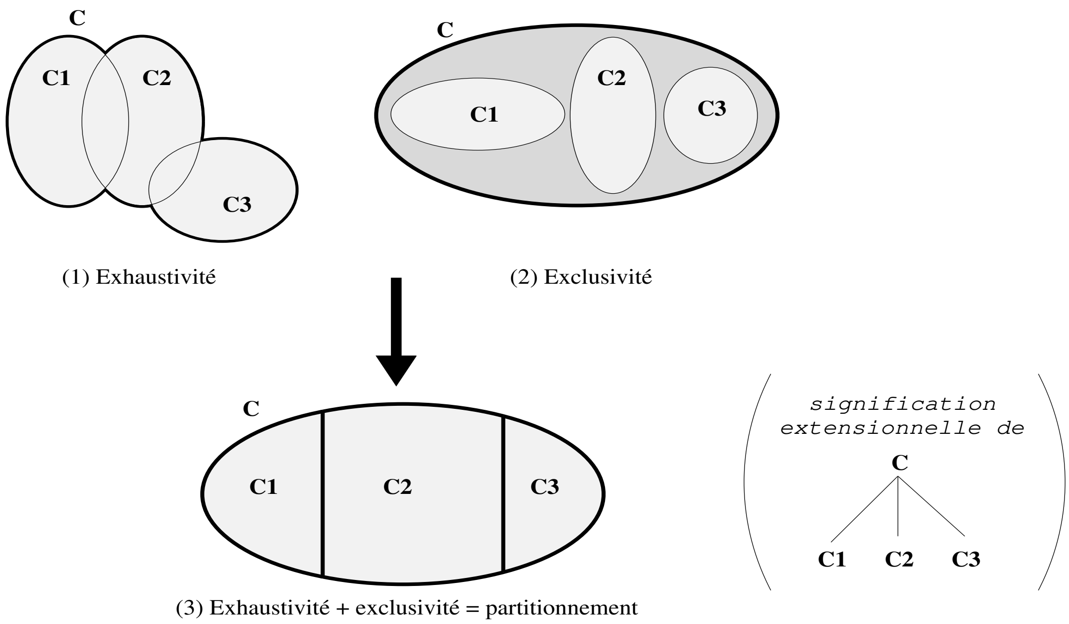


FIG. 2.9 - : Représentation extensionnelle des contraintes portant sur la relation de spécialisation.

2.3 Mécanismes de Tropes

Les concepts, classes, instances et attributs, sont des *entités de représentation*, leur rôle en tant que telles est de permettre de représenter la modélisation d'un domaine d'application, de telle façon que les connaissances ainsi décrites, en termes de propriétés et de relations, soient accessibles et compréhensibles sans la mise en œuvre de processus dynamiques de déduction.

Sur ces descriptions statiques que sont les entités de représentation, le modèle TROPES permet la réalisation d'inférences, en développpant des mécanismes qui permettent de rendre accessibles des connaissances implicitement présentes dans une base de connaissances. Deux groupes de mécanismes d'inférence se distinguent, ceux qui réalisent le calcul de valeurs d'attributs pour des instances particulières, et ceux qui exploitent ces valeurs d'attributs pour affiner l'identification des instances en les descendant dans les hiérarchies. À ces deux catégories de mécanisme s'ajoute le processus d'héritage qui consiste à parcourir la hiérarchie de spécialisation afin de récupérer les informations factorisées.

2.3.1 Mécanisme d'héritage

Le mécanisme d'héritage est présent dans tous les systèmes à objets. Il ne s'agit pas d'un processus de déduction, mais d'un procédé issu de la factorisation de l'information le long des hiérarchies de classes.

La structure intensionnelle d'une classe est constituée d'une agrégation de propriétés représentées et spécifiées par des attributs. Une classe possède les propriétés de sa super-classe, qu'elle peut affiner en complétant leur spécification, et elle peut définir de nouvelles propriétés. Au niveau des structures de classes, cela se traduit par : une classe possède les attributs de sa super-classe, spécifiés par les descripteurs, qu'elle peut compléter par d'autres descripteurs ; une classe peut en outre ajouter des attributs et leur spécification à ceux déjà présents dans sa super-classe. Ainsi, on distingue trois moyens simultanés et complémentaires de spécialisation intensionnelle d'une classe :

1. récupérer, sans modification, des attributs de la super-classe et leurs descripteurs associés (attributs hérités),
2. récupérer des attributs de la super-classe et leurs descripteurs associés, et affiner la spécification de ces attributs par ajouts de descripteurs (attributs affinés),
3. définir de nouveaux attributs qui n'existent pas dans la super-classe (attributs ajoutés).

Dans les deux premiers cas, les spécifications des attributs existants, qu'elles soient complétées ou non, ne sont pas explicitement réécrites dans le schéma de la sous-classe : seuls les ajouts de descripteurs y sont précisés. Le mécanisme d'héritage est alors le processus qui permet d'accéder, depuis la sous-classe, à ces informations disponibles dans sa super-classe (figure 2.10). De même que la spécialisation, l'héritage est transitif. En outre, les graphes de spécialisation des points de vue étant des arbres, l'héritage dans TROPES est simple[6].

On distinguera la *définition partielle* d'un attribut de classe (celle explicitement fournie dans la structure de la classe), de sa *définition complète* obtenue après activation de l'héritage. Trivialement, la définition complète d'un attribut ajouté est la même que sa définition partielle.

[6] La notion d'héritage multiple est tout de même présente, pour l'instant conceptuellement, dans TROPES, grâce à l'existence des passerelles.

Classe Collective *(Concept HABITATION Point de vue DESCRIPTION)*

```
dénomination $un Chaîne $domaine ("Cours" "Boulevard" "Rue"
        "Chemin" "Passage" "Route" "Impasse" "Avenue" "Place")
numéro $un Entier  $intervalle [0 ; +inf]
nom $un Chaîne
code-postal $liste-de Entier $cardinalité [5 ; 5] $parmi [0 ; 9]
commune $un Commune
exposition $liste-de Chaîne $parmi ("Sud" "Est" "Ouest" "Nord")
          $cardinalité [1 ; 4]
chauffage $un Chaîne $domaine ("Collectif" "Mazout" "Gaz"
                         "Electrique" "sans" "solaire" "bois")
                   $sauf ("solaire")
nb-pièces $un Entier $sauf-intervalle [250 ; +inf]
                   $intervalle [1 ; 20]
surface $un Entier $intervalle [9 ; +inf]  $intervalle [9 ; 400]
ascenseur $un Booléen
voisins-immédiats $ens-de Personne
étage $un Entier $intervalle [0 ; 78]
```

FIG. 2.10 - : Le mécanisme d'héritage permet de compléter la description d'une classe, en récupérant l'information contenue dans ses super-classes, qu'elle ne recopie pas localement : on dit qu'une classe hérite les propriétés de sa super-classe. Ici, la description de la classe `Collective` est complète, elle est obtenue par activation de l'héritage le long du lien de spécialisation qui la lie à la classe `Habitation` (définie dans la figure 1.6). Les expressions en caractères gras indiquent ce qui est ajouté à `Habitation` par `Collective`, les autres expressions étant héritées.

L'héritage est un parcours ascendant (*bottom-up*) des liens de spécialisation, activés lors de l'accès à des spécifications d'attributs. Couplé avec l'instanciation, l'héritage consiste à parcourir les super-classes de la classe d'appartenance d'une instance afin de récupérer des spécifications contenues dans ces super-classes et directement applicables à l'instance (par exemple, récupération de la valeur du descripteur $valeur pour un attribut).

2.3.2 Inférences de valeurs d'attributs

TROPES autorise la création d'instances incomplètes (section 2.1.6), mais dispose de quatre moyens directs pour l'inférence de valeurs d'attributs. Ces mécanismes sont attachés à l'attribut considéré, dans la description de la classe contenant cet attribut. L'attachement d'un mécanisme d'inférence à un attribut de classe s'effectue au moyen de *descripteurs d'inférence*.

Attachement procédural

Le descripteur `$sib-exec`[7] permet d'associer à un attribut de classe une fonction paramétrée de calcul de la valeur de cet attribut, pour une instance de la classe (figure 2.11). Cette technique est classique dans les systèmes de représentation issus des frames, depuis que KRL l'a définie [BW77].

La fonction associée est une fonction du langage hôte, liée dynamiquement au schéma de la classe. Elle est activée lors de l'accès à une valeur d'attribut d'une instance, valeur jusque là restée inconnue. Pour éviter la mise en place d'un système de révision des connaissances, la valeur résultante de l'activation d'une fonction n'est pas stockée dans la structure de l'instance, elle sera re-calculée lors de chaque accès.

[7] Si besoin, exécute...

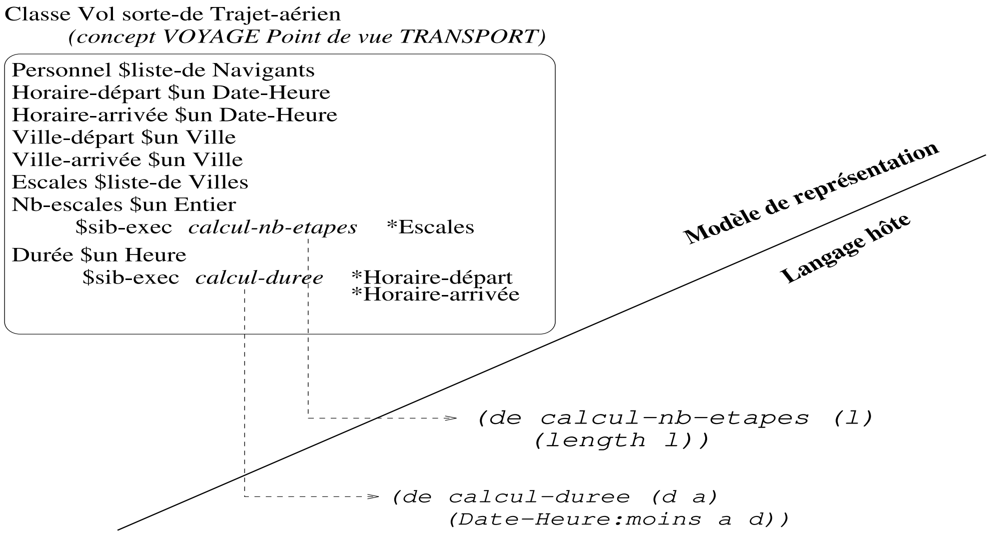


FIG. 2.11 - : L'attachement procédural. Des fonctions de calcul sont associées à des attributs de classes, et écrites dans le langage hôte. Le modèle de connaissances s'assure de gérer le lien entre la fonction appelée et son code. Le caractère étoilé (*) désigne l'objet pour lequel sera activé l'attachement procédural, cette notation est comparable au *self* rencontré dans les langages à objets.

Ce principe de calcul d'une valeur d'attribut est communément appelé *attachement procédural*[8], pour désigner le fait qu'une fonction est attachée à un attribut particulier, dans une classe particulière. L'activation d'une telle fonction n'est ainsi possible que pour des instances qui appartiennent à la classe, intensionnellement *et* extensionnellement. En réalité, TROPES implémente le *détachement procédural*, qui consiste à définir les fonctions de calcul au niveau des attributs du *concept*, de façon à ce qu'elles soient activables pour toute instance de ce concept, quelles que soient ses classes de rattachement [Rec93].

Terry Winograd a soutenu dans [Win85] que l'attachement procédural est un bon compromis en ce qui concerne la controverse déclaratif/procédural. D'une part, le contrôle de l'attachement procédural est effectué par le système de représentation, qui associe à ce mécanisme une sémantique commune à tous les objets. D'autre part, l'attachement procédural est un processus relativement puissant en ce qui concerne la connaissance dynamique. Pour préserver le contrôle du système, une propriété nécessaire de l'attachement procédural est qu'il ne doit rien modifier d'autre que la valeur de l'attribut qu'il calcule ; le fait que la fonction soit définie à l'extérieur, et non pas dans l'objet, garantit le respect de cette propriété.

Défaut

Le descripteur inférentiel `$défaut` permet de traduire la connaissance prototypique [Bra85] [Kay84] que l'on a sur certaines catégories d'individus. À ce descripteur est associée une valeur particulière pour l'attribut, qui est typique des individus de cette classe (figure 2.12).

De la même façon que dans le cas de l'attachement procédural, la valeur par défaut est considérée lors de la consultation d'une valeur d'attribut inconnue. Le comportement du modèle est ainsi de considérer qu'en l'absence d'autres informations sur cette valeur d'attribut, la valeur prototypique est reconnue comme valeur de l'attribut, mais peut être bien entendu remise en cause ultérieurement.

[8] Le terme *procédural* n'est pas, ici, à opposer à la notion de fonction, mais plutôt à la notion de *déclaratif*.

```
Classe Éléphant sorte-de Gros-être-vivant
        (concept ETRE-VIVANT Point de Vue ASPECT)

couleur $un Chaîne
        $domaine ("gris" "blanc")
        $défaut "gris"
longueur-trompe $un Entier
                $intervalle [30..187]
```

FIG. 2.12 - : En règle générale, tous les éléphants sont gris, ce que permet de représenter la facette `défaut`, sans rejeter pour autant le fait qu'il existe des éléphants albinos

TROPES ne considère pas le mécanisme dual de celui de la gestion des défauts, à savoir les exceptions [Pag92].

Que ce soit la spécification de valeurs par défaut ou la donnée de fonctions de calcul de valeurs d'attributs, il s'agit de purs moyens d'inférence de valeurs, qui ne peuvent en tant que tels intervenir dans les critères d'appartenance à une classe. Par exemple, dans l'exemple de la figure 2.12, ce n'est pas parce qu'une instance possède toutes les caractéristiques d'un éléphant, exceptée que sa couleur est blanche, qu'il ne s'agit pas d'un éléphant. Cependant, la distance d'une valeur d'attribut à la valeur par défaut peut indiquer le degré de typicalité de l'instance ainsi caractérisée.

Filtrage

Le descripteur **`$sib-filtre`**[9] est utilisé pour inférer des valeurs d'attributs, au même titre que les deux précédents mécanismes. Un filtre est associé à un attribut, il est posé sur un concept ou un ensemble de classes dont il sélectionne un ensemble d'instances satisfaisant les critères énoncés par le filtre. Une des instances candidates est alors associée à l'attribut pour lequel est spécifié le filtre.

La structure d'un filtre est similaire à la structure d'une classe : elle consiste à fixer un ensemble de conditions sur les domaines de certains attributs du concept balayé par le filtre. Un filtre permet en outre de fixer des contraintes dynamiques entre ces attributs (figure 2.13).

De la même façon que l'attachement procédural, l'activation d'un filtre est dynamique. Sa résolution est actuellement triviale, elle consiste à parcourir séquentiellement l'extension du concept (ou de l'ensemble des classes) sur lequel est posé le filtre ; la première instance testée qui respecte le filtre constitue le résultat de ce filtre[10].

Pose de contraintes

TROPES dispose d'un module consacré à la pose de contraintes et à leur gestion. MICRO, développé pour TROPES par Jérôme Gensel [Gen93], pré-définit un ensemble de contraintes génériques d'une grande variété, parmi lesquelles des contraintes portant sur un attribut, sur un ensemble d'attributs, ou directement sur une entité, que ce soit au niveau du concept, de la classe ou de l'instance.

Contrairement aux attachements procéduraux, filtres ou défauts, la pose d'une contrainte sur un

---

[9] Si besoin, filtre...

[10] dans le cas où le filtre est déclaré pour un attribut dont le type principal est multivalué, toutes les instances qui satisfont le filtre sont retenues, tout au moins tant que la cardinalité est respectée.

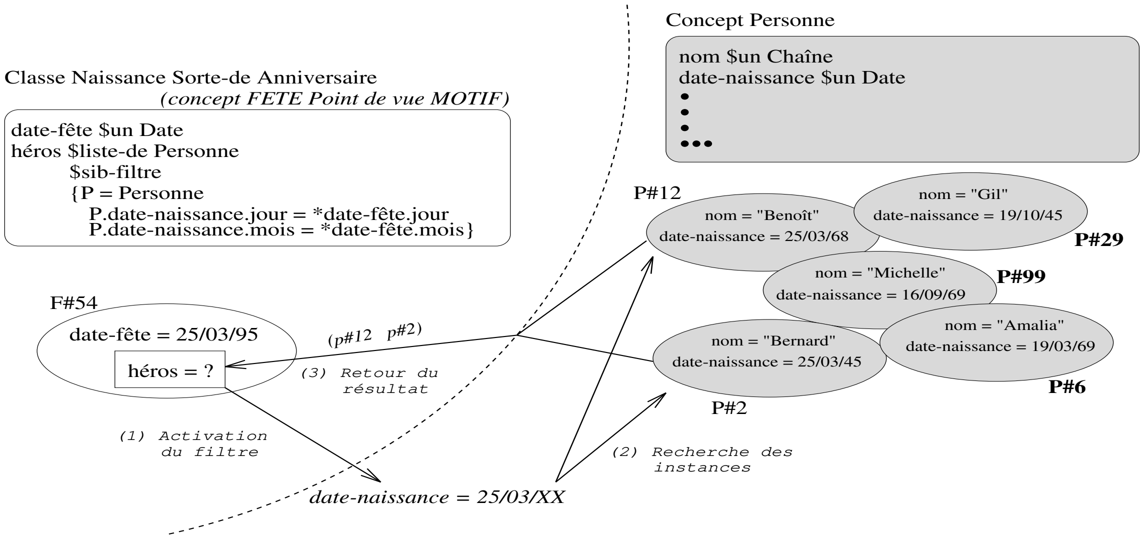


FIG. 2.13 - : L'anniversaire de naissance fêté le 25/03/95 est, compte tenu de l'extension du concept `Personne`, celui de `Benoît` et de `Bernard`. Cette information est inférée grâce au filtre posé sur l'attribut `héros` de la classe `Naissance`, dont l'activation consiste ici à chercher toutes les `Personnes` qui sont nées le même jour et le même mois que la date fêtée.

attribut n'est pas attachée explicitement à cet attribut par le biais d'un descripteur d'inférence. Une contrainte sur un attribut, ou entre des attributs, fait partie de la description de l'entité englobante (instance, classe ou concept).

Le statut d'une contrainte est tout autant inférentiel que descriptif.

- Certaines contraintes permettent d'inférer des valeurs d'attributs, au même titre que l'attachement procédural ou que le filtrage. Ce type de contrainte se rapproche d'ailleurs des attachements procéduraux, la différence majeure étant qu'elles sont écrites à partir de primitives définies, et donc interprétables, par MICRO.
- D'autres contraintes ne permettent pas d'inférer des valeurs, mais restreignent le domaine de l'attribut sur lequel est posé la contrainte, à partir de valeurs connues des autres attributs impliqués dans la contrainte. Il s'agit dans ce cas d'un mécanisme descriptif, dynamique car ces contraintes correspondent en interne à des procédures pré-cablées gérées par MICRO, activées pour une instance, donc dans un environnement particulier. Ce type de contrainte est complémentaire aux descripteurs statiques de typage, ce qui enrichit considérablement la précision dans la caractérisation des propriétés représentées par les attributs.

Il est intéressant de remarquer qu'un filtre constitue en réalité une contrainte sur un ou plusieurs attributs *complexes*, qui permet l'accès aux attributs de l'objet référencé. Cette correspondance filtre/contrainte est étudiée dans [Gen95], qui propose de gérer les filtres de la même façon que les contraintes.

En particulier, les filtres, comme les contraintes, présentent un aspect descriptif important : dans la mesure où un filtre peut être perçu comme une restriction de l'extension de l'entité sur lequel il est posé, il affine le type principal de l'attribut auquel il est associé.

D'ailleurs, filtres et contraintes participent tous deux aux vérifications qui sont effectuées lors d'une instanciation : lorsqu'une valeur est déjà associée à un attribut auquel est associé un filtre (ou une contrainte), le modèle, après s'être assuré du respect des conditions imposées par les facettes de typage, vérifie que la valeur respecte aussi le filtre (ou la contrainte). Autrement dit, filtres et contraintes doivent être pris en compte dans le critère d'appartenance intensionnelle d'une instance à un concept ou à une classe, dans le cas où ils ont un effet sur le domaine des attributs considérés. Plus généralement, contraintes et filtres doivent faire partie de la définition intensionnelle des entités de représentation.

Ordre d'activation des inférences

Le descripteur de défaut mis à part (il ne doit être considéré qu'en l'absence de toute autre information), TROPES ne définit pas d'ordre d'importance pour l'activation de plusieurs inférences disponibles pour un même attribut. L'ordre d'activation correspond à l'ordre de définition des inférences. Cette décision, a priori arbitraire, provient de l'hypothèse faite quant aux résultats d'activations de plusieurs moyens d'inférence différents dans un contexte équivalent.

Les mécanismes d'inférence exploitent des connaissances disponibles dans la base, pour déterminer une valeur d'attribut, en cas d'absence de cette valeur. Cette valeur est pourtant unique pour une instance à un instant t, qu'elle soit connue du système ou non, elle est déterminée dans le monde modélisé. De ce fait, quel que soit le mécanisme utilisé pour déterminer cette valeur au sein du modèle, la valeur inférée à l'instant t doit être unique. En ce sens, l'ordre d'activation des inférences n'a pas d'intérêt puisqu'elles doivent toutes calculer la même valeur. Si tel n'est pas le cas, cela traduit une mauvaise représentation de la connaissance procédurale [Rec93].

2.3.3 Classification

La classification dans les systèmes terminologiques est un mécanisme qui permet de positionner un concept, individuel ou défini, dans une taxonomie de concepts ordonnée par la relation de subsomption.

Dans TROPES, qui distingue les classes des instances, un tel mécanisme existe pour classer les *instances*, c'est-à-dire déterminer, dans chaque point de vue, la classe la plus spécifique à laquelle l'instance peut être attachée. Il s'agit dans ce cas d'un processus d'identification incrémentale d'un objet. La classification de classes, qui correspond à la classification de concepts définis dans les langages terminologiques, consiste à déterminer la position d'une classe dans une hiérarchie, compte-tenu de sa description en termes d'attributs [Cap93]. Cette section est dédiée à l'étude de l'algorithme de classification d'instances de TROPES, qui est un mécanisme d'inférence, alors que la classification de classes sera étudiée au chapitre 5 en tant que processus d'aide à la construction des structures arborescentes d'une base de connaissances.

La classification est le processus d'identification adapté aux systèmes qui organisent hiérarchiquement les représentations. Le but de la classification d'une instance dans un modèle à objet est de déterminer les classes de rattachement les plus spécialisées pour l'instance. Il s'agit d'un enrichissement de la connaissance sur l'instance puisque plus une classe est basse dans la hiérarchie de spécialisation, plus elle est précise en ce qui concerne la caractérisation des individus qu'elle dénote (les conditions d'appartenance qu'elle impose sont pour cela de plus en plus restreintes).

La classification d'instances dans TROPES est sémantiquement comparable à la classification d'individuels dans les langages terminologiques. Toutefois, l'algorithme de classification d'instances dans TROPES est plus complexe dans la mesure où les entités de représentation y sont plus structurées que dans la plupart des systèmes terminologiques. De plus, contrairement à ces systèmes, TROPES maintient des hiérarchies de spécialisation, dont les propriétés sont largement utilisées pour guider et optimiser la descente d'une instance.

Olga Mariño a proposé pour TROPES un algorithme de classification d'instances (dont la procédure est notée `Classify-instance(`K, I`)` I étant l'instance à classer dans les arbres de spécialisation du concept K) qui exploite autant la coopération entre les points de vue représentée par les passerelles que la relation de spécialisation [Mar93]. Son principe est de faire descendre l'instance à classer le long des liens de spécialisation, dans chaque point de vue simultanément. Cette descente coordonnée entraîne une interaction permanente avec l'utilisateur afin que ce dernier fournisse les valeurs d'attributs qui ne sont pas disponibles, qui ne peuvent être inférées faute de moyens, et qui sont cependant requises pour décider de l'appartenance d'une instance à une classe. L'algorithme est itératif, chaque étape consistant à tester une classe dans un point de vue déterminé par avance. L'existence des passerelles permet de poursuivre la classification dans un point de vue lorsqu'elle est bloquée dans d'autres. Cette possibilité est d'autant plus puissante qu'elle permet de classer une instance dans un point de vue sans que toutes les valeurs d'attributs requises dans ce point de vue soient disponibles pour l'instance.

La classification d'instance est réalisée en attribuant aux classes des marques relatives à l'appartenance intensionnelle de l'instance à ces classes. TROPES reprend de SHIRKA [Rec89] les trois marques attribuables à une classe C lors de la classification d'une instance I :

- *sûre*, signifie que l'instanciation intensionnelle *stricte* de I à C est respectée,
- *possible*, signifie que l'instanciation intensionnelle *large* de I à C est respectée (il manque des informations permettant de statuer sur l'instanciation stricte, ce qui entraîne des possibilités de remise en cause de cette possibilité d'instanciation),
- *impossible*, signifie que l'instanciation intensionnelle large n'est pas vérifiée, une valeur d'attribut de I ne respectant pas les conditions posées sur le domaine associé à cet attribut dans C.

L'initialisation de l'algorithme de classification consiste à déterminer, dans chaque point de vue, une première classe d'appartenance de l'instance, appartenance vérifiée selon les critères intensionnels. Cette donnée peut être fournie en paramètre de la procédure `Classify-instance`: à la place du concept dans lequel se déroule la classification, le paramètre indique alors, pour chaque point de vue, la classe de départ de la classification. Par défaut, la classe de départ de l'instance dans un point de vue est la racine de ce point de vue. L'algorithme est alors une itération sur cinq étapes séquentielles. On note I l'objet à classer, et $C|PV$ la classe sûre la plus spécialisée pour I dans le point de vue PV, connue par l'algorithme.

Obtention d'information Le système interroge l'utilisateur quant à une information manquante sur I, à savoir les valeurs de I associées aux attributs décrivant $C|PV$.

Appariemment Une fois l'information sur I complétée, l'algorithme cherche à descendre l'instance dans le point de vue PV. Pour cela, l'appariemment de l'instance avec chacune des sous-classes de $C|PV$ est testé, c'est-à-dire que l'algorithme teste le respect des conditions sur les attributs par les valeurs connues de l'instance. Durant cette phase, l'utilisateur se voit demander les valeurs des attributs ajoutés dans les sous-classes. Le résultat de la phase d'appariemment est soit une classe sûre (toutes les autres étant alors impossibles du fait de l'hypothèse d'exclusivité de la hiérarchie de spécialisation), soit un ensemble de classes possibles et impossibles. Il est à noter que la propriété d'exhaustivité permet aussi de déterminer une classe sûre sans la tester : si toutes ses classes sœurs sont impossibles, il ne reste qu'elle.

Le principe algorithmique qui régit cette phase d'appariemment est de limiter le nombre de classes à tester et d'ordonner les questions posées à l'utilisateur en fonctions des moyens d'inférence disponibles.

Propagation de marques La procédure d'appariemment étant achevée, l'algorithme propage les marques qui traduisent les déductions qui peuvent être faites compte-tenu du résultat. Nous précisons quelques règles de propagation, la totalité de ces règles étant disponible dans [Mar93].

- les sous-classes des classes impossibles sont elles aussi marquées impossibles,
- si une classe impossible est la destination d'une passerelle de source unique, cette source est elle aussi marquée impossible,
- si une classe sûre est la source unique d'une passerelle, la classe destination est elle aussi sûre, ses classes sœurs devenant impossibles,
- lorsqu'une classe est sûre, toutes ces super-classes deviennent transitivement sûres.

L'algorithme réalise toutes ces propagations dans tous les points de vue, jusqu'à ce qu'aucune marque ne puisse être effectuée. Les points de vue et passerelles "inactifs" sont alors détectés, ils ne seront plus consultés lors de la prochaine itération.

Mise à jour de l'information L'étape précédente a permis de mettre en évidence une ou plusieurs classes sûres pour l'instance. Ces classes possèdent vraisemblablement des attributs dont les valeurs sont inconnues pour l'instance (notamment si l'une des classes appartient à un autre point de vue que celui considéré aux étapes précédentes). La mise à jour de l'information consiste à inférer ces valeurs lorsque cela est possible.

Choix du prochain point de vue Cette dernière étape consiste à déterminer le point de vue dans lequel la classification va se poursuivre lors de la prochaine itération. Cette sélection mêle des considérations telles que l'importance relative accordée aux points de vue (définie au préalable par l'utilisateur), ou la quantité d'informations disponible dans chacun. Le choix du prochain point de vue fixe la prochaine classe sûre de laquelle partira le processus d'identification. La boucle est bouclée.

L'itération constituant la classification se termine lorsque plus aucune information n'est disponible, ou bien évidemment, et c'est l'idéal, lorsque la classe d'appartenance la plus spécialisée (une feuille) a été déterminée pour I dans chaque point de vue.

Il est très important de voir que la classification d'une instance n'aboutit pas nécessairement au rattachement de l'instance aux classes déterminées. En fait, cette classification est la recherche d'une position de l'instance qui soit intensionnellement correcte, c'est-à-dire qui respecte l'instanciation intensionnelle stricte ou au moins large. Le rattachement effectif de l'instance aux classes trouvées relève de la validation extensionnelle d'une telle instanciation, confiée à l'utilisateur.

La correction et la terminaison de cet algorithme ont été prouvées par Olga Mariño, Alejandro Quintero El Guapo Caballero et Rodrigo Cardoso, sur la base des sémantiques intensionnelle et extensionnelle des relations d'instanciation de spécialisation et celle induite par la pose de passerelles, ainsi qu'à partir des propriétés d'exclusivité et d'exhaustivité des taxonomies des points de vue [Mar93] [Qui93].

Cet algorithme s'applique de la même façon en présence d'attributs élémentaires qu'en présence d'attributs complexes, en considérant interdites les définitions récursives (cycliques) [MRU90].

2.4 Objets de représentation et types sous-jacents

Les sections précédentes ont montré la sémantique mixte des entités de représentation, interprétables intensionnellement et extensionnellement, et en conséquence la double interprétation des relations entre ces entités. Cette section a pour but de montrer que la manipulation des interprétations intensionnelles des entités de représentation peut être considérée indépendamment de leur sémantique extensionnelle, au moyen de procédés typiquement “informatiques”, ou plus précisément calculatoires.

2.4.1 Interprétation intensionnelle et typage : vers une équivalence

Cette section a pour objectif de montrer pour quelles raisons l'intension des entités de représentation peut être exprimée sous forme de types.

Intension vs. extension

Le concept d'intension n'existe que par la nécessité de distinguer formellement *ce qu'un concept dénote dans le monde modélisé*, explicitement représenté dans la base de connaissances (extension) et *les propriétés cherchant à caractériser cette dénotation* (intension). Ainsi en représentation des connaissances par objets, l'intension est directement liée à la structure des classes et concepts, c'est-à-dire à leur décomposition en termes d'attributs (les propriétés).

Dans une base de connaissances, l'extension est assertionnelle : une instance est dite appartenir à un concept (indépendamment de la structure des objets informatiques représentant l'instance et le concept) car la dénotation de l'instance dans l'univers modélisé appartient à l'ensemble dénoté par le concept.

Inversement, l'intension d'un concept est calculable à partir de la décomposition de ce concept en attributs, c'est-à-dire que l'on peut décider quels sont, parmi les éléments de l'univers dénoté par toutes les valeurs possibles des attributs de ce concept, ceux qui possèdent les propriétés requises pour appartenir au concept. Tous ces éléments ne font pourtant pas tous partie de l'extension de ce concept : seuls ceux qui ont une correspondance dans le monde modélisé sont de réelles instances du concept.

La différence entre intension et extension est alors plus nette lorsque l'on remarque que les co-domaines des fonctions d'interprétation intensionnelle et extensionnelle ne sont pas les mêmes, le premier incluant le second. En conclusion, disons que l'intension est à l'extension ce que la potentialité est à la réalité. Cette affirmation doit être considérée en toute relativité. Elle est en fait une conséquence de notre incapacité à identifier les propriétés *définitionnelles* de ce que l'on veut représenter : le choix des propriétés définissant l'intension d'un concept, et leur spécification, dénotent, ensemble, plus d'éléments qu'il n'y a d'éléments réels. Idéalement, intension et extension devraient être équivalentes.

De nombreuses tentatives ont été faites pour la mise en œuvre d'algorithmes fondés sur la *subsomption extensionnelle* [BS85]. La subsomption extensionnelle, basée sur la théorie des modèles, s'est avérée non-efficace du fait de l'inefficacité de la déduction en logique du premier ordre. Les premières solutions trouvées ont été de restreindre cette logique, mais se sont révélées irrecevables car elles menaient à des systèmes trop peu expressifs. La plupart des langages terminologiques se sont alors orientés vers la *subsomption intensionnelle*, c'est-à-dire calculable sur la base de la structuration des concepts en termes de propriétés (d'attributs). L'accent est alors mis sur la relation entre subsomption intensionnelle et subsomption extensionnelle : à l'issue du test de subsomption sur les structures des concepts, chaque instance du concept subsumé doit être une instance du subsumant. Autrement dit, la subsomption intensionnelle doit être correcte vis-à-vis de la subsomption extensionnelle[11].

Cette correspondance essentielle entre intension et extension est ainsi à la base de nombreux algorithmes de systèmes de représentation des connaissances par objets, tels que la classification (d'instances ou de classes), le filtrage ou la détection d'incohérences. Ces algorithmes sont actuellement fondés sur les intensions des entités de représentation, c'est-à-dire sur leur structure d'agrégation et l'interprétation de ces structures dans *tous les mondes possibles.* Ils doivent toutefois garantir l'extension de la base de connaissances, à savoir l'interprétation de ses concepts dans *le monde modélisé.* Ce problème est simplifié dans la plupart des langages terminologiques, car ils supposent l'équivalence entre intensions et extensions des concepts définis.

Notre but est de représenter, indépendamment de l'extension d'une base de connaissances, le co-domaine de la fonction d'interprétation intensionnelle, afin de favoriser la mise en évidence d'une application surjective de correspondance entre ce domaine et le co-domaine de la fonction d'interprétation extensionnelle. Cette correspondance est dépendante des termes du langage de représentation qui décrivent la base de connaissances (figure 2.14). La nécessité de cette distinction vient de l'utilisation même des intensions et des extensions : les opérations sur les intensions sont des manipulations des structures des entités de représentation, alors que les opérations sur les extensions décrivent le comportement des ensembles d'objets dénotés dans le monde modélisé. Ce sont donc deux exploitations différentes des entités de représentation que nous cherchons à mettre en évidence, en prétendant que les opérations sur les intensions peuvent être décrites indépendamment du contenu sémantique des entités de représentation.

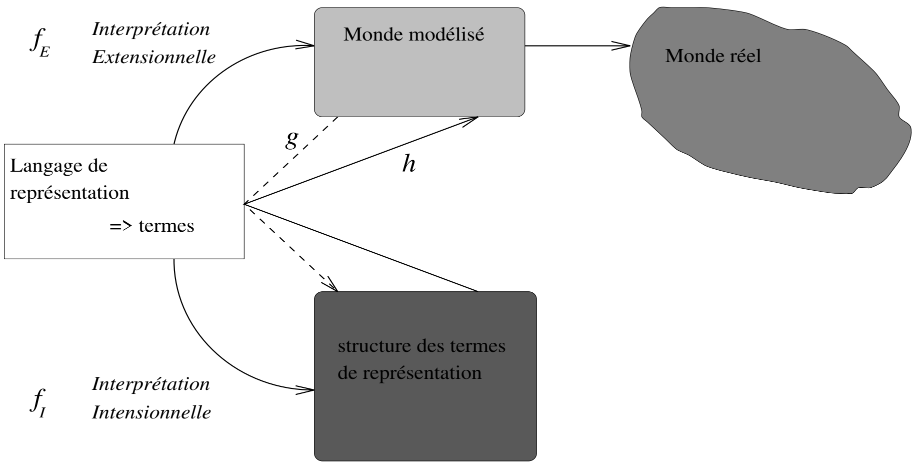


FIG. 2.14 - : Les systèmes de représentation des connaissances admettent généralement que le co-domaine de la fonction d'interprétation intensionnelle ($\mathcal{D}_I$) contient le co-domaine de la fonction d'interprétation extensionnelle ($\mathcal{D}_E$). Pourtant, les opérations effectuées sur les éléments de chacun de ces domaines sont de natures très différentes : les opérations sur des éléments de $\mathcal{D}_I$ sont essentiellement calculatoires (manipulation de structures de données), alors que les opérations (s'il en est) sur les éléments de $\mathcal{D}_E$ relèvent du comportement de ces objets vis-à-vis de l'univers modélisé. Par exemple, les opérations `miauler` et `imprimer-chat`, relatives toutes deux à la classe des `chats`, ne sont pas de même nature car l'une ne se base que sur la structure de la classe alors que l'autre dénote un comportement des chats indépendant de la représentation qui en est faite dans le modèle.

[11] Idéalement, la réciproque devrait pouvoir être établie, c'est-à-dire que la subsomption intensionnelle devrait entraîner la subsomption extensionnelle.

Nous proposons de distinguer explicitement les domaines d'interprétation $\mathcal{D}_I$ et $\mathcal{D}_E$, afin d'illustrer ces deux niveaux d'exploitation (intensionnel et extensionnel) des termes de représentation. Ces deux domaines peuvent être reliés de par la sémantique du système de représentation, en considérant les inverses des deux fonctions d'interprétation f_I et f_E:

$$g = f_I \circ f_E^{-1} \text{ et } h = g^{-1} = f_E \circ f_I^{-1}$$

Types vs. classes, valeurs vs. objets

Les notions de type et de classe sont souvent confondues car elles présentent de nombreuses similitudes. Aucun de ces deux concepts n'a de définition propre, les deux permettent de *grouper des éléments*. Le but de cette section est d'analyser les différences entre ces deux notions dans le cadre de la représentation des connaissances par objets.

Nous proposons ci-dessous une définition ciblée de chacune de ces deux notions :

- Une classe de représentation permet de regrouper des dénotations d'éléments dont l'existence est complètement justifiée par les objets réels qu'ils approximent.
- Un type permet de regrouper tous les éléments, pris dans un univers de valeurs [Sco76], qui satisfont le critère d'appartenance de ce type. Ainsi, l'existence d'un élément d'un certain type n'est pas dépendante d'une modélisation particulière.

Plus précisément, l'ensemble représenté par le type est complètement caractérisé par la définition même de ce type, alors que dans l'absolu, l'ensemble des éléments que regroupe une classe ne dépend pas totalement de la description de cette classe mais aussi de l'existence d'une correspondance entre un élément potentiel de la classe et une approximation de l'univers modélisé.

La distinction que l'on cherche à établir, dans le cadre de la représentation des connaissances par objets, entre classe et type, est dépendante des différences existant entre les notions d'*objet* (élément d'une classe) et de *valeur* (élément d'un type), que nous nous proposons d'examiner rapidement. Les différences fondamentales entre objet et valeur sont issues des notions d'état et surtout d'*identité*, pertinentes pour un objet et inconnues des valeurs [Mac82]. Pour appuyer ceci, un objet est parfois comparé à un pointeur vers une valeur : l'adressage d'une valeur par un pointeur est une fonction totale, de même que la dénotation d'un élément du monde modélisé par un objet de représentation. Ici, l'adressage d'une valeur donne un contexte à la valeur, elle possède une signification dans ce contexte ; c'est pourquoi plusieurs significations peuvent être associées à une même valeur, selon le contexte dans lequel elle est adressée. À l'opposé, un objet de représentation a toujours la même signification quel que soit son contexte d'utilisation, il dénote toujours une approximation du monde modélisé.

En conséquence, la manipulation d'un objet, par le système de représentation, revient à la manipulation de sa signification. Les opérations effectuées sur les valeurs associées à un objet sont indépendantes de la dénotation de cet objet, jusqu'au moment où le résultat de ces opérations sert à modifier l'état de l'objet. À ce stade, les valeurs en question n'interviennent pas dans la phase de modification de l'objet, c'est-à-dire dans la phase de modification de la représentation du monde modélisé ; elles n'ont servi qu'à établir le résultat.

Distinguer valeur et objet, en représentation des connaissances, devient une réelle nécessité, à partir du moment où l'on considère les *opérations* dont ils sont les acteurs. Définir, au niveau du modèle de connaissances, une opération sur un objet admettant une dénotation dans le monde modélisé, signifie représenter un comportement de la dénotation de cet objet. À l'opposé, une opération sur des valeurs n'a pas de signification *dans l'absolu*, elle est le reflet des propriétés de l'ensemble des valeurs auxquelles elle peut s'appliquer. Et ce sont justement la manipulation de ces propriétés, les résultats qu'on peut exploiter à partir de ces propriétés, qui rendent les valeurs intéressantes pour représenter (sans forcément quantifier) des propriétés entre objets sans que l'on puisse représenter exactement ces objets. Par exemple, la notion d'`âge` comporte beaucoup de significations, en particulier cette notion peut être plongée dans un ensemble totalement ordonné, d'où le naturel que l'on trouve à représenter l'âge d'une personne par un nombre entier. On pourra ainsi utiliser l'ordre sur les entiers pour déterminer la plus jeune d'entre deux personnes, mais ce n'est pas pour autant que l'addition de deux entiers correspond toujours à une addition d'âges de personne : c'est le fait de lier ces entiers à des personnes qui donne une signification particulière, dans le monde modélisé, à leur addition (l'âge étant en fait un nombre d'années).

La figure 2.15 résume la raison pour laquelle nous avons besoin de distinguer les notions de valeur et d'objet, car les deux collaborent, à des degrés d'abstraction différents, pour la représentation en machine d'une approximation du monde modélisé : l'objet se "décrit" à l'aide de valeurs ou d'autres objets, mais l'interprétation des valeurs considérées dans les objets est guidée par la signification de ces objets dans le monde modélisé.

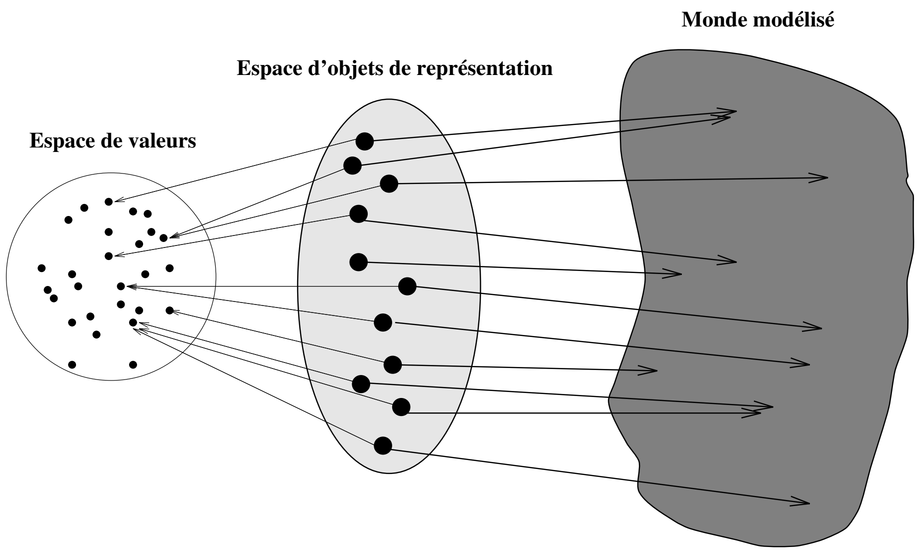


FIG. 2.15 - : La différence entre les notions d'objet et de valeur, dans le cadre de la représentation des connaissances par objets, est que l'objet a un contenu sémantique propre, donné par sa dénotation dans le monde modélisé. La valeur est une primitive pour décrire des objets, sa signification est multiple, elle dépend des objets qui l'utilisent.

La différence que nous avons illustrée entre valeur et objet, fondée sur les notions d'état et d'identité, peut immédiatement se généraliser aux ensembles qui regroupent ces éléments : la classe regroupe un ensemble d'objets, le type regroupe un ensemble de valeurs. En ce sens, la classe a une dénotation dans le monde modélisé, alors que le type n'en a pas : le type permet de regrouper des valeurs (ou des données) qui ont en commun, généralement, un comportement, c'est-à-dire que les mêmes opérations caractérisées peuvent être appliquées aux valeurs d'un même type.

Type : expression formelle de l'intension

L'intension d'une classe (resp. d'un concept) est la description d'un ensemble d'éléments d'un univers, c'est-à-dire d'un ensemble d'instances potentielles pour cette classe (resp. ce concept), indépendamment du fait que ces instances puissent avoir ou non une dénotation dans le monde modélisé, mais plutôt compte-tenu de la décomposition en attributs (propriétés) de la classe (resp. du concept). Le domaine de l'intension peut alors être ramené à un ensemble de *valeurs*, à condition d'établir la correspondance entre une valeur et l'objet ayant cette valeur. Ceci est motivé par le fait que la plupart des processus d'un système de représentation manipulent en partie l'interprétation intensionnelle des objets, pour effectuer des calculs indépendants de l'interprétation extensionnelle. La correspondance inverse est importante justement parce qu'à l'issue de ces calculs, le système doit interpréter les résultats obtenus (donc sous forme de valeurs) en considérant ce que les objets dénotent dans le monde modélisé.

Nous proposons de représenter ces ensembles de valeurs, c'est-à-dire les intensions des entités de représentation, par des types, qui définiront eux-mêmes les opérations applicables sur leurs valeurs. Cette représentation permet de distinguer, au sein du système de représentation des connaissances, les opérations liées essentiellement aux intensions des entités, indépendamment de leur signification dans le monde modélisé, des opérations et processus manipulant l'extension de ces entités, indépendamment de leur structure.

Cette distinction effective entre intension et extension, par la représentation explicite des deux, doit cependant respecter le fait que ces deux vues d'un objet de représentation ne sont pas orthogonales, l'intension devant respecter l'extension. En ce sens, le typage d'une base de connaissances, et les opérations mises en œuvre sur les types obtenus, devront garantir la correction de l'interprétation extensionnelle de la base de connaissances.

2.4.2 Un support pour la création de nouvelles structures de données

La création d'une base de connaissances nécessite la référence à des types dits "de base" (`Entier`, `Réel`, `Chaîne`), ainsi qu'à des constructeurs de types simples (`Liste`, `Ensemble`, `Séquence`), mais aussi parfois à des types et constructeurs plus complexes, tels que `Date` ou `Matrice`. Généralement, les types effectivement référencés sont ceux définis par le langage hôte et sont interprétés par le système de représentation des connaissances. Pourtant, ces types de données ne sont pas toujours fournis par le langage hôte, et les concepteurs de bases de connaissances utilisent alors le langage de représentation des connaissances pour les définir. C'est ainsi que l'on constate, au sein d'une même base de connaissances, la représentation conjointe de concepts tels que `Protéine` et `Matrice`. Autrement dit, on représente à un même niveau d'interprétation des concepts alors qu'on ne les représente pas pour les mêmes raisons. En effet, si le concept `Protéine` admet une interprétation extensionnelle unique (le but de ce concept est de regrouper toutes les protéines existantes, chacune ayant son identité propre), le concept `Matrice` n'a pas d'extension dans le *même* monde modélisé, ses instances ne représentent pas un fait unique dans le sens où une matrice est la représentation mathématique d'une multitude de phénomènes sémantiquement disjoints.

Cette possibilité de permettre la modélisation de concepts sans dénotation dans le monde modélisé, et ce en termes du langage de représentation, présente deux inconvénients majeurs. D'une part, le langage de représentation a tendance à évoluer pour faciliter de telles définitions, ce qui revient à modifier le rôle initial de ce langage. Nous reviendrons d'ailleurs sur ce point dans le dernier chapitre de ce mémoire.

D'autre part, nous avons vu que la sémantique du système de représentation des connaissances par objets est dénotationnelle, dans le sens où les entités décrites sont toujours supposées avoir une correspondance dans le monde modélisé. Or à partir du moment où la représentation de types de données est possible en termes du langage de représentation, cette sémantique n'est plus pertinente, puisque certains objets, pour lesquels une identité est reconnue, n'ont justement pas de dénotation dans l'univers modélisé : ils n'existent que par les relations qui les lient aux autres objets (de la même façon qu'une valeur acquiert une signification vis-à-vis de l'application à partir du moment où elle est référencée par la description d'un objet, comme nous l'avons vu dans la section précédente).

Une solution évidente à ce problème est d'expliciter la différence de statut entre ces objets, vis-à-vis de la dénotation. Il existe pour cela deux possibilités.

1. *Préciser le statut de chaque objet (ou classe d'objets) dans la description même de l'objet.* La sémantique du modèle doit alors prendre en considération l'ensemble des statuts existants. Cette solution a l'inconvénient de rendre compliquée la sémantique du système, si l'on imagine que chaque objet peut avoir son propre statut vis-à-vis de chaque opération possible du système de représentation.
2. *Développer, parallèlement au langage de représentation, un langage pour l'expression de ce type de données et pour la représentation de ses valeurs.* Les termes d'un tel langage n'ont ainsi pas à être interprétés par la sémantique dénotationnelle qui définit le modèle de connaissances.

Chacune des deux solutions ci-dessus offre la possibilité au concepteur de préciser le statut d'un objet ou d'un ensemble d'objets, ce qui présente l'avantage d'obliger le concepteur à s'interroger sur ce qui est intéressant dans cet objet : sa dénotation unique ou seulement sa description. Toutefois, la seconde solution proposée est plus séduisante que la première, et ceci pour deux raisons majeures :

- un langage pour l'expression de types de données sera obligatoirement adapté à ce type particulier de définitions. En effet, l'ensemble des *dates* est totalement ordonné, il existe des opérations sur ses éléments qu'il est intéressant de vouloir spécifier. C'est d'ailleurs ce qui rend l'identification de ce type intéressante, car sinon pourquoi ne pas représenter une date comme une chaîne de caractères ?
- le fait que le système n'ait pas à interpréter les termes du langage de définition de types de données rend bien entendu sa sémantique plus simple car toujours *totalement* fondée sur les extensions[12].

Pour ces deux raisons, nous avons opté pour la seconde solution, qui revient à définir un support pour la définition de types de données. La définition d'un tel type revient à identifier ses valeurs et à caractériser les opérations sur ces valeurs. L'implémentation d'un type doit être indépendante du système de représentation des connaissances, mais ses opérations et valeurs doivent pouvoir y être utilisées. En conséquence, le système de représentation doit avoir accès à la spécification des types.

[12] Lorsque le statut de l'objet, vis-à-vis de son interprétation extensionnelle, est fourni par l'objet même, la fonction d'interprétation des objets de représentation dans le monde modélisé n'est plus qu'une fonction partielle.

Chapitre 3

Le système de types Metéo

L'élaboration d'un système de types dédié à un langage n'est pas une originalité en soi. L'objectif de ces systèmes est, de façon générale, d'apporter l'abstraction, la possibilité de construire de nouvelles structures, et la protection. La particularité du système de types MÉTÉO (**M**odule **E**xtensible de **T**ypes **É**laborés pour les **O**bjets) est qu'il est dédié à un modèle de connaissances par objets. En ce sens, les objectifs de METÉO se distinguent sensiblement de ceux habituellement mis en avant (section 3.1), par conséquent son architecture, même si elle se fonde sur les mêmes résultats théoriques (section 3.2), diffère de celles usuellement rencontrées.

Les sections 3.3 et 3.4 s'attachent à exposer techniquement les deux niveaux de types considérés dans METÉO tels qu'ils ont été introduits dans la section 3.1. Après avoir spécifié l'architecture générale de METÉO (section 3.5), nous illustrons l'extensibilité de METÉO, à savoir la possibilité qu'il offre d'ajouter de nouvelles structures de données 3.6. L'exposé de METÉO qui est donné dans ce chapitre ne reflète pourtant pas son intégrité, car il n'y est pas encore tenu compte de ses objectifs, propres à la représentation des connaissances.

3.1 Objectifs de Metéo

Le chapitre précédent a montré à quels niveaux se situe la notion de *type de données* dans un modèle de connaissances à objets. En résumé, des structures de données, et leurs opérations, sont implicitement contenues dans les termes du langage de représentation, ces termes dénotant, ou non, une partie du monde modélisé. Dans le premier cas, il s'agit de données issues d'une base de connaissances particulière (ex. le type *record* issu de la classe `protéine`, qui en représente son intension). Dans le second cas, il s'agit d'une mauvaise utilisation du langage de représentation, dont le rôle initial est de ne permettre que la représentation du monde modélisé.

Dans les deux cas, il est certain que ce n'est pas au modèle de connaissances de définir ces types de données, car même si certaines d'entre elles sont issues des connaissances représentées, les opérations sur des données ayant mêmes structures peuvent se définir indépendamment de toute application.

Nous proposons alors l'élaboration d'un système de types qui représente ces deux niveaux de types (structures) de données. Les deux composantes essentielles du système de types que nous envisageons sont résumées ci-dessous.

– Il concerne la représentation explicite des structures de données présentes à différents niveaux dans le modèle de connaissances. Il s'agit, d'une part, de structures dont les données peuvent résulter de l'interprétation intensionnelle des entités de représentation (ex. la structure d'agrégation, de *record*), et d'autre part de structures ne correspondant à aucun objet de représentation, mais qui se révèlent être des supports adaptés à la mise en œuvre efficace de processus calculatoires (ex. la structure matricielle qui modélise très bien, par exemple, un mouvement de rotation).

– Outre la définition de ces structures, le système de types doit comporter (implémenter) toutes les opérations et propriétés de leurs valeurs. En particulier, il doit comporter toutes les opérations de manipulation des intensions, qui habituellement sont définies par le système de représentation.

Le système de types ainsi conçu est faiblement couplé au système de représentation des connaissances, par caractérisations de la fonction totale d'interprétation intensionnelle et de sa réciproque. Toutefois, le couplage entre le système de types et le système de représentation est plus riche, car s'il doit y avoir interprétation des éléments de représentation vers l'espace de valeurs, il doit aussi y avoir interprétation des opérations sur ces éléments, avec conservation de leurs propriétés. D'un point de vue algébrique, il s'agit d'élaborer un morphisme de l'algèbre des termes de représentation vers l'algèbre des valeurs.

Principe de l'organisation de Metéo

METÉO est un système de types auquel on associe une sémantique dénotationnelle. Les types définis dans METÉO sont interprétés par des sous-ensembles caractérisés d'un univers de valeurs. Cette interprétation des types permet une définition du sous-typage fondée sur l'inclusion des ensembles.

METÉO considère, dans l'univers de valeurs, deux niveaux imbriqués de types :

– Un C-type est une structure de données munie d'un ensemble d'opérations et de propriétés sur ces données.

– Un δ-type, issu d'un C-type, représente un sous-ensemble de l'ensemble défini par le C-type. Aux valeurs d'un δ-type sont applicables les opérations définies par le C-type.

Ces deux niveaux de typage sont motivés par la nécessité de distinguer, d'une part des structures de données dont la définition est indépendante du modèle de représentation (les C-types), et d'autre part les types issus des entités de représentation d'une base de connaissances particulière (les δ-types). Autrement dit, les δ-types seront issus du typage des entités de représentation de connaissances, alors que les C-types définissent la structure de représentation des δ-types, et les opérations sur ces structures.

Afin de caractériser l'espace de valeurs sur lequel doit être défini le système de types METÉO, et pour donner un aperçu des propriétés formelles de ce système vis-à-vis des propriétés classiques, la section suivante est réservée à l'exposé des principales caractéristiques des systèmes de types rencontrés en programmation.

3.2 La notion de type en programmation

"Types arise informally in any domain to categorize objects according to their usage and behavior. The classifica-

tion of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system." Luca Cardelli, Peter Wegner [CW85][p.473].

Avant de décrire MeTéo, nous proposons un bref aperçu de la notion de *type* telle qu'elle est définie et manipulée dans les langages de programmation. Après avoir énoncé les critères de différentiation des systèmes de types actuels, le modèle de treillis des idéaux est exposé comme le fondement d'une sémantique dénotationnelle des types. Les notions importantes de polymorphisme et de types abstraits de données seront ensuite abordées dans les sections 3.2.3 et 3.2.4.

3.2.1 Caractéristiques des systèmes de types

Une classification possible des langages de programmation pourrait être basée sur la présence ou l'absence d'un système de types dédié à chacun, puis plus précisement sur les caractéristiques de ces systèmes de types.

Apports des systèmes de types

Les systèmes de types sont généralement associés à un langage pour plusieurs raisons :

- *L'abstraction de données* : l'abstraction permet de travailler avec des données complexes, sans qu'il soit nécessaire de connaître leur représentation en machine ni leur implémentation.
- *La sécurité* : la fiabilité des programmes, c'est-à-dire l'assurance d'une bonne application, peut être en partie assurée grâce au typage des valeurs et des fonctions/procédures, qui évite que des opérations soient appliquées à des éléments sans que cela ait une signification. Le typage assure la cohérence des programmes.
- *La construction de nouvelles structures* : un des intérêts des systèmes de types est de permettre la construction de nouveaux types de données à partir de types prédéfinis, comme des types composés, *records* ou vecteurs.

Sécurité des programmes

En ce qui concerne le degré de protection, trois catégories de langages se distinguent :

- *les langages non typés* : n'assurent aucune vérification (ces langages sont ceux de très bas niveau, tels que les assembleurs),
- *les langages faiblement typés* : assurent la sécurité en ne détectant qu'une classe restreinte d'erreurs de types (qu'un sous-ensemble d'incohérences possibles). C peut être considéré comme un langage faiblement typé.
- *les langages fortement typés* : assurent la sécurité en détectant toute erreur possible de type. Parmi les langages fortement typés, on relève bien entendu Pascal. Ces langages fortement typés permettent une relative fiabilité des systèmes, incitent une programmation disciplinée, mais leur rigidité n'est pas toujours adaptée à certaines catégories de problèmes.

Et parmi les deux dernières catégories, on distingue encore les systèmes de types *statiques* (les vérifications sont effectuées à la compilation) et les systèmes *dynamiques* (vérifications à l'exécution).

Flexibilité des systèmes

En ce qui concerne la flexibilité des systèmes de types, deux autres catégories apparaissent, qui traduisent à la fois la puissance d'expression et d'abstraction de ces systèmes de types :

- *Les systèmes monomorphes* : une valeur (et donc une fonction) n'appartient qu'a un seul type ; le langage PASCAL est un langage monomorphe.
- *Les systèmes polymorphes*: une valeur (donc une fonction) peut prendre plusieurs types. Les langages polymorphes les plus célèbres sont certainement ML [Mil84], Smalltalk [GR83], et Flavors [WM81], ces deux derniers n'étant pourtant pas typés.

Inférences de types

Une dernière grande caractéristique des systèmes de types indique qui, du compilateur ou du programmeur, donne l'information sur les types des programmes. On distingue ainsi :

- *les systèmes à typage implicite*, où le compilateur infère les types des programmes (fonctions et valeurs), c'est le cas du langage fonctionnel ML qui autorise toutefois le typage d'expressions par l'utilisateur,
- *les systèmes à typage explicite*, où le programmeur déclare les types de ses programmes, tels que les systèmes de types de PASCAL [WSH77] ou Algol68, ce dernier autorisant malgré tout la cœrcion.

3.2.2 Une théorie des types : le modèle de treillis des idéaux

Il existe plusieurs théories formelles des types, qui peuvent se regrouper en deux catégories :

- *Les théories du premier ordre*, basées sur l'algèbre universelle, qui se fondent sur une distinction très nette entre valeurs et fonctions.
- *Les théories d'ordre supérieur*, basées sur le λ-calcul, qui traitent à un même niveau les valeurs et les fonctions.

Dans le cadre de la seconde classe des théories formelles, Dana Scott a été le premier à décrire un modèle des λ-termes, en introduisant le *domaine récursif de Scott* [Sco76]. Il s'agit de la caractérisation d'un univers des valeurs $\mathcal{V}$, qui a la propriété d'être clos par les opérations de produit, de somme et d'application fonctionnelle, c'est-à-dire que le domaine récursif de Scott $\mathcal{V}$ vérifie l'équation $\mathcal{N} + \mathcal{T} + (\mathcal{V} \times \mathcal{V}) + (\mathcal{V} + \mathcal{V}) + (\mathcal{V} \rightarrow \mathcal{V}) \subseteq \mathcal{V}$, où $\mathcal{N}$ est l'ensemble des entiers naturels et $\mathcal{T}$ est l'ensemble des valeurs de vérités. En conséquence, le domaine récursif de Scott peut être considéré comme *l'ensemble de toutes les valeurs et fonctions calculables*. Ce modèle est intéressant car il reflète l'idée intuitive qu'un type est un ensemble de valeurs, et permet alors de donner une sémantique dénotationnelle aux types.

Le modèle du domaine récursif a été ensuite muni d'un ordre partiel complet par David MacQueen et Ravi Sethi [MS82], et ce pour introduire la notion d'inclusion ensembliste sur des types, donnant lieu au *modèle de treillis des idéaux*. Dans cette sémantique, *un type est interprété comme un idéal (voir définition 13) dans le domaine récursif*. Autrement dit, un type dénote un ensemble de valeurs de $\mathcal{V}$, cet ensemble possédant des propriétés particulières (les propriétés des idéaux), de façon à ne caractériser que des types dits *légaux*, c'est-à-dire des types qui ne regroupent pas des valeurs selon aucun critère [CW85]. L'ensemble des idéaux est ensuite organisé par un ordre partiel complet.

Définition 13 (Idéal d'un ensemble ordonné) *Un* Idéal *I d'un ensemble D, muni d'un ordre partiel noté $\sqsubseteq$, est un sous-ensemble de D respectant les propriétés suivantes :*

1. *$I \neq \emptyset$,*
2. *$\forall x \in I$, $\forall y \in D$ tel que $y \sqsubseteq x$, alors $y \in I$*
3. *$\forall n$, Pour toute suite croissante $x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n$ dans I, le plus petit majorant $x = inf\{x_1, x_2, \cdots, x_n\}$ existe et $x \in I$*

Les idéaux sur $\mathcal{V}$ ne sont pas disjoints. L'ensemble des idéaux issus d'un domaine D, noté $\mathcal{I}_D$, peut être muni de l'ordre partiel correspondant à l'inclusion ensembliste. $\mathcal{I}_D$ est alors un treillis, dont l'élément inférieur ($\perp_{\mathcal{I}}$) est l'idéal-singleton ne contenant que la valeur minimum de $\mathcal{D}$ (qui existe), et dont l'élément supérieur ($\top_{\mathcal{I}}$) est $\mathcal{D}$ lui-même.

En ce sens, et d'après la définition 13, un idéal I_1 peut être sous-ensemble d'un autre idéal I_2, avec comme contrainte qu'un élément de I_2 qui précède par $\sqsubseteq$ l'élément maximum de I_1 appartient obligatoirement à I_1.

Les types existants dans les langages de programmation représentent en fait un petit sous-ensemble de l'ensemble des idéaux sur $\mathcal{V}$. Par exemple, n'importe quel sous-ensemble des entiers naturels ayant un maximum est un idéal, et peut ainsi être considéré comme un type, ce qui montre l'infinité des types que l'on peut caractériser par ce modèle.

Sémantique dénotationnelle des types en programmation

Le modèle de treillis des idéaux permet de définir une sémantique dénotationnelle des types. L'intérêt d'une telle sémantique construite sur ce modèle est que l'appartenance d'un élément à un type est interprétée naturellement comme l'appartenance d'une valeur à un ensemble.

Soit $\mathcal{T}$ l'ensemble des types d'un langage, et $\mathcal{E}$ l'ensemble des valeurs de ces types, tel que $\forall e \in \mathcal{E}, \exists t \in \mathcal{T}$ tel que $e : t$, où $e : t$ dénote l'appartenance d'un élément à un type. L'ensemble des idéaux sur $\mathcal{V}$ est noté $\mathcal{I}_{\mathcal{V}}$. Nous définissons la fonction d'interprétation ξ telle que

$$||.||^{\xi} : \mathcal{E} \rightarrow \mathcal{V}$$

$$||.||^{\xi} : \mathcal{T} \rightarrow \mathcal{I}_{\mathcal{V}}$$

Définition 14 (Appartenance d'un élément à un type) *L'appartenance d'un élément $e \in \mathcal{E}$ à un type $t \in \mathcal{T}$ est définie comme suit :*

$$\forall e \in \mathcal{E}, \exists t \in \mathcal{T}, e : t \Leftrightarrow ||e||^{\xi} \in ||t||^{\xi}$$

À partir du moment où l'appartenance d'un élément à un type est interprétée par ξ, la relation de sous-typage peut-être interprétée comme l'inclusion ensembliste :

Définition 15 (Sous-typage) *Le lien de sous-typage "t_1 est un sous-type de t_2" est noté $t_1 \leq_\tau t_2$.*

$$\forall t_1, t_2 \in \mathcal{T}, t_1 \leq_\tau t_2 \Leftrightarrow \|t_1\|^\xi \subseteq \|t_2\|^\xi$$

La relation de sous-typage est alors complètement définie dans la mesure où l'ensemble $\mathcal{I}_\mathcal{V}$ des idéaux sur $\mathcal{V}$, muni de l'inclusion ensembliste, forme un treillis [1].

Le modèle de treillis des idéaux n'est pas le seul modèle dénotationnel pour l'interprétation des types, mais il est le plus adapté, de par sa simplicité, pour exprimer et définir naturellement le polymorphisme de types.

3.2.3 Les polymorphismes

Étant donné que les idéaux sur $\mathcal{V}$ ne sont pas disjoints, mais que $\mathcal{I}_\mathcal{V}$ est un ensemble muni d'un ordre partiel complet non aplati (induit par l'inclusion ensembliste), une même valeur de $\mathcal{V}$ peut appartenir à plusieurs idéaux. Lorsque l'on se ramène aux langages de programmation, cela signifie qu'une valeur peut appartenir à plusieurs types. Il s'agit de la notion de *polymorphisme*, par opposition au *monomorphisme* qui défend l'unicité du type d'une valeur.

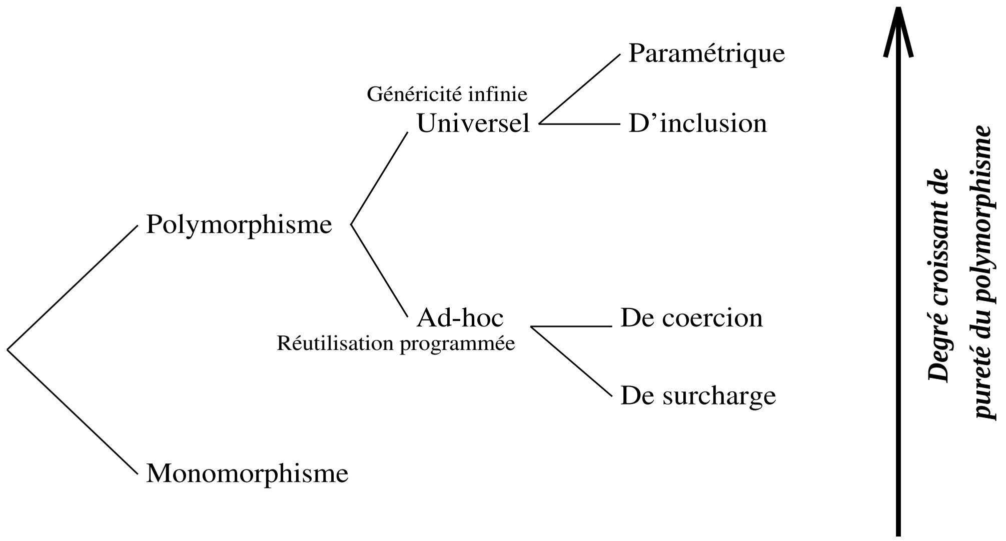


FIG. 3.1 - : Classification des systèmes de types relativement à leur pouvoir d'abstraction. Plus le polymorphisme est pur, plus le pouvoir d'abstraction est élevé.

Christopher Strachey a identifié deux catégories de polymorphismes [Str67] : le polymorphisme *paramétrique*, et le polymorphisme *ad-hoc*. Luca Cardelli et Peter Wegner ont ensuite affiné cette distinction (figure 3.1, extraite de [CW85]), en particulier pour rapprocher le polymorphisme et l'héritage des langages à objets.

[1] L'ensemble des idéaux (des types) formant un treillis, une autre propriété très intéressante du modèle de treillis des idéaux est que les types récursifs peuvent être définis comme le plus petit point fixe d'une équation de type récursif [MPS86].

Polymorphisme ad-hoc

Le polymorphisme ad-hoc concerne des systèmes dont les fonctions opèrent sur différents types n'ayant pas forcément une structure commune, ces fonctions pouvant se comporter différemment d'un type à l'autre. Du point de vue de l'implémentation, une fonction polymorphe exécute des codes différents pour des types de paramètres différents. Ceci montre qu'il s'agit d'un polymorphisme limité, dans le sens où ces fonctions ne peuvent opérer que sur un nombre fini de types. On peut alors dire qu'une fonction polymorphe est un ensemble fini de fonctions monomorphes. Suite à un tel discours, Cardelli et Wegner ont qualifié le polymorphisme ad-hoc de "faux-polymorphisme".

Il existe deux spécialisations, qui se recoupent parfois, des systèmes polymorphiques "ad-hoc" :

- *le polymorphisme de cœrcion* est une caractéristique des systèmes de types dans lesquels une opération sémantique convertit la valeur d'un argument de fonction vers le type attendu par cette fonction, et ce pour éviter une erreur,
- *le polymorphisme de surcharge* désigne une propriété de ces systèmes dans lesquels un même symbole (une variable de même nom) peut être utilisé pour dénoter différentes fonctions, c'est-à-dire différents codes, selon le type et/ou le nombre des arguments (c'est par exemple le cas de C++). Il s'agit en fait d'une abbréviation syntaxique dans le sens où un pré-traitement permet d'éliminer la surcharge de nom.

Polymorphisme universel

À l'opposé du polymorphisme ad-hoc, même si les objectifs se recoupent, le polymorphisme universel caractérise ces systèmes de types dont chaque fonction dite "universellement polymorphe" opère de façon uniforme sur une collection de types ayant une structure commune. Du point de vue de l'implémentation, et pour appuyer le contraste avec le polymorphisme ad-hoc, une fonction universellement polymorphe exécute le même code, quel que soit le type de ses arguments. C'est pour cette raison que ces fonctions sont dites "génériques", puisqu'elles peuvent s'appliquer à un nombre de types potentiellement infini. Une des conséquences de cette propriété du polymorphisme universel est qu'il confère au système de types un pouvoir expressif plus grand et une plus grande extensibilité, méritant alors le titre de "polymorphisme pur" décerné par Cardelli et Wegner. Parmi les langages dotés d'un système de types ayant cette propriété, citons EIFFEL, qui est un langage à objets intégrant vérifications statiques de types et polymorphisme d'inclusion

De même que le polymorphisme ad-hoc, l'universel admet deux sous-catégories :

- *le polymorphisme paramétrique* est ainsi appelé car l'uniformité des structures de types est normalement assurée grâce aux paramètres de types, c'est-à-dire qu'une fonction paramétrée a un paramètre de type (implicite ou explicite), qui détermine le type des arguments pour chaque application de cette fonction,
- *le polymorphisme d'inclusion* a été introduit pour établir formellement, dans le cadre de la programmation par objets, la correspondance entre héritage et polymorphisme, de par les liens forts qui les unissent avec le sous-typage. Il est fondé sur le typage des classes, et sur le fait qu'un objet peut être vu comme appartenant à différentes classes qui ne sont pas forcément disjointes, et même plus précisement liées par l'héritage (inclusion des classes).

La notion de polymorphisme est effectivement proche de la notion d'héritage rencontrée en programmation et en représentation par objets. D'ailleurs, le polymorphisme est une propriété largement rencontrée dans les systèmes de types dédiés aux langages à objets, même s'il n'en est pas toujours fait explicitement état. En effet, l'*héritage* illustre le polymorphisme, classant des ensembles de types similaires par la factorisation de propriétés communes engendrant un super-type, que l'on peut voir comme le type polymorphe résultant [Weg86]. Peter Wegner souligne en outre, de façon très pertinente, que les types polymorphes étendent les mécanismes de classification des langages de programmation, des valeurs vers les types. Autrement dit, le polymorphisme provient en fait du besoin des systèmes de ne pas classer seulement les valeurs (instances), mais aussi les types (classes) : "*When types are viewed as a mechanism for classifying values, then polymorphic types may naturally be interpreted as a mechanism for classifying classes.*" ([Weg86][p.174]).

Un système de types universellement polymorphe associé à un langage est garant, entre autres, de la capacité expressive de ce langage.

3.2.4 Les types abstraits de données

Les types abstraits ont été introduits dans les langages de programmation dans le but de combler le manque d'abstraction de données [Gut77], puisque dans un même temps, la programmation modulaire était permise grâce aux routines, procédures et fonctions. L'objectif principal des types abstraits de données était ainsi de fournir aux programmeurs la possibilité de définir des données et d'utiliser les opérations sur ces données, sans avoir à les implémenter.

En ce sens, la notion de type abstrait de données (ADT) est une extension des types pré-définis dans les langages, tels que `Entier` ou `Chaîne`. En effet, un programmeur utilise les valeurs de ces types et les opérations sur ces valeurs, sans s'inquiéter de leur implémentation en machine. Cette confiance marquée par le programmeur envers l'implémentation de ces types vient du fait qu'il connaît les propriétés de ces types de données ainsi que les caractéristiques des opérations qu'il utilise. De ce fait, même si le programmeur ne connaît pas l'implémentation d'un type de données, il doit en connaître sa spécification exacte et complète (structuration des données et sémantique des opérations) afin d'en apprécier la validité au regard de l'utilisation qu'il en fait.

La spécification formelle des ADT peut être axiomatique ou algébrique [GH78]. Ces deux formalismes de spécification d'un type ont pour but la caractérisation des données de ce type, des propriétés de ces données, et des opérations applicables sur ces données. Nous nous intéresserons plus particulièrement aux spécifications algébriques, dans la mesure où elles présentent l'avantage de permettre la définition de nouveaux types de données en fonction de ceux existants.

La spécification algébrique minimale d'un ADT est un triplet $\langle S, F, E \rangle$, où :

- S est un ensemble de noms de domaines (ou de noms de types),
- F est un ensemble d'opérateurs à paramètres et résultats dans S,
- et E est un ensemble d'équations, qui traduisent les propriétés de ces opérateurs.

Ces trois ensembles peuvent être affinés, de façon à distinguer, parmi les opérateurs, lesquels sont en réalité des constructeurs et lesquels sont des opérations sur les valeurs (figure 3.2). En effet, la combinaison des opérateurs et des équations reflète parfois la définition inductive de l'ensemble des données dénoté par le type ainsi défini, si l'on considère que les données de la base inductive sont des opérateurs constants.

Les données du type abstrait, ainsi que ses opérateurs, sont alors utilisables dans le contexte

Spécification **`Naturel`**

Sortes Naturel, Bool — S

Constructeurs
zero : -> Naturel
succ : Naturel -> Naturel

Opérateurs
. + . : Naturel -> Naturel
maximum: Naturel,Naturel -> Naturel
equal : Naturel, Naturel -> Bool

F

Variables
n,n' : Naturel

Équations — E
zero+n = n
succ(n)+n'=succ(n+n')
equal(zero,zero) = T
equal(zero,succ(n)) = F
equal(succ(n),succ(n')) = equal(n,n')
equal(n,n') = equal(n',n)
maximum(zero,zero) = zero
maximum(zero,succ(n)) = succ(n)
maximum(succ(n),succ(n')) = maximum(n,n')
maximum(n,n') = maximum(n',n)

FIG. 3.2 - : Spécification algébrique du C-type `Naturel`, issue de la définition inductive de Peano

d'exportation de l'implémentation de ce type, l'utilisateur étant renseigné sur le comportement exact des données du type.

3.3 Classes de types : les C-types

MetÉo est dédié d'une part à l'interprétation en intension des entités et ordres définis dans le modèle de connaissances, et d'autre part à résoudre la problématique qui concerne l'intégration de nouvelles structures de données non disponibles dans le langage hôte, et pour lesquelles le langage de représentation n'est pas adapté, et n'a pas à s'adapter. Nous nous attachons, dans cette section, à exposer le niveau de MetÉo qui répond à la seconde raison d'être de ce système. MetÉo doit ainsi fournir un cadre pour la spécification et l'intégration homogène de types de données tels que ceux rencontrés classiquement en programmation, c'est-à-dire des ensembles de valeurs structurées sur lesquelles des opérations sont activables. La propriété de polymorphisme est alors intéressante à mettre en œuvre, dans la mesure où nous souhaitons la possibilité d'enrichissement du nombre de structures de données.

MetÉo est un système de types qui considère deux niveaux de typage. Cette section explore la notion de C-type, et la section 3.4 est concernée par l'étude des δ-types.

Le système de types MetÉo est conçu, entre autres, pour permettre le dévelopement de structures de données et des opérations sur ces structures, dans un langage autre que celui dédié à la représentation des connaissances. De ce fait, MetÉo définit la notion de C-type, qui est dédiée à la représentation d'ensembles de valeurs structurées auxquelles correspondent des collections d'opérations.

Un C-type regroupe un ensemble de données dont il définit la structure et les opérations sur ces structures. Nous les appelons abusivement classes de types dans la mesure où à un C-type est associé un ensemble de δ-types, comme nous le verrons dans la section suivante.

La définition d'un C-type est indépendante du modèle de connaissances. Cependant, la spécification d'un C-type doit être accessible au modèle puisque les données qu'il regroupe doivent pouvoir être utilisées dans une base de connaissances avec le statut de *valeur*. Nous avons vu que le choix d'un ensemble de valeurs pour quantifier une propriété d'une entité de représentation est motivé par les propriétés de ces valeurs et par les opérations activables. En ce sens, la spécification de ces propriétés et de ces opérations doit aussi être connue du système de représentation.

Un concepteur de base de connaissances doit ainsi pouvoir utiliser, en toute connaissance de cause, des données définies par un C-type, ainsi que les opérations sur ces données, sans pour autant avoir à les implémenter. On rejoint ici la raison majeure de l'existence des *types abstraits de données* rencontrés en programmation et spécification de logiciels.

3.3.1 Les C-types vus par le système de représentation

Compte-tenu de ce qui a été dit précédemment, *du point de vue du modèle de connaissances, un C-type est un type abstrait de données, dont l'implémentation est effectuée dans le système de types, ce dernier exportant les opérateurs du type pour qu'ils puissent être utilisés au sein même d'une base de connaissances.*

On propose la possibilité de définition de types abstraits de données dans un système de représentation des connaissances, pour permettre la définition d'ensembles de valeurs sur lesquelles des opérations sont applicables. L'intérêt de cette fonctionnalité est double :

- éviter l'utilisation du langage de représentation pour de telles définitions, tout en allégeant le travail du concepteur d'une base de connaissances qui n'a pas à définir des concepts indépendants de son application.
- manipulation à un même niveau (perception du concepteur d'une base de connaissances) de tout type de données, les données n'étant pour lui qu'un domaine réciproque. En particulier, les données de ces types seront considérées par le modèle de connaissances comme des valeurs, sans dénotation dans le monde réel. À ce titre, valeurs entières ou valeurs issues d'un type de données plus complexe appartiennent toutes au domaine récursif de Scott.

METÉO offre au système de représentation une batterie de C-types, caractérisés par leur spécification, qui regroupe des types de base tels que `Entier` ou `Chaîne`, des types plus élaborés comme `Date` ou `Point`, ainsi que des constructeurs de types, parmi lesquels `Liste`, `Ensemble` ou `Record`. Notons ici que les sortes qui apparaissent dans les spécifications algébriques doivent toutes correspondre à un C-type défini.

Le développeur d'une base de connaissances a accès aux valeurs et opérations des C-types de METÉO. Plus concrètement, le type principal d'un attribut de classe (de concept) est un C-type, les valeurs de ce C-type sont les valeurs de cet attribut pour les instances de la classe (du concept). Les opérations sur ces valeurs sont utilisables au niveau des attachements procéduraux, ainsi qu'au niveau de la pose de contraintes.

3.3.2 Organisation des C-types dans Metéo

METÉO dispose d'une base de C-types qu'il réalise à partir de leur spécification. Les spécifications des C-types sont disponibles au niveau du modèle de connaissances, mais l'implémentation est confiée à METÉO.

Définition

Les C-types qui constituent le système METÉO correspondent à la notion de type telle qu'elle est habituellement définie dans les systèmes informatiques. Un C-type est en ce sens la réalisation d'une structure de données et des opérations sur ces données. Ils correspondent à l'implémentation, *indépendante du modèle de connaissances*, des ADT dont les spécifications sont disponibles dans le système de représentation.

Un C-type est interprété comme un ensemble de valeurs, caractérisé par la définition algébrique de ce C-type. Toutefois, il serait faux d'affirmer qu'un C-type représente un idéal de $\mathcal{V}$ dans la mesure où la condition 2 de la définition 13 n'est pas toujours vérifiée.

Les C-types sont partiellement ordonnés en treillis, par la relation de *C-sous-typage*, selon la sémantique de l'inclusion ensembliste des ensembles de valeurs dénotés. Cet ordre est représenté par un arbre dont le sommet (type `Univers`) dénote l'univers des valeurs défini par Scott. En conséquence, un C-type dénote un sous-ensemble du domaine récursif $\mathcal{V}$.

Définition 16 (C-type) *Un C-type T est la donnée d'un quadruplet $\langle T', C, F, P\rangle$ où :*

- *T' est le C-super-type de T,*
- *C contient les opérateurs de construction des valeurs du C-type,*
- *F contient les fonctions sur ces valeurs,*
- *P est constitué d'au moins trois prédicats, le premier testant l'appartenance d'une valeur quelconque à ce C-type ($\in_T (v)$), le deuxième testant l'égalité de deux valeurs ($=_T (v, v')$), et le troisième testant l'ordre sur les valeurs ($\sqsubseteq_T (v, v')$).*

Dans la définition d'un type abstrait de données standard, les prédicats de P n'apparaissent pas explicitement. En réalité, comme nous le verrons par la suite, ces prédicats peuvent être déduits de l'ensemble C des constructeurs.

Notons les similitudes qu'il existe entre la définition d'un C-type et la définition d'un type abstrait qui a été donnée en section 3.2.4, exceptée la donnée des sortes utilisées par un C-type, qui, dans MeTéo, sont par défaut toutes accessibles. La donnée d'un C-super-type s'apparente à l'ordre établi sur les sortes dans les algèbres de sortes ordonnées [Erw93]. Les équations portant sur les opérateurs des C-types ne sont pas données dans la définition de ce C-type, elles sont implicitement présentes dans le code de ces opérateurs, puisqu'un C-type est l'implémentation d'un type abstrait.

Un C-type est défini incrémentalement à partir de son C-super-type, dont il peut bénéficier des diverses opérations, fonctions et prédicats.

Définition 17 (C-sous-typage) *Soient T_1 et T_2 deux C-types, avec $T_2 = \langle T, C_2, F_2, P_2\rangle$. Un lien de C-sous-typage entre T_1 et T_2 est noté $T_1 \leq_C T_2$ et se définit comme suit :*

$$T_1 \leq_C T_2 \Leftrightarrow \begin{cases} T_1 = \langle T_2, C_1, F_1, P_1\rangle \\ \qquad et \\ \Sigma(C_2) \subseteq \Sigma(C_1), \Sigma(F_2) \subseteq \Sigma(F_1) \\ et \\ \forall e,\ si\ \in_{T_1}(e),\ alors\ \in_{T_2}(e) \\ \qquad et \\ \forall e, e'\ tels\ que\ \in_{T_1}(e)\ et\ \in_{T_1}(e'),\ si\ =_{T_1}(e,e')\ alors\ =_{T_2}(e,e') \\ \qquad et \\ \forall e, e'\ tels\ que\ \in_{T_1}(e)\ et\ \in_{T_1}(e'),\ si\ \sqsubseteq_{T_1}(e,e')\ alors\ \sqsubseteq_{T_2}(e,e') \end{cases}$$

où $\Sigma(E)$ est l'ensemble des signatures des opérations de E.

D'après la définition 17, un lien de C-sous-typage ne peut être vérifié que dynamiquement, c'est-à-dire lors d'une exécution. Le C-sous-typage traduit plus que l'inclusion ensembliste, il concerne aussi la spécialisation d'un comportement, telle qu'elle est considérée dans les langages de programmation par objets. Ceci a été traduit dans la définition précédente par une condition simplifiée portant sur la spécialisation des signatures des ensembles d'opérateurs, mais une telle spécialisation ne peut garantir que le raffinement des opérations respecte toujours la sémantique ensembliste associée aux types. Il s'agit d'un problème de vérification qui est source de très nombreuses recherches, dans le domaine des langages de programmation par objets, mais dont les limites de résolution sont connues. En conséquence, nous ne traitons volontairement pas de ce problème dans METÉO, et METÉO considère que la relation de C-sous-typage est une relation "assertée" : un C-type est déclaré être un C-sous-type d'un autre, sans qu'aucune vérification ne soit effectuée.

Le C-sous-typage montre que METÉO est un système à polymorphisme d'inclusion, puisque les opérateurs et les constructeurs d'un C-type T peuvent être appliqués (sauf redéfinition éventuelle) à toutes valeurs des C-sous-types de T.

METÉO distingue deux catégories de C-types : les C-types prédéfinis, et les C-types construits. Les premiers sont représentés dans METÉO à partir de définition existantes dans le langage hôte, les seconds sont entièrement spécifiés et définis par METÉO, à partir d'autres C-types (parmi lesquels les C-types prédéfinis bien entendu).

C-types prédéfinis

METÉO est construit au dessus d'un langage de programmation, qui définit certains types tels que `Entier` ou `Date`. METÉO utilise ces définitions pour réaliser les C-types correspondant. On parle alors de C-types prédéfinis : les constructeurs de valeurs de ces C-types sont implicitement récupérés du langage de programmation, de même qu'un sous-ensemble des fonctions sur les valeurs. METÉO récupère de même les trois prédicats qui sont définis par le langage de programmation.

Ces types prédéfinis sont donc explicitement représentés par des C-types (donc en termes du langage de types de METÉO), mais leur interprétation dans le domaine récursif de Scott est celle du langage hôte.

Constructeurs de C-types

Nous appelons *constructeur de type* tout type dont les éléments se construisent à partir d'éléments d'un ou plusieurs autres C-types, tels que `Liste`, `Point` ou `Record`. Les C-types qui sont référencés pour la définition d'un autre peuvent être généraux et substituables. Cette possibilité est une conséquence immédiate du polymorphisme d'inclusion qu'autorise METÉO. Par exemple, METÉO définit le C-type `Liste(X)` où `X` est un paramètre à valeurs dans l'ensemble des C-types existants. Il ne s'agit pas exactement de polymorphisme paramétrique dans la mesure où les C-types en paramètres ne peuvent être substitués que par leurs C-sur-types. Dans METÉO, `Liste(X)` est en réalité `Liste(Univers)`, où `Univers` admet comme C-sous-types tous ceux définis dans METÉO.

La définition 16 d'un C-type est plus qu'une simple caractérisation de l'ensemble de valeurs dénoté par un C-type. En effet, les opérations sur les valeurs, autres que les constructeurs, ne caractérisent pas cet ensemble mais s'attachent à donner une raison d'être au regroupement de ces valeurs, de par l'utilisation que l'on peut en faire (on dit abusivement que les valeurs partagent un *comportement*, les fonctions étant le reflet de ce comportement). Le prédicat d'appartenance peut être déduit des constructeurs (une valeur appartient à un type si elle peut être élaborée par application des opérateurs de construction de ce type), de même que le prédicat d'egalité (deux valeurs sont égales pour un type si elles peuvent être construites par la même application des constructeurs). Quant au prédicat d'ordre, il se définit lui aussi à partir des constructeurs, de façon à refléter l'ordre partiel complet dont est muni le domaine récursif de Scott.

C'est donc à partir de ses constructeurs que l'on peut définir complètement la dénotation d'un C-type. Un contructeur $\chi \in C$ est toujours à valeur dans le C-type qu'il définit. Il est soit d'arité nulle, il s'agit dans ce cas d'un opérateur de construction constant (comme `nil` pour les listes), soit la combinaison d'un nombre fini de valeurs. Cette combinaison est limitée à être l'application fonctionnelle, le produit cartésien ou la somme, de façon à pouvoir être interprétée dans $\mathcal{V}$.

Définition 18 (Opérateur de construction d'un C-type) *Soit χ un opérateur de construction (un constructeur) du type T. χ est syntaxiquement défini par :* $\chi = \chi^0 | \chi \rightarrow \chi | \chi + \chi | \chi \times \chi$

Si ξ est la fonction d'interprétation qui à un type T associe son sous-ensemble dans $\mathcal{V}$, à un élément de type associe sa valeur associée dans $\mathcal{V}$ (la valeur dans $\mathcal{V}$ associée à un symbole e (élément d'un type) est notée $\dot{e}$), et à un constructeur de valeur associe son domaine de valeurs dans $\mathcal{V}$, nous pouvons formellement interpréter un C-type T dans $\mathcal{V}$ comme suit :

Définition 19 (Sémantique d'un C-type) *Soit $T = \langle T', C, F, P \rangle$, l'interprétation d'un symbole S par ξ dans $\mathcal{V}$ est notée $\|S\|^\xi$.*

- $\|T\|^\xi \subseteq \|T'\|^\xi \bigcap \|C\|^\xi$
- $\|C\|^\xi = \{\|\chi_i\|^\xi\}_{i \in I}$
- $\|\chi^0\|^\xi = \{v \in \mathcal{V} | v = \dot{\chi}\}$
- $\|\chi_1 \rightarrow \chi_2\|^\xi = \{v \in \mathcal{V} | \exists v_1 \in \|\chi_1\|^\xi, v_2 \in \|\chi_2\|^\xi, v = v_1 \rightarrow v_2\}$
- $\|\chi_1 \times \chi_2\|^\xi = \{v \in \mathcal{V} | \exists v_1 \in \|\chi_1\|^\xi, v_2 \in \|\chi_2\|^\xi, v = v_1 \times v_2\}$
- $\|\chi_1 + \chi_2\|^\xi = \{v \in \mathcal{V} | \exists v_1 \in \|\chi_1\|^\xi, v_2 \in \|\chi_2\|^\xi, v = v_1 + v_2\}$

P et F ne participent pas à l'interprétation ensembliste d'un C-type dans la mesure où ils contiennent des opérations de manipulation ou de consultation des valeurs de l'ensemble dénoté par le C-type. Autrement dit, seuls les opérateurs de contruction et la référence au C-sur-type font partie de l'identification de l'ensemble dénoté par le C-type. C'est pour cette raison que P et F n'apparaissent pas dans la définition sémantique d'un C-type.

La définition des C-types qui correspondent à des constructeurs est inspirée de la théorie des types constructifs [ML79], dans le sens où cette définition comprend obligatoirement la définition des opérateurs de formation de valeurs de ces types.

La définition d'un C-type comprend en outre les principales opérations sur les valeurs construites. Ces opérations sont essentielles dans la mesure où ce sont elles qui donnent un sens à l'existence d'un C-type. Lorsqu'un paramètre d'un constructeur est substitué, ce qui a pour effet la création d'un C-sous-type de ce constructeur, certaines opérations sont automatiquement spécialisées (au sens du polymorphisme d'inclusion), et d'autres peuvent être définies, pour prendre en compte les

C-type `Liste` **est**

Sortes X, Liste, Bool

Opérateurs

nil : -> Liste
cons: X, Liste -> Liste
car: Liste -> X
cdr: Liste -> Liste
remove: X, Liste -> Liste
member: X, Liste -> Bool
append: Liste, Liste -> Liste
vide: Liste -> Bool
equal: Liste, Liste -> Bool

Équations

car(nil) = nil
car(cons(e,l)) = e
cdr(nil) = nil
cdr(cons(e,l)) = l
append(nil,l) = l
append(l,nil) = l
append(l,l') = cons(car(l),append(cdr(l),l'))
remove(e,nil) = nil
remove(e, cons(e,l)) = remove(e,l)
remove(e,cons(e',l)) = cons(e',remove(e,l))
member(e,nil) = F
member(e,cons(e,l)) = T
member(e,cons(e',l)) = member(e,l)
vide(nil) = T
vide(cons(e,l)) = F
equal(nil,nil) = T
equal(cons(e,l),cons(e',l)) = false
equal(cons(e,l),cons(e,l')) = equal(l,l')
equal(cons(e,l),nil) = F
equal(l,l') = equal(l',l)

FIG. 3.3 - : Spécification algébrique partielle du C-type `Liste`, prenant en paramètre n'importe quel autre C-type, ici noté X. Tous les opérateurs de la spécification sont implémentés par MÉTÉO, en respectant les propriétés traduites par les équations, et utilisables au niveau du système de représentation.

C-type `Liste-Entier` **est**

Affine `Liste` **avec** X <- Entier

Complète `Liste` **avec**

Sortes Entier, Liste-Entier, Bool + {$\perp$}

Opérateurs

min: Liste-Entier -> Entier
tri-croissant: Liste-Entier -> Liste-Entier
incrémente: Liste-Entier -> Liste-Entier

Équations

min(nil) = $\perp$
min(cons(e,nil)) = e
min(cons(e,cons(e',l)) = min(cons(*minimum* (e,e'), l))
tri-croissant(nil) = nil
tri-croissant(l) = cons(min(l),tri-croissant(remove(min(l),l)))
incrémente(nil) = $\perp$
incrémente(cons(e,l)) = cons(*succ* (e),incrémente(l))

FIG. 3.4 - : Spécification du C-type `Liste(Entier)`. Par rapport à la spécification de `Liste`, des opérateurs et leur définition sont ajoutés, qui considèrent les opérations disponibles dans le C-type en paramètre.

opérations existant sur les valeurs du type en paramètre. La figure 3.3 montre une spécification simplifiée du C-type `Liste(X)`, alors que la figure 3.4 montre une "instanciation" du C-type `Liste(X)` où X est le type `Entier`.

La définition 16 indique qu'il est nécessaire, pour tout C-type, de spécifier le prédicat d'ordre entre deux valeurs du C-type, de façon à ce que l'ensemble des valeurs dénotées par un C-type puisse être plongé dans l'ordre partiel dont est muni le domaine récursif de Scott. Si pour certains types de données la relation d'ordre est significative et naturelle, comme chez les entiers ou les chaînes de caractères, elle ne l'est pas pour tous. Par exemple, définir un ordre sur des listes est possible mais il ne constitue pas une des raisons d'être de ce type de données. Dans ce cas, un ordre est tout de même défini, mais il s'agit d'un ordre *plat*, à savoir que les valeurs ne sont pas comparables entre elles, exceptées l'élément maximum (plus grand que tout autre valeur) et l'élément minimum (plus petit que toutes les autres valeurs). Tout C-type, qu'il définisse un ordre plat ou non, doit en conséquence désigner explictement un plus grand élément et un plus petit, de façon à ce que l'ordre puisse être représenté par un treillis. La donnée de ces deux éléments significatifs fait partie de la définition du prédicat d'ordre. Pour un C-type T, nous désignons par $\lceil T \rceil$ le plus grand élément de T, et par $\lfloor T \rfloor$ le plus petit. Tout C-type définissant (par des symboles) ces deux éléments, il peut récupérer du C-type `Univers` la définition du prédicat d'ordre suivante :

- $\forall e$ tel que $\in_{Univers}(e), \sqsubseteq_{Univers}(e, \lceil Univers \rceil)$
- $\forall e$ tel que $\in_{Univers}(e), \sqsubseteq_{Univers}(\lfloor Univers \rfloor, e)$
- $\forall e_1, e_2$ tels que $\in_{Univers}(e_1)$, $\in_{Univers}(e_2)$, si $\neq_{Univers}(e_1, e_2)$, $\neq_{Univers}(e_1, \lfloor Univers \rfloor)$, $\neq_{Univers}(e_1, \lceil Univers \rceil)$, $\neq_{Univers}(e_2, \lfloor Univers \rfloor)$ et $\neq_{Univers}(e_2, \lceil Univers \rceil)$ alors $\not\sqsubseteq_{Univers}$ (e_1, e_2).

La récupération de cette définition par tout C-type T est possible, avec substitution du C-type `Univers` par T, du fait du polymorphisme d'inclusion de MetéO. Ce prédicat d'ordre peut bien entendu être redéfini par d'autres C-types pour lesquels l'ordre est significatif : dans ce cas, le prédicat défini par `Univers` est simplement complété par les informations relatives à l'ordre sur les autres valeurs.

Exemple des *Records*

Une *valeur record* est un ensemble fini non ordonné de valeurs étiquetées, comme par exemple $v = \langle e_1 = 21, e_2 = true, e_3 = abcd \rangle$, qui dénote dans le domaine récursif de Scott la même valeur que $v = \langle e_1 = 21, e_3 = abcd, e_2 = true \rangle$ [Car84].

L'association étiquette/valeur peut être vue comme l'application fonctionnelle, une valeur *record* est ainsi le produit cartésien d'un ensemble fini d'applications des étiquettes vers les valeurs.

Nous choisissons de définir dans MetéO le C-type qui dénote l'ensemble de toutes les valeurs *records*. Cela revient à identifier son super-type, les constructeurs de valeurs *records*, les opérations principales sur ces valeurs, ainsi que les trois prédicats nécessaires (appartenance, égalité, ordre).

- Les valeurs *records* sont construites à partir de valeurs d'autres types. Pour cette raison, le C-type `Record` est un constructeur de type, C-sous-type de `Construits`.
- Il existe deux opérations pour la construction de valeurs *records* [CM91].

  record vide $\langle\ \rangle$, qui ne contient aucune association étiquette/valeur (opérateur constant)

  extension $\langle r | e = v \rangle$, qui ajoute l'étiquette e associée à la valeur v dans la valeur *record* r, à condition que l'étiquette e ne soit pas déjà présente dans r.
- Les opérations légales et significatives sur les valeurs *records*, outre les opérations de construction, sont les suivantes :

  restriction $r' = r \backslash e$, qui enlève de la valeur *record* l'association dont l'étiquette est e, à condition que cette étiquette existe. Cette fonction a pour résultat la valeur *record* r'.

extraction $r : e$, qui a pour résultat la valeur associée à l'étiquette e dans la valeur *record* r, à condition que cette étiquette existe dans r.

À partir des constructeurs et des deux opérations ci-dessus, il est possible de définir d'autres opérations plus complexes, comme par exemple :

renommage $r[e \leftarrow e']$ qui renomme dans r l'étiquette e en l'étiquette e'.

surcharge $\langle r \leftarrow e = v \rangle$, qui, si l'étiquette e est présente, modifie sa valeur associée.

concaténation $r \& r'$, qui est la concaténation des deux ensembles d'associations, valides à condition que les étiquettes de r soient différentes des étiquettes de r'.

- Le prédicat d'égalité entre deux valeurs est obtenu, par induction, à partir des opérateurs de construction :

  1. $r = \langle\ \rangle, r' = \langle\ \rangle \Rightarrow =_{record} (r, r')$
  2. $r = \langle r_1 | e = v \rangle, r' = \langle r_2 | e = v \rangle, =_{record} (r_1, r_2) \Rightarrow =_{record} (r, r')$

  De même, le prédicat d'appartenance se définit comme suit :

  1. $r = \langle\ \rangle \Rightarrow \in_{record} (r)$
  2. $r = \langle r' | e = v \rangle, \in_{record} (r') \Rightarrow \in_{record} (r)$

- L'ordre sur les valeurs *records* est un ordre partiel qui forme un treillis plat, dont le sommet est représenté par l'expression $\lceil vrec \rceil$ et le plus petit élément est la valeur *record* vide $\langle\ \rangle$. Ainsi, cet ordre se définit comme suit :

  1. $\sqsubseteq_{record} (\langle\ \rangle, \lceil vrec \rceil) = true$
  2. $r = \langle r' | e = v \rangle \Rightarrow \sqsubseteq_{record} (\langle\ \rangle, r) = true$ et $\sqsubseteq_{record} (r, \lceil vrec \rceil) = true$
  3. $r = \langle r_1 | e_1 = v_1 \rangle, r' = \langle r_2 | e_2 = v_2 \rangle \Rightarrow \sqsubseteq_{record} (r, r') = false$ et $\sqsubseteq_{record} (r', r) = false$

Le prédicat d'ordre sur les valeurs *records* est en fait récupéré de celui défini pour le C-type `Univers`.

Arbre de C-sous-typage

Actuellement, MÉTÉO fournit une base de C-types ordonnés en arbre (figure 3.5). Chacun de ces C-types est entièrement spécifié et réalisé. La propriété de C-sous-typage est exploitée bien qu'arbitraire : il ne s'agit pas d'un C-sous-typage inféré mais asserté.

La figure 3.5 montre une hiérarchie arbitraire de C-types, dont l'ordre est guidé par l'inclusion ensembliste. La raison d'être des C-types intermédiaires, tels que `Ordonnés`, est leur capacité à grouper un ensemble de propriétés génériques qui sont communes à leurs C-sous-types. En conséquence, les opérations issues de ces propriétés peuvent être elles aussi définies de façon générique au niveau des C-types intermédiaires.

Tous ces C-types sont utilisables au niveau du modèle de connaissances, qu'ils soient intermédiaires ou feuilles, dans la mesure où leur spécification est donnée et leurs opérations exportées.

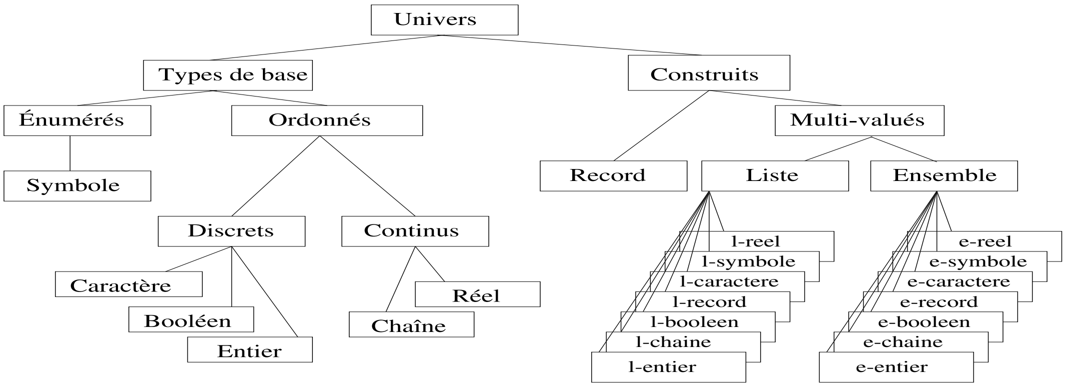


FIG. 3.5 - : Arbre de C-types de METÉO. Le sommet de l'arbre est le type `Univers`, dont l'ensemble de valeurs est celui formellement défini par Scott. Les types construits et les types de base se distinguent, les premiers étant ceux dont les valeurs sont construites à partir d'autres C-types, et les seconds pouvant être qualifiés d'"intra-définis" dans la mesure où ils ne dépendent d'aucun autre C-type, exceptés leurs sur-types. Chacune de ces deux catégories est à son tour affinée, de façon à regrouper des ensembles intéressants de propriétés et d'opérations sur les valeurs. Par exemple, le C-type `Ordonnés` est explicitement défini : l'ordre sur les valeurs est ici exploité pour la réalisation d'opérations particulières sur cet ordre. L'ordre défini dans `Ordonnés` sera affiné dans les C-sous-types, mais les opérations génériques sur cet ordre, définies dans `Ordonnées`, pourront être partagées dans ses C-sous-types : c'est une illustration du polymorphisme d'inclusion.

3.3.3 Conclusion

Les C-types sont un support pour la définition de structures de données et d'opérations sur ces données. Les C-types sont la réalisation, dans METÉO, de types abstraits de données dont la spécification est disponible au niveau du modèle de connaissances. En ce sens, la définition des C-types est indépendante du modèle de connaissances. Cette indépendance est essentielle, elle illustre le fait que la définition de structures de données en termes d'un langage de représentation des connaissances n'est pas pertinente.

METÉO, défini sur la base des C-types, répond aux critères d'utilité relatifs à l'abstraction de données et à la construction de nouvelles structures :

- l'abstraction des données est illustrée par l'indépendance entre la définition des structures dans METÉO et leur utilisation dans le modèle de connaissances,
- la construction de nouvelles structures est rendue possible grâce à la propriété de polymorphisme d'inclusion ; une analyse plus précise de l'extensibilité de METÉO fait l'objet de la section 3.6.

Nous avons montré dans cette section de quelle façon la définition de structures de données, dédiées au modèle de connaissances, est possible dans METÉO. Toutefois, l'intérêt d'un système de types réside aussi dans sa capacité à garantir la correction des "programmes". Dans le cadre de la représentation des connaissances, il s'agit donc d'étudier un mode pour l'expression des types des entités de représentation, de façon à permettre la vérification des relations portant sur les extensions de ces entités.

3.4 δ-types

La section précédente a montré que METÉO offre un support pour la définition de structures de données et d'opérations sur ces structures. Nous allons montrer dans cette section que les C-types permettent aussi de définir les structures, et opérations sur ces structures, qui accueilleront l'interprétation intensionnelle des entités de représentation et des relations définies dans TROPES : les δ-types.

Un δ-type est l'expression d'un ensemble de valeurs, sans la donnée des opérations qui s'exécutent sur ces valeurs.

Nous présentons dans cette section la définition, le mode d'expression (en terme d'un langage de types appelé EOLE), l'organisation et les propriétés des δ-types dans METÉO, indépendamment du modèle de connaissances. Cependant, il est important de souligner dès maintenant que les δ-types sont introduits en vue d'être chargés de la représentation, dans METÉO, des types de données associés aux entités de modélisation (qu'il s'agisse indifféremment des types de concepts, de classes, d'attributs et même d'instances comme nous le verrons dans le chapitre suivant).

Du point de vue de METÉO, un δ-type est l'expression d'un ensemble de valeurs, le regroupement de ces valeurs étant, dans l'absolu, indépendant de leur comportement. De ce fait, la différence essentielle entre un C-type et un δ-type est la composante comportementale de l'ensemble de valeurs dénoté.

3.4.1 Définitions

D'une part, un C-type dénote un ensemble de valeurs sur lesquelles il définit des opérations qui traduisent le comportement de ces valeurs. D'autre part, un δ-type dénote lui aussi un ensemble de valeurs, mais il ne définit pas explicitement de comportement sur ces valeurs.

Pourtant, la nécessité de représenter des faits par des valeurs est issue de l'intérêt porté aux opérations sur ces valeurs. Pour cette raison, tout δ-type est attaché explicitement à un C-type, de façon à effectivement pouvoir disposer des opérations de ce C-type sur les valeurs du δ-type. Cet attachement se traduit par le fait qu'*un δ-type dénote un sous-ensemble de l'ensemble des valeurs dénoté par son C-type*. Nous appelons *domaine* d'un δ-type l'ensemble de valeurs qu'il représente.

Le typage d'une entité de représentation du modèle se fera alors en deux phases successives : spécification du C-type (choix de l'ensemble de valeurs et des opérations sur ces valeurs), puis affinement de l'ensemble en un sous-ensemble représenté par un δ-type[2].

Les δ-types sont l'expression des partitions d'un C-type (figure 3.6). Théoriquement, si l'ensemble dénoté par un C-type est infini, il existe une infinité de δ-types issus de ce C-type. L'ensemble des δ-types associé à un C-type T est noté $\Delta(T)$.

Définition 20 (δ-type) *Un δ-type t est un doublet $\langle T, E \rangle$ où :*

- *T est le C-type de rattachement de t,*
- *E est une expression syntaxique normalisée qui représente le domaine de t*

[2] Dans le cas d'un attribut, cet affinement se fait par application des descripteurs statiques et par prise en compte des contraintes.

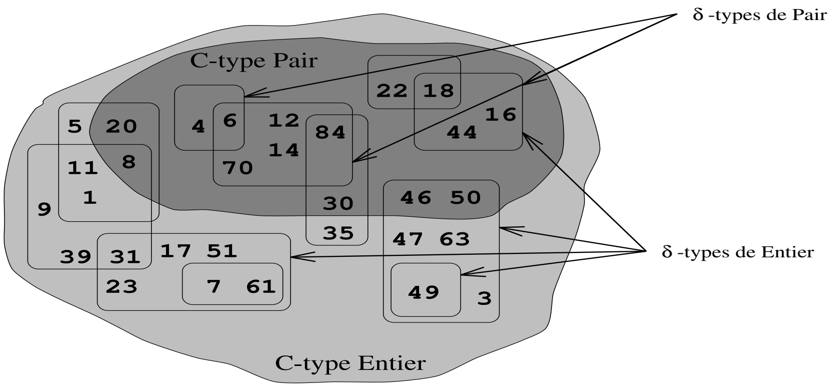


FIG. 3.6 - : Les δ-types sont l'expression de sous-ensembles issus de l'ensemble dénoté par leur C-type de rattachement. Le C-type `Entier` est C-sur-type de `Pair`, les δ-types issus de `Pair` dénotent des sous-ensembles de `Pair` et donc d'`Entier`.

La forme syntaxique E dépend de T, ou plus précisément des propriétés relatives aux constructeurs de T, comme nous le montrons dans la section suivante.

3.4.2 Eole : un langage d'expressions de δ-types

La définition 20 montre qu'un δ-type est la donnée d'un C-type d'attachement et d'une expression syntaxique permettant d'exprimer le domaine du δ-type. Cette expression est normalisée, de telle façon qu'*à deux domaines de valeurs égaux correspond la même expression syntaxique*, autrement dit le même δ-type.

L'expression syntaxique E d'un δ-type t doit prendre en considération les propriétés du C-type T qui peuvent optimiser la représentation de sous-ensembles de valeurs. Par exemple, les sous-ensembles d'un ensemble infini dénombrable (existence des opérations `successeur` et `prédécesseur`) peuvent être complètement exprimés sous la forme d'unions finies d'intervalles à bornes fermées (à condition de pouvoir représenter les infinis supérieur et inférieur).

Chaque C-type possède des propriétés sur ses valeurs, qui influent sur la représentation de domaines sur ces valeurs. Certains C-types ont en commun un mode de représentation car ils présentent des propriétés d'organisation communes. Ceci constitue un deuxième critère pour la hiérarchisation des C-types telle que nous l'avons présentée figure 3.5 (le premier critère étant le respect de l'inclusion ensembliste). Étudions, pour chaque C-type, ou groupe de C-types, les propriétés qui permettent de représenter des formes normales de sous-ensembles de valeurs (donc des δ-types) :

- Un sous-ensemble s de l'ensemble dénoté par **un type construit** est donné soit par énumération de ses valeurs construites, soit par énumération du complémentaire, soit plus simplement par la donnée des sous-ensembles de valeurs sur lesquels les valeurs de s sont construites. La représentation d'un δ-type construit s est donc composée de trois parties :

  type-ref la donnée des δ-types de référence

  domaine la liste exhaustive (énumération) des valeurs constituant s

  comp-dom la liste exhaustive des valeurs constituant $\bar{s}$[3].

[3] La notation $\bar{E}$ dénote l'ensemble complémentaire de E vis-à-vis d'un sur-ensemble de valeurs.

Ce mode de représentation peut être pris tel quel pour représenter des sous-ensembles de valeurs *records*, c'est à dire ce que l'on appelle communément des *types record*. Quant aux types multivalués, un renseignement supplémentaire est capital, qui concerne la cardinalité des valeurs. Dans le cas des multivalués, on ajoute donc :

card la donnée des cardinalités minimum et maximum.

- Un sous-ensemble s de l'ensemble dénoté par un type énuméré ne peut être donné que par son énumération ou l'énumération de son complémentaire. En effet, le C-type **Énumération** concerne justement ces ensembles de valeurs qui ne peuvent être organisés autrement que par énumération explicite. Donc la représentation d'un δ-type énuméré s comporte deux parties :

domaine la liste exhaustive (énumération) des valeurs constituant s

comp-dom la liste exhaustive des valeurs constituant $\bar{s}$.

- Tous les C-types définissent un ordre sur leurs valeurs, mais l'appellation **types ordonnés** dans la hiérarchie de C-sous-typage désigne ces C-types dont l'ordre est explicitement utilisé pour représenter leurs δ-types, au moyen d'ensembles finis d'intervalles[4]. Cette représentation sous forme d'ensembles d'intervalles nécessite par ailleurs un ordre total sur les ensembles. La représentation d'un δ-type s issu d'un C-type (totalement) ordonné est alors obtenue à l'aide d'un seul composant :

domaine une liste finie d'intervalles dont l'union représente le sous-ensemble. Les caractéristiques de ces intervalles changent alors selon les propriétés de l'ensemble dénoté par le C-type. Se distinguent les C-types :

  - `Continus`, dont les sous-ensembles se représentent par des listes d'intervalles à *bornes ouvertes*.
  - `Discrets`, (par opposition à `continus`), ou plus précisément (infinis) dénombrables, réalisant les opérations `successeur` et `prédécesseur`, dont les sous-ensembles peuvent être exprimés par des listes d'intervalles à *bornes fermées*.

- Les C-types `Univers` et `Types de base` n'autorisent que la représentation explicite du domaine de valeur, de par le champ de description **domaine**.

L'étude ci-dessus montre que pour chaque C-type, un certain nombre de champs permet la représentation complète de sous-ensembles de ces C-types. EOLE est un langage d'expressions de δ-types, dont la syntaxe est guidée par ces champs, et dont la sémantique doit être ensembliste, de façon à être intégrée homogènement à la sémantique des C-types.

La syntaxe du langage d'expressions de types EOLE est répartie dans les C-types, puisque ce sont les propriétés organisationnelles de ces C-types qui conditionnent le mode de représentation des δ-types (table 3.1).

Par exemple, le terme EOLE $\langle Entier; [-2;4] + [7;12] + [15; \lceil Entier \rceil] \rangle$ dénote l'ensemble des valeurs entières comprises entre -2 et 4 ou entre 7 et 12 ou supérieures ou égales à 15.

Les δ-types étant des expressions issues des C-types, ils s'interprètent selon le même principe, dans le domaine récursif de Scott $\mathcal{V}$ muni de l'ordre partiel $\sqsubseteq$ sur les valeurs. Soit ξ la fonction

[4] Le fait d'avoir ainsi classé le C-type `chaîne` peut être discuté, car la représentation de sous-ensembles de chaînes de caractères au moyen d'intervalles n'est plus pertinente, du point de vue de l'optimalité, lorsque les valeurs de ce type ne sont utilisées que comme des identificateurs, l'ordre n'étant alors utilisé, à la rigueur, que pour des tris ou des opérations similaires.

δT	$::= \langle T; E(T) \rangle$
T	$::= TU \mid TC \mid TR \mid TO \mid TOD \mid TM \mid TE$
TU	$::=$ `Univers` $\mid$ `Types-Base`
TC	$::=$ `Types-construits`
TR	$::=$ `Record`
TO	$::=$ `Types-ordonnés` $\mid TCT$
TCT	$::=$ `Continus` $\mid$ `Réel` $\mid$ `Chaîne`
TOD	$::=$ `Discrets` $\mid$ `Entier` $\mid$ `Booléen` $\mid$`Caractère`
TM	$::=$ `Multi-valués` $\mid$ `Liste` $\mid$ `Ensemble`
TE	$::=$ `Énumération`
$E(TU)$	$::= [DE]$
$E(TC)$	$::= [TF; DE; DE]$
$E(TR)$	$::= [R; DE; DE]$
$E(TCT)$	$::= [DI]$
$E(TO)$	$::= [DI]$
$E(TOD)$	$::= [DF]$
$E(TM)$	$::= [\delta T; DE; DE; C]$
$E(TE)$	$::= [DE; DE]$
DE	$::= DEL \mid$ `all` $\mid$ `nothing`
DEL	$::= V \ DEL \mid V$
DI	$::= DF \mid DI + I \mid I$
DF	$::= DF + F \mid F$
I	$::= F \mid [V; V[\mid]V; V] \mid]V; V[$
F	$::= [V; V] \mid [;]$
TF	$::= \delta T \mid TF \rightarrow TF \mid TF \times TF \mid TF + TF$
R	$::= \langle\langle \ \rangle\rangle \mid R \times (N \rightarrow \delta T)$
C	$::= [N; N]$
N	$::= 1 \mid 2 \mid \cdots$
V	$::= \text{symbole} \mid \lceil T \rceil \mid \lfloor T \rfloor$

TAB. 3.1 - : Syntaxe de EOLE

d'interprétation à valeurs dans $\mathcal{V}$ (définition 19). ξ interprète un δ-type en lui associant le sous-ensemble de son C-type qu'il dénote. La sémantique des δ-types est donnée dans la table 3.2. La syntaxe et la sémantique de EOLE sont données ici pour les besoins de l'exposé de la normalisation des δ-types, qui est le thème de l'annexe A. Toutefois, par la suite, nous simplifierons cette syntaxe, notamment dans le cas des types construits et des *records*.

3.4.3 δ-sous-typage : définition

L'ordre sur les δ-types est appelé *δ-sous-typage*, et tout comme le C-sous-typage, il peut s'interpréter par l'inclusion ensembliste.

Définition 21 (δ-sous-typage) *La relation de sous-typage entre deux δ-types issus du C-type T est notée $\leq_T^\delta$. Elle représente l'inclusion ensembliste sur $\mathcal{V}$.*

$$\forall T \in \mathcal{T}, \forall t_1, t_2 \in \Delta(T), t_1 \leq_T^\delta t_2 \Leftrightarrow \|t_1\|^\xi \subseteq \|t_2\|^\xi$$

$\delta T = \langle T; E(T)\rangle$	$\|\delta T\|^\xi \subseteq \|T\|^\xi$
$\delta T = \langle T; E(T)\rangle$	$\|\delta T\|^\xi = \|T\|^\xi \bigcap \|E(T)\|^\xi$
$E(TU) = [DE]$	$\|E(TU)\|^\xi = \|DE\|^\xi$
$E(TC) = [TF; DE_1; DE_2]$	$\|E(TC)\|^\xi = (\|TF\|^\xi \bigcap \|DE_1\|^\xi)\backslash\|DE_2\|^\xi$
$E(TR) = [R; DE_1; DE_2]$	$\|E(TR)\|^\xi = (\|R\|^\xi \bigcap \|DE_1\|^\xi)\backslash\|DE_2\|^\xi$
$E(TCT) = [DI]$	$\|E(TCT)\|^\xi = \|DI\|^\xi$
$E(TO) = [DI]$	$\|E(TO)\|^\xi = \|DI\|^\xi$
$E(TOD) = [DF]$	$\|E(TOD)\|^\xi = \|DF\|^\xi$
$E(TM) = [\delta T; DE_1; DE_2; C]$	$\|E(TM)\|^\xi = \{e \in (\|\delta T\|^\xi \bigcap \|DE_1\|^\xi)\backslash\|DE_2\|^\xi \mid card(e) \in \|C\|^\xi\}$
$E(TE) = [DE_1; DE_2]$	$\|E(TE)\|^\xi = \|DE_1\|^\xi\backslash\|DE_2\|^\xi$
$DE = DEL$	$\|DE\|^\xi = \|DEL\|^\xi$
$DE = \texttt{all}$	$\|DE\|^\xi = \mathcal{V}$
$DE = \texttt{nothing}$	$\|DE\|^\xi = \emptyset$
$DEL = V$	$\|DEL\|^\xi = \{e \in \mathcal{V} \mid e = \|V\|^\xi\}$
$DEL = V\ DEL$	$\|DEL\|^\xi = \|V\|^\xi + \|DEL\|^\xi$
$DI = DF$	$\|DI\|^\xi = \|DF\|^\xi$
$DI = DI_1 + I$	$\|DI\|^\xi = \|I\|^\xi \bigcup \|DI_1\|^\xi$
$DI = I$	$\|DI\|^\xi = \|I\|^\xi$
$DF = DF_1 + F$	$\|DF\|^\xi = \|DF_1\|^\xi \bigcup \|F\|^\xi$
$DF = F$	$\|DF\|^\xi = \|F\|^\xi$
$I = F$	$\|I\|^\xi = \|F\|^\xi$
$F = [V_1; V_2]$	$\|F\|^\xi = \{e \in \mathcal{V} \mid \|V_1\|^\xi \sqsubseteq e \sqsubseteq \|V_2\|^\xi\}$
$F = [;]$	$\|F\|^\xi = \emptyset$
$I =]V_1; V_2]$	$\|I\|^\xi = \{e \in \mathcal{V} \mid \|V_1\|^\xi \sqsubset e \sqsubseteq \|V_2\|^\xi\}$
$I = [V_1; V_2[$	$\|I\|^\xi = \{e \in \mathcal{V} \mid \|V_1\|^\xi \sqsubseteq e \sqsubset \|V_2\|^\xi\}$
$I =]V_1; V_2[$	$\|I\|^\xi = \{e \in \mathcal{V} \mid \|V_1\|^\xi \sqsubset e \sqsubset \|V_2\|^\xi\}$
$TF = \delta T$	$\|TF\|^\xi = \|\delta T\|^\xi$
$TF = TF_1 \rightarrow TF_2$	$\|TF\|^\xi = \{v \in \mathcal{V} \mid \exists v_1 \in \|TF_1\|^\xi, \exists v_2 \in \|TF_2\|^\xi$ telles que $v = v_1 \rightarrow v_2\}$
$TF = TF_1 \times TF_2$	$\|TF\|^\xi = \{v \in \mathcal{V} \mid \exists v_1 \in \|TF_1\|^\xi, \exists v_2 \in \|TF_2\|^\xi$ telles que $v = v_1 \times v_2\}$
$TF = TF_1 + TF_2$	$\|TF\|^\xi = \{v \in \mathcal{V} \mid \exists v_1 \in \|TF_1\|^\xi, \exists v_2 \in \|TF_2\|^\xi$ telles que $v = v_1 + v_2\}$
$R = \langle\langle\ \rangle\rangle$	$\|R\|^\xi = \|\texttt{Record}\|^\xi$
$R = R' \times (N \rightarrow \delta T)$	$\|R\|^\xi = \{v \in \mathcal{V} \mid \exists v_1 \in \|R'\|^\xi, v_2 \in \|N\|^\xi$ et $v_3 \in \|\delta T\|^\xi$ telles que $v = v_1 \times (v_2 \rightarrow v_3)\}$
$C = [N_1; N_2]$	$\|C\|^\xi = \{e \in \|N\|^\xi \mid \|N_1\|^\xi \sqsubseteq e \sqsubseteq \|N_2\|^\xi\}$
N	$\|N\|^\xi = \|\texttt{Entier}\|^\xi$
$V =$ symbole	$\|V\|^\xi = \dot{V}$
$V = \lceil T \rceil$	$\|V\|^\xi = e \in \|T\|^\xi$ tel que $\forall e' \in \|T\|^\xi, e' \sqsubseteq e$
$V = \lfloor T \rfloor$	$\|V\|^\xi = e \in \|T\|^\xi$ tel que $\forall e' \in \|T\|^\xi, e \sqsubseteq e'$

TAB. 3.2 - : Sémantique de EOLE (nous rappelons que l'interprétation du terme S par ξ dans $\mathcal{V}$ est notée $\|S\|^\xi$).

Propriété 1 *La relation de δ-sous-typage, sur un C-type T, définit un ordre partiel complet sur $\Delta(T)$.*

MeTéo définit localement, pour chaque C-type, une opération de calcul/vérification de la relation de δ-sous-typage basée sur la forme syntaxique de l'expression des δ-types, en accord avec la sémantique ensembliste. En effet, le calcul effectif des liens de δ-sous-typage entre deux types dépend des champs qui expriment l'ensemble dénoté, selon les méthodes suivantes (la donnée formelle de ces règles est donnée en annexe B) :

- dans le cas du type **univers** et des **types de base**, l'inférence d'un lien de δ-sous-typage entre deux types t_1 et t_2 est effectuée en testant l'inclusion des champs *domaine* : soient $t_1 = \langle TU; D_1\rangle, t_2 = \langle TU; D_2\rangle$,

$$t_1 \leq^{\delta}_{TU} t_2 \Leftrightarrow \text{si } e \in D_1 \text{ alors } e \in D_2$$

- en ce qui concerne les **constructeurs**, le calcul doit prendre en compte les trois champs exprimant l'ensemble dénoté, selon le schéma suivant : soient $t_1 = \langle TC; [t'_1; D_1; C_1]\rangle$ et $t_2 = \langle TC; [t'_2; D_2; C_2]\rangle$, avec $t'_1 = \langle T; E_1(T)\rangle$ et $t'_2 = \langle T; E_2(T)\rangle$,

$$t_1 \leq^{\delta}_{TC} t_2 \Leftrightarrow \begin{cases} t'_1 \leq^{\delta}_{T} t'_2 \text{ et} \\ \text{si } e \in D_1 \text{ alors } e \in D_2 \text{ et} \\ \text{si } e \in C_2 \text{ alors } e \in C_1 \text{ ou } e \notin D_1 \text{ ou } e \notin t'_1 \end{cases}$$

  Dans le cas des **multivalués**, il s'agit de prendre en compte le champ portant sur la cardinalité, de telle façon que la cardinalité du δ-sous-type soit plus restreinte que celle du super-type. Ceci relève de l'arithmétique des intervalles de valeurs entières, puisqu'il s'agit de tester l'inclusion d'un intervalle dans un autre. Par ailleurs, la cardinalité entre aussi en jeu au niveau des conditions sur le domaine complémentaire.

- le δ-sous-typage dans le cas du type **énumération** est similaire à celui inféré dans le cas des constructeurs dont on retire le test sur le type de référence : soient $t_1 = \langle TE; [D_1; C_1]\rangle$ et $t_2 = \langle TE; [D_2; C_2]\rangle$,

$$t_1 \leq^{\delta}_{TE} t_2 \Leftrightarrow \begin{cases} \text{si } e \in D_1 \text{ alors } e \in D_2 \text{ et} \\ \text{si } e \in C_2 \text{ alors } e \in C_1 \text{ ou } e \notin D_1 \text{ ou } e \notin_{TC} t_1 \end{cases}$$

- le δ-sous-typage dans le cas des **types ordonnés** relève d'un seul champ, *domaine*, qui est une liste d'intervalles. Il s'agit donc de considérer l'arithmétique des intervalles pour tester l'inclusion d'ensembles exprimés par des intervalles.

Il est important de remarquer que l'inférence de liens de δ-sous-typage ne dépend finalement que des prédicats d'appartenance d'une valeur à un δ-type, et du prédicat d'égalité entre deux valeurs du même C-type : les deux suffisent à tester l'inclusion des domaines. C'est le prédicat d'appartenance d'une valeur à un δ-type qui prend effectivement en considération la forme syntaxique de l'expression des δ-types. Le théorème suivant établit la correction et la complétude du δ-sous-typage vis-à-vis de l'inclusion ensembliste des interprétations de types.

Théorème 1 *Pour tout C-type T, pour tous t_1, $t_2 \in \Delta(T)$,*

$$t_1 \leq^{\delta}_{T} t_2 \Longleftrightarrow \|t_1\|^{\xi} \subseteq \|t_2\|^{\xi}$$

3.4.4 Treillis de δ-types

L'ensemble $\Delta(T)$ des δ-types attachés à un C-type T forme un graphe ordonné par la relation de δ-sous-typage (définition 22).

Définition 22 (Graphe de δ-types) *Soit $T \in \mathcal{T}$ un C-type auquel est associé un ensemble non-vide de δ-types, noté $\Delta(T)$. Cet ensemble est partiellement ordonné par la relation de δ-sous-typage. On définit pour chaque C-type T le graphe $\mathcal{G}(T) = \langle T, D, A \rangle$, où :*

T est le C-type auquel est associé le graphe

D est l'ensemble des nœuds du graphe, $D = \Delta(T) \bigcup \{\bot\} \bigcup \{\top\}$, où $\bot$ est le δ-type dénotant l'ensemble vide, et $\top$ est le δ-type qui dénote le même ensemble que T dans $\mathcal{V}$.

A est l'ensemble des arêtes du graphe, $A = \{(t_1, t_2) | t_1, t_2 \in D, t_1 \leq^\delta_T t_2\}$. En particulier, $\forall t \in D, (\bot, t) \in A$ et $(t, \top) \in A$.

Le graphe des δ-types associé à un C-type *T, noté $\widehat{\mathcal{G}(T)}$, est déduit de $\mathcal{G}(T)$ par réduction transitive de l'ensemble des arêtes de $\mathcal{G}(T)$, noté $\widehat{A}$.*

La relation de δ-sous-typage étant un ordre partiel complet, nous en déduisons que le graphe des δ-types associé à un C-type est un *treillis complet* (propriété 2).

Propriété 2 *$\forall T \in \mathcal{T}, \widehat{\mathcal{G}(T)}$ est un treillis complet.*

Une conséquence importante de la propriété 2 est que le graphe des δ-types associé à un C-type peut être le support de la sémantique du plus petit (ou grand) point fixe, dans le cadre de la résolution de définitions récursives [Neb91] [AC91]. Cette possibilité est très importante à partir du moment où l'on considère des constructeurs de types : il est possible, du fait de la propriété de treillis complet, de donner une signification dénotationnelle au fait qu'un type puisse être construit à partir de lui même. La sémantique des points fixes s'appuie sur deux opérations usuelles sur les treillis, définies dans MeTéo (définitions 23 et 24).

Définition 23 (Plus petit majorant (SUP)) [5] *Soit $\sqcup$ l'opération qui détermine le plus petit majorant de deux δ-types dans un treillis $\widehat{\mathcal{G}(T)} = \langle T, N, \widehat{A} \rangle$.*

$$\forall t_1, t_2 \in N, \textit{ il existe un unique } t \in N \textit{ tel que } t = t_1 \sqcup t_2,$$
$$\textit{avec } \|t\|^\xi = \|t_1\|^\xi \bigcup \|t_2\|^\xi$$

Définition 24 (Plus grand minorant (INF)) [6] *Soit $\sqcap$ l'opération qui détermine le plus grand minorant de deux δ-types dans un treillis $\widehat{\mathcal{G}(T)} = \langle T, N, \widehat{A} \rangle$.*

$$\forall t_1, t_2 \in N, \textit{ il existe un unique } t \in N \textit{ tel que } t = t_1 \sqcap t_2,$$
$$\textit{avec } \|t\|^\xi = \|t_1\|^\xi \bigcap \|t_2\|^\xi$$

[5] *En anglais, LUB pour Least Upper Bound*
[6] *En anglais, GLB pour Great Lower Bound*

Ces opérations sont définies dans MÉTÉO pour chacune des catégories de C-types identifiées précédemment, c'est-à-dire pour chaque forme syntaxique d'expression des δ-types, selon le même principe que les inférences de δ-sous-typage vues précédemment.

De l'opération INF, MÉTÉO déduit, *au niveau de généricité maximum*, de nouvelles opérations classiques, telles que le test d'exclusivité (deux δ-types sont exclusifs si et seulement si leur INF dénote l'ensemble vide), ou encore la recherche du sous-treillis commun. Bien entendu, le INF est directement utilisable pour résoudre la récursivité dans les définitions de constructeurs.

3.4.5 Le γ-sous-typage

L'interprétation ensembliste qui définit les relations de δ-sous-typage et de C-sous-typage permet d'avoir la propriété suivante (propriété 3), qui montre qu'une correspondance peut être facilement établie entre les δ-types de deux C-types liés par le C-sous-typage :

Propriété 3 *Soient $T_1, T_2 \in \mathcal{T}$ tels que $T_2 \leq_C T_1$. Il existe un homomorphisme $h : \Delta(T_2) \rightarrow \Delta(T_1)$ tel que*

$$\forall t_1, t'_1 \in \Delta(T_1) \textit{ tels que } t_1 \leq^{\delta}_{T_1} t'_1,$$
$$h(t_1) \in \Delta(T_2), h(t'_1) \in \Delta(T_2) \textit{ et } h(t_1) \leq^{\delta}_{T_2} h(t'_1)$$

L'homomorphisme h a en fait pour effet de représenter, avec le mode de représentation de T_2, l'ensemble de valeurs de t auquel les valeurs n'appartenant pas à T_2 sont enlevées. Par exemple, si `Pair` $\leq_C$ `Entier`, et si $t = \langle Entier; [11; 32] + [51; 90] \rangle$ alors $h(t) = \langle Pair; [12; 32] + [52; 90] \rangle$. Un tel homomorphisme ne peut toutefois pas garantir que, si deux δ-types sont en relation stricte de δ-sous-typage dans T_1, ils le seront aussi sous T_2 : au pire ce lien de δ-sous-typage deviendra une égalité (puisqu'il y a perte d'éléments lors de l'homomorphisme). Par exemple, $t_1 = \langle Entier; [2; 9] \rangle$ et $t_2 = \langle Entier; [2; 8] \rangle$; on a trivialement $t_2 <^{\delta}_{Entier} t_1$ et $h(t_1) = \langle Pair; [2; 8] \rangle = h(t_2)$.

Au niveau des treillis de δ-types, la propriété 3 se traduit par un plongement du treillis de T_2 dans celui de T_1.

Cette propriété est intéressante car elle montre que tous les δ-types d'un C-type T peuvent être comparés aux δ-types d'un C-super-type de T, et vice-versa. Ainsi, la relation de sous-typage ne se limite plus à des liens entre δ-types d'un même C-type, ou entre C-types, mais elle s'étend à des comparaisons sensées de δ-types issus de C-types différents. La combinaison du C-sous-typage et du δ-sous-typage définit le γ-sous-typage (définition 25).

Définition 25 (γ-sous-typage) *Le γ-sous-typage est une relation qui lie deux δ-types qui ne sont pas issus du même C-type, par combinaison des relations de C-sous-typage et de δ-sous-typage. Soient $t_1 = \langle T_1; E_1(T_1) \rangle$ et $t_2 = \langle T_2; E_2(T_2) \rangle$ tels que $T_2 \leq_C T_1$. Un lien de γ-sous-typage entre t_2 et t_1 est noté $t_2 \leq_\gamma t_1$ et se définit comme suit :*

$$t_2 \leq_\gamma t_1 \Leftrightarrow \begin{cases} T_1 = T_2 = T \textit{ et } t_2 \leq^{\delta}_{T} t_1 \\ \textit{ou } T_2 <_C T_1 \textit{ et } t_2 \leq^{\delta}_{T_2} h(t_1) \end{cases}$$

Où h est l'homomorphisme introduit dans la propriété 3.

Si le C-sous-typage et le δ-sous-typage sont maintenus dans MÉTÉO, le γ-sous-typage, quant à lui, n'est pas explicitement représenté, mais il est vérifié/calculé sur sollicitation.

3.4.6 Nouvelle définition des C-types

Nous résumons ici la définition complète d'un C-type, et c'est désormais cette définition à laquelle nous ferons référence dans la suite de notre étude.

Nous avons vu dans la section 3.3.2 que les C-types sont l'implémentation de structures de données : chacun représente un ensemble de valeurs, dont il fournit les constructeurs. Nous avons vu que le C-type définit en outre des opérations sur ses valeurs, qui représentent ensemble le *comportement* des valeurs du C-type.

À partir du moment où les δ-types sont introduits, il s'agit pour chaque C-type de définir la syntaxe pour l'expression de ses δ-types, selon certains critères organisationnels issus des propriétés du C-type. Toutes les opérations relatives au δ-sous-typage et celles concernant la manipulation des treillis étant tributaires des formes syntaxiques des δ-types, propres à chaque C-type, les C-types doivent ainsi définir les opérations de manipulation de ces expressions syntaxiques, en accord avec leur interprétation ensembliste. Pour cela, la définition 26 étend la définition des C-types qui est donnée en section 3.3.2 :

Définition 26 (C-type) *Un C-type T est la donnée d'un sextuplet $\langle T', C, F, P, S, M\rangle$ où :*

- *T', C, F et P sont ceux introduits dans la définition 16 (C-super-type, constructeurs, opérations sur les valeurs, prédicats),*
- *S indique les champs nécessaires à l'expression d'un δ-type de T,*
- *M est l'ensemble des opérations de manipulation des δ-types, c'est-à-dire les opérations qui dépendent de la forme syntaxique des δ-types (donnée par S), et dont la sémantique est celle dictée par l'interprétation ensembliste. Parmi ces opérations, cinq sont essentielles, à savoir :*
  - l'appartenance d'une valeur à un δ-type, *notée $v \in^{\delta}_{T} \delta t$, qui est vrai si la valeur v appartient au δ-type δt issu du C-type T,*
  - le δ-sous-typage, *noté $\delta t_1 \leq^{\delta}_{T} \delta t_2$, qui détermine la validité d'un lien de δ-sous-typage entre deux δ-types de T,*
  - le INF, *noté $\delta t_1 \sqcap_T \delta t_2$, qui détermine le plus grand minorant de deux δ-types issus de T,*
  - le SUP, *noté $\delta t_1 \sqcup_T \delta t_2$, qui détermine le plus petit majorant de deux δ-types issus de T,*
  - l'élimination, *notée $\delta t_1 \backslash_T \delta t_2$, qui a pour résultat le δ-type dont le domaine est celui de δt_1 auquel ont été enlevées toutes les valeurs du domaine de δt_2.*

La mise en œuvre des cinq opérations principales sur les δ-types d'un C-type nécessite la définition de quelques primitives sur les champs de description des δ-types. Il s'agit principalement du prédicat d'appartenance d'une valeur à une liste (ou intervalle) de valeurs du même type T (noté $member_T$), du test d'inclusion entre deux listes (intervalles) de valeurs du même type T (noté $include_T(D_1, D_2)$), de l'union et intersection de deux listes (intervalles) de valeurs du même type (notées $union_T(D_1; D_2)$ et $inter_T(D_1; D_2)$), de l'élimination des valeurs d'un premier domaine de l'ensemble des valeurs du second ($remove_T(D_1; D_2)$), et de quelques autres opérations portant sur des listes (intervalles) de valeurs d'un même C-type. Ces opérations peuvent se définir de façon générique car elles ne sont spécifiques à chaque C-type que du fait de la spécificité du prédicat d'égalité entre deux valeurs d'un C-type, à la base de la réalisation de telles opérations. La généricité de ces fonctions est une illustration du polymorphisme d'inclusion de MÉTÉO.

Il est important de remarquer que certaines fonctions de F sont nécessaires pour certains C-types, car elles peuvent être utilisées pour l'expression des δ-types dans une syntaxe particulière. Par exemple, le C-type `Pair`, pour être un C-type `Discret` et pour ainsi représenter ses sous-ensembles par des intervalles à bornes fermées, doit définir *obligatoirement* les opérations `successeur` et `prédécesseur`. Cela permet de définir, au niveau de C-type `Discret`, toutes les opérations de manipulation des δ-types issus des C-sous-types de `Discret` : ces opérations sont définies à partir des opérations exigées pour ces C-sous-types. Par exemple, l'opération d'élimination dépend, dans le cas des C-types de la catégorie des `Discret`, du mode de représentation des intervalles, défini au niveau de `Discret`, ce mode dépendant à son tour des prédicats d'ordre sur les valeurs, d'appartenance d'une valeur à un δ-type, d'égalité de deux valeurs, et des opérations `successeur` et `prédécesseur`. En résumé, pour $\mathtt{X} \leq_C \mathtt{Discret}$, $\langle\ \mathtt{X};E_1(\mathtt{X})\ \rangle \backslash_{\mathtt{X}} \langle\ \mathtt{X};E_2(\mathtt{X})\ \rangle =$

$$F(\leq^{\delta}_{\mathtt{X}}, =_{\mathtt{X}}, \in^{\delta}_{\mathtt{X}}, \mathtt{Succ}_{\mathtt{X}}, \mathtt{Pred}_{\mathtt{X}})$$

Ainsi, l'opération d'élimination n'a à être définie qu'au niveau du C-type maximum d'une catégorie de C-types correspondant à un mode d'expression des δ-types particulier. Cela illustre une fois de plus le polymorphisme de METÉO.

Enfin, il a été précisé que les expressions EOLE de δ-types sont des formes normalisées, à savoir que deux expressions différentes issues d'un même C-type dénotent dans $\mathcal{V}$ des ensembles de valeurs différents. Pour cela, METÉO dispose, pour chaque C-type, d'une procédure de *normalisation* qui, à partir d'une description initiale d'un δ-type, en calcule la forme normalisée. Cette procédure associée à un C-type T est notée $NormalForm_T$, nous y reviendrons plus précisément dans le chapitre suivant.

Exemple du C-type `Record`

La section 3.3.2 montre la définition du C-type `Record` comme l'implémentation d'un type abstrait de données. Autrement dit, cette définition s'attache à la spécification des valeurs réunies par ce C-type.

Il s'agit de compléter la définition du C-type `Record` afin d'y ajouter les opérations qui manipulent des sous-ensembles de valeurs *records*, communément appelés *types records* [CM91]. Un type *record*, d'après la définition de Luca Cardelli, regroupe un ensemble de valeurs *records* en restreignant le domaine de valeurs de certaines étiquettes. Par exemple, l'expression $t = \langle\langle e_1 : entier, e_2 : booleen, e_3 : chaine\rangle\rangle$ est le type *record* dont les valeurs ont au moins les étiquettes e_1, e_2 et e_3, les valeurs respectives desquelles sont de types *entier*, *booleen* et *chaine*.

De même que les valeurs *records*, un type *record* peut être construit au moyen de deux opérateurs :

$\langle\langle\ \rangle\rangle$ le type *record* qui ne restreint aucune étiquette, et qui en conséquence regroupe toutes les valeurs *records* (il dénote dans $\mathcal{V}$ le même ensemble que celui dénoté par le C-type `Record`)

extension $\langle\langle R|e : t\rangle\rangle$ ajoute l'étiquette e restreinte au type t, à condition que cette étiquette n'existe pas déjà dans R.

Un type *record* est donc une collection de types étiquetés, et c'est sur cette base que le C-type `Record` définit la syntaxe de représentation de ses δ-types. D'après la syntaxe de EOLE présentée dans la table 3.1, un δ-type δt du C-type `Record` s'exprime par l'intermédiaire de trois champs : TR, qui contient les δ-types sur lesquels est construit δt, DE qui contient éventuellement une liste exhaustive de valeurs *records* et DEC (noté aussi DE dans la syntaxe d'EOLE puisqu'il s'agit d'un domaine de valeurs), qui contient une liste de valeurs interdites. Ainsi, un δ-type du C-type `Record` est un type *record* (exprimé par TR) auquel il est possible de retirer explicitement certaines valeurs, ou bien qui peut être explicitement substitué par la liste exhaustive des valeurs *records* constituant le δ-type[7] (figure 3.7).

$$\begin{array}{l} \delta t_1 = \langle\ Record; \\ \qquad [\ \langle\langle\ age : \langle Entier; [18; 65]\rangle, sexe : \langle Booleen; [0; 1]\rangle\ \rangle\rangle; \\ \qquad\ all; \\ \qquad\ (\ \langle age = 20, sexe = 1\rangle, \langle age = 25, sexe = 0\rangle\ )\ ]\rangle \end{array}$$

FIG. 3.7 - : δt_1 est un exemple d'expression de δ-type en EOLE qui regroupe l'ensemble des valeurs *records* qui ont au moins les champs age et $sexe$, le premier prenant ses valeurs dans les entiers entre 18 et 65, le second parmi tous les booléens, à l'exception des valeurs ayant 20 pour age et 1 pour $sexe$, ainsi que les valeurs ayant 25 pour age et 0 pour $sexe$.

À partir de ce choix de représentation des δ-types issus du C-type `Record`, les quatre opérations principales que doit définir `Record` sur ses δ-types sont immédiates :

Appartenance L'appartenance d'une valeur *record* r à un δ-type δt issu de `Record` est notée $r \in^{\delta}_{Record} \delta t$ et se définit comme suit :

- $\forall \delta t \in \Delta(Record), \langle\ \rangle \in^{\delta}_{Record} \delta t$
- $\forall v$ telle que $\in_{Record} (v), v \in^{\delta}_{Record} \langle Record; [\langle\langle\ \rangle\rangle; all; nothing]\rangle$
- $r = \langle r' | e = v\rangle \in^{\delta}_{Record} \langle Record; [R; D_1; D_2]\rangle$, avec $R = \langle\langle R' | e : t\rangle\rangle$ et $t = \langle T; E(T)\rangle$

$$\Leftrightarrow v \in^{\delta}_{T} t \text{ et } \left\{ \begin{array}{c} member_{Record}(r, D_1) \\ et \\ \neg member_{Record}(r, D_2) \\ et \\ r' \in^{\delta}_{Record} \langle Record; [R'; all; nothing]\rangle \end{array} \right.$$

δ-sous-typage Le δ-sous-typage entre deux δ-types issus de `Record` δt_1 et δt_2 est noté $\delta t_1 \leq^{\delta}_{Record} \delta t_2$ et se définit inductivement comme suit :

- $\forall \delta t \in \Delta(Record), \delta t \leq^{\delta}_{Record} \langle Record; [\langle\langle\ \rangle\rangle; all; nothing]\rangle$
- $\forall \delta t \in \Delta(Record), \perp \leq^{\delta}_{Record} \delta t$
- $\delta t_1 = \langle Record; [\langle\langle R_1 | e : t_1\rangle\rangle; D_1; D'_1]\rangle, \delta t_2 = \langle Record; [\langle\langle R_2 | e : t_2\rangle\rangle; D_2; D'_2]\rangle, \delta t_1 \leq^{\delta}_{Record} \delta t_2$

$$\Leftrightarrow t_1 \leq \gamma\ t_2 \text{ et } \left\{ \begin{array}{c} \forall r \text{ telle que } member_{Record}(r, D_1), r \in^{\delta}_{Record} \delta t_2 \\ et \forall r \text{ telle que } member_{Record}(r, D'_2), r \notin^{\delta}_{Record} \delta t_1 \\ et\ \langle Record; [R_1; all; nothing]\rangle \leq^{\delta}_{Record} \langle Record; [R_2; all; nothing]\rangle \end{array} \right.$$

[7] La nécessité des deux champs DE et DEC provient du fait que tout ensemble de valeurs *records* ne peut pas s'exprimer qu'avec la donnée seule d'un type *record*, car on peut vouloir exprimer certains critères de regroupement inter-étiquettes. La solution proposée ici n'est pas non plus optimale car ne permet pas l'expression de ces contraintes, mais présente toutefois l'avantage de permettre la représentation en extension du résultat d'une telle pose de contrainte, dans la mesure où cette extension est finie.

INF L'opération de calcul du plus grand minorant entre deux δ-types δt_1 et δt_2 issus de `Record` est notée $\delta t_1 \sqcap_{Record} \delta t_2$ et se définit comme suit :

- $\forall \delta t \in \Delta(Record), \delta t \sqcap_{Record} \langle Record; [\langle\langle\ \rangle\rangle; all; nothing]\rangle = \delta t$
- $\forall \delta t \in \Delta(Record), \delta t \sqcap_{Record} \perp = \perp$
- $\delta t_1 = \langle Record; [R_1; D_1; D'_1]\rangle$ et $\delta t_2 = \langle Record; [R_2; D_2; D'_2]\rangle$.

$$\delta t_1 \sqcap_{Record} \delta t_2 = NormalForm_{Record}(\langle Record; [R_3; D_3; D'_3]\rangle) \text{ avec :}$$

  - $R_3 = R_1 \bigtriangledown R_2$ où $\bigtriangledown$ est l'opération de conjonction de types *records* définie par :

$$\begin{cases} \forall R = \langle\langle R' \mid e : t\rangle\rangle, R \bigtriangledown \langle\langle\ \rangle\rangle = R \\ \forall R = \langle\langle R' \mid e : t\rangle\rangle, R \bigtriangledown \perp = \perp \\ R_1 = \langle\langle R'_1 \mid e : t_1\rangle\rangle \bigtriangledown R_2 = \langle\langle R'_2 \mid e : t_2\rangle\rangle = \langle\langle R'_1 \bigtriangledown R'_2 \mid e : t_1 \sqcap_T t_2\rangle\rangle \\ \quad \text{si } t_1 = \langle T_1; E(T_1)\rangle \text{ et } t_2 = \langle T_2; E(T_2)\rangle \\ \quad \text{et } T = Max_C(T_1, T_2) \text{ est le plus grand des deux C-types} \end{cases}$$

  - $D_3 = inter_{Record}(D_1, D_2)$
  - $D'_3 = inter_{Record}(D'_1, D'_2)$

SUP L'opération de calcul du plus petit majorant entre deux δ-types δt_1 et δt_2 issus de `Record` est notée $\delta t_1 \sqcup_{Record} \delta t_2$ et se définit comme suit :

- $\forall \delta t \in \Delta(Record), \delta t \sqcup_{Record} \langle Record; [\langle\langle\ \rangle\rangle; all; nothing]\rangle = \langle Record; [\langle\langle\ \rangle\rangle; all; nothing]\rangle$
- $\forall \delta t \in \Delta(Record), \delta t \sqcup_{Record} \perp = \delta t$
- $\delta t_1 = \langle Record; [R_1; D_1; D'_1]\rangle$ et $\delta t_2 = \langle Record; [R_2; D_2; D'_2]\rangle$.

$$\delta t_1 \sqcap_{Record} \delta t_2 = NormalForm_{Record}(\langle Record; [R_3; D_3; D'_3]\rangle) \text{ avec :}$$

  - $R_3 = R_1 \bigtriangleup R_2$ où $\bigtriangleup$ est l'opération de disjonction de types *records* définie par :

$$\begin{cases} \forall R = \langle\langle R' \mid e : t\rangle\rangle, R \bigtriangleup \langle\langle\ \rangle\rangle = \langle\langle\ \rangle\rangle \\ \forall R = \langle\langle R' \mid e : t\rangle\rangle, R \bigtriangleup \perp = R \\ R_1 = \langle\langle R'_1 \mid e : t_1\rangle\rangle \bigtriangleup R_2 = \langle\langle R'_2 \mid e : t_2\rangle\rangle = \langle\langle R'_1 \bigtriangleup R'_2 \mid e : (t_1 \sqcup_T t_2)\rangle\rangle \\ \quad \text{si } t_1 = \langle T_1; E(T_1)\rangle \text{ et } t_2 = \langle T_2; E(T_2)\rangle \\ \quad \text{et } T = Max_C(T_1, T_2) \text{ est le plus grand des deux C-types} \end{cases}$$

  - $D_3 = union_{Record}(D_1, D_2)$
  - $D'_3 = union_{Record}(D'_1, D'_2)$

Élimination L'opération d'élimination appliquée aux δ-types *records* se définit champ par champ, de la façon suivante :

- $\forall \delta t \in \Delta(Record), \delta t \backslash_{Record} \perp = \delta t$
- $\forall \delta t \in \Delta(Record), \perp \backslash_{Record} \delta t = \perp$
- $\delta t_1 = \langle Record; [R_1; D_1; D'_1]\rangle$ et $\delta t_2 = \langle Record; [R_2; D_2; D'_2]\rangle$.

$$\delta t_1 \backslash_{Record} \delta t_2 = NormalForm_{Record}(\langle Record; [R_3; D_3; D'_3]\rangle) \text{ avec :}$$

  - $R_3 = R_1 \wr R_2$ où $\wr$ est l'opération sur les *records* qui, pour chaque étiquette de R_1 enlève de son domaine associé le domaine associé à la même étiquette dans R_2 :

$$\begin{cases} \forall R, \langle\langle\ \rangle\rangle \wr R = \langle\langle\ \rangle\rangle \\ R_1 \wr R_2 = \langle\langle R'_1 \wr R_2 \mid e : t_1\rangle\rangle \\ \quad \text{si } R_1 = \langle\langle R'_1 \mid e : t_1\rangle\rangle \text{ et } \nexists (e : t_2) \text{ tq } R_2 = \langle\langle R'_2 \mid e : t_2\rangle\rangle \\ R_1 = \langle\langle R'_1 \mid e : t_1\rangle\rangle \wr R_2 = \langle\langle R'_2 \mid e : t_2\rangle\rangle = \langle\langle R'_1 \wr R'_2 \mid e : (t_1 \backslash_T t_2)\rangle\rangle \\ \quad \text{si } t_1 = \langle T_1; E(T_1)\rangle \text{ et } t_2 = \langle T_2; E(T_2)\rangle \\ \quad \text{et } T = Max_C(T_1, T_2) \text{ est le plus grand des deux C-types} \end{cases}$$

- $D_3 = remove_{Record}(D_2, D_1)$
- $D'_3 = remove_{Record}(D'_2, D'_1)$

Ces définitions d'opérations sont en accord avec la sémantique qui leur a été donnée, car les δ-types sont des expressions normalisées.

Nous définissons maintenant une opération propre au C-type `Record`, qui est la *projection*. Elle consiste simplement à extraire d'un δ-type issu de `Record` le δ-type correspondant à la projection du champ **type-ref** le long d'un ensemble donné d'étiquettes. La **projection** d'un δ-type δt selon l'ensemble d'étiquettes $\{e_i\}_{i\in[1;p]}$ a comme résultat un δ-type $\delta t'$ dont le champ **type-ref** est réduit (les éléments constituant le type *record* qui ne sont pas étiquetés par une des étiquettes de l'ensemble de projection, disparaissent, et les valeurs des champs **domaine** et **comp-dom** sont réduites selon le même schéma).

Projection l'opération de projection est définie comme suit :

- $\forall E, \prod_E(\perp) = \perp$
- $\forall \delta t = \langle Record; E(Record)\rangle, \prod_{\emptyset}(\delta t) = \delta t$
- Soit $E = \{e_i\}_{i\in[1;p]}$, soit $\delta t = \langle Record; [R; D_1; D_2]\rangle$, avec $R = \langle\langle e_1 : t_1 \times \cdots \times e_n : t_n\rangle\rangle$, alors
$$\prod_E(\delta t) = NormalForm_{Record}(\langle Record; [R'; D'_1; D'_2]\rangle), \text{ avec :}$$
  - $R' = \langle\langle e_{p+1} : t_{p+1} \times \cdots \times e_n : t_n\rangle\rangle$
  - $D'_1 = \begin{cases} \text{si } D_1 = \text{TOUT alors TOUT} \\ \text{si } D_1 = v_1, \cdots, v_m \text{ alors } v'_1, \cdots, v'_m \\ \quad \text{avec } \forall j \in [1; m], v'_j = v_j \backslash \{e_j\}_{j\in[1;p]} \\ \quad (\backslash \text{ est l'opération de restriction définie page 99}) \end{cases}$
  - $D'_2 = \begin{cases} \text{si } D_2 = \emptyset \text{ alors } \emptyset \\ \text{si } D_2 = v_1, \cdots, v_m \text{ alors } v'_1, \cdots, v'_m \\ \quad \text{avec } \forall j \in [1; m], v'_j = v_j \backslash \{e_j\}_{j\in[1;p]} \end{cases}$

Cette nouvelle définition du C-type `Record` est largement inspirée des travaux menés par L. Cardelli concernant les types *records*. En particulier, les résultats théoriques établis sur les *records* sont pris en considération dans METÉO, et notamment le fait que leur sémantique soit fondée sur le modèle de treillis des idéaux.

Toutefois, METÉO ne conserve qu'une petite partie des caractéristiques des *records*. En effet, les *records* sont en réalité la formulation de la spécification abstraite d'un module dans un langage de programmation : d'un point de vue des langages de programmation par objets, certaines des étiquettes de *records* sont réservées à la spécification des attributs, d'autres étiquettes servent à la spécification des méthodes. La notion de *méthode* n'étant pas présente dans les modèles de connaissances à objets que nous considérons, tous les travaux relatifs aux champs réservés aux méthodes n'ont pas été pris en compte dans METÉO, en particulier en ce qui concerne le sous-typage.

3.4.7 Conclusion

Un δ-type est défini de par un C-type comme étant une partition de ce C-type. De ce fait, un C-type est plus que la définition d'une structure de données et des opérations sur ces données, il contient aussi la définition du mode de représentation de ses δ-types ainsi que les opérations de manipulation de cette représentation, en accord avec la sémantique dénotationnelle du système.

Les δ-types issus d'un même C-type sont organisés en treillis ordonné par la relation de δ-sous-typage. Cette relation est définie localement dans un C-type, d'après la syntaxe de représentation des δ-types, mais la sémantique de cette relation est globale (elle correspond à l'inclusion ensembliste), ce qui permet, à un niveau de spécification élevé, une exploitation homogène et cohérente du δ-sous-typage. Une définition générique du δ-sous-typage permet d'ailleurs la définition générique du γ-sous-typage, par composition du C-sous-typage et du δ-sous-typage.

La différence majeure qui existe entre les notions de C-types et de δ-type est liée au comportement. En effet, le regroupement de valeurs caractérisé par un C-type puise tout son sens dans la nature des opérations applicables sur ces valeurs. C'est effectivement selon ce critère que le type associé à une variable est délibérement choisi, quel que soit le contexte du typage. À l'opposé, le regroupement des valeurs dans un δ-type ne répond à aucun critère absolu et objectif, il n'est pas guidé par les opérations sur ces valeurs, ce regroupement est parfaitement arbitraire si l'on ne considère pas le contexte de typage (à condition, bien sûr, que le type principal duquel est issu le δ-type ait été précédemment choisi, c'est-à-dire le comportement des valeurs déjà identifié).

3.5 Module de contrôle de Metéo

Le module de contrôle de Metéo a pour rôle la centralisation des opérations définies localement dans les C-types. Certaines de ces opérations ont une sémantique commune, d'un C-type à un autre, comme la relation de δ-sous-typage, le test d'appartenance d'une valeur à un δ-type ou la recherche du minimum de deux valeurs quelconques. Cette sémantique est alors reflétée au niveau du module de contrôle, qui lui associe une procédure globale paramétrée. Metéo définit ainsi globalement un ensemble de mécanismes, qui sont partagés en deux groupes : les mécanismes spécifiques, qui correspondent à la centralisation des opérations sur les δ-types, et les mécanismes globaux qui résultent d'une synthèse entre les opérations sur les δ-types et celles sur les C-types.

3.5.1 Mécanismes spécifiques

Ils concernent la représentation à un niveau global de la sémantique des opérations sur les δ-types, qui sont définies pour tout C-type mais dont la réalisation dépend du mode de représentation des C-types.

Parmi les mécanismes spécifiques du module de contrôle, les huit immédiats sont :

1. la création d'un δ-type, `mgs-create-dtype(T,`$\langle champ_i :< info >_i\rangle$`)` $\mapsto \delta t$, le second paramètre dépend du C-type considéré,
2. l'opération de INF, `mgs-INF(T,`δt_1`, `δt_2`)` $\mapsto \delta t_3$,
3. l'opération de SUP, `mgs-INF(T,`δt_1`, `δt_2`)` $\mapsto \delta t_3$,
4. l'élimination, `mgs-elimine(T,`δt_1`, `δt_2`)` $\mapsto \delta t_3$,
5. le test de l'appartenance d'une valeur à un C-type, `mgs-member-Ctype(T,`v`)` $\mapsto$ `Booléen`,

6. le test de l'appartenance d'une valeur à un δ-type, `mgs-member-dtype(T,`δt`, `v`)` $\mapsto$ `Booléen`; il existe en réalité deux opérations, l'une testant l'appartenance stricte, l'autre testant l'appartenance large,
7. le test d'égalité entre deux valeurs, `mgs-equal-values(T,`v_1`,`v_2`)` $\mapsto$ `Booléen`,
8. le test de γ-sous-typage, `mgs-subtyping(T, `δt_1`, `δt_2`)` $\mapsto$ `Booléen`.

Au niveau du module de contrôle, ces opérations sont paramétrées par le C-type, leur réalisation consiste en partie à identifier le paramètre et à activer l'opération locale correspondant à ce paramètre.

Mécanismes calculatoires

On remarque le statut particulier de l'opération de création d'un δ-type, qui est, entre autres, appelée par les opérations de INF, de SUP et d'élimination. En effet, ces trois dernières ne modifient pas un δ-type donné, mais en créent un nouveau à partir de ceux en entrée et du rôle de l'opération. Et c'est au niveau de l'opération de création de δ-type qu'est effectuée la normalisation : la création d'un δ-type consiste en premier lieu à initialiser les champs de représentation, puis en second lieu à activer la normalisation sur ces champs :

`mgs-create-dtype(T,`$\langle champ_i :< info >_i\rangle$`)` :
    `normal-form(T,initialize(T,`$\langle champ_i :< info >_i\rangle$`))`

Où `normal-form` et `initialize` sont des fonctions définies localement pour chaque C-type. La seconde est triviale, la première est décrite, pour chaque catégorie de C-type, en annexe A : elle fait en particulier appel aux fonctions de manipulation des champs, telles que $remove_T$ ou $inter_T$, définies génériquement pour chaque catégorie de C-type, grâce à une paramétrisation constante du C-type.

Cette définition de l'opération de création d'un δ-type garantit que seules des expressions normalisées seront présentes et manipulées dans MÉTÉO.

Mécanismes de test

En ce qui concerne les quatre derniers mécanismes, ils correspondent simplement à la centralisation des opérations locales, avec une pré-sélection de l'opération locale à considérer.

Par exemple, dans le cas du test de γ-sous-typage, c'est l'opération correspondant au plus petit des C-types de chaque δ-type en entrée, qui sera activée.

3.5.2 Mécanismes globaux

Les mécanismes globaux sont des processus qui réalisent la combinaison entre les opérations sur les C-types et les opérations de même sémantique sur les δ-types.

Il est évident que seuls les δ-types explicitement créés sont représentés dans MÉTÉO, et par conséquent les treillis de δ-types ne sont pas complets, ils évoluent au gré des actions de typage. Ainsi, à partir de ces opérations propres aux δ-types qu'il centralise, le module de contrôle définit des mécanismes plus évolués qui concernent la gestion des treillis de δ-types: insertion / suppression d'un δ-type dans son treillis, ou détermination du sous-treillis commun à deux δ-types, extraction d'un sous-treillis, etc., selon des algorithmes classiques développés en annexe. Nous commentons ces algorithmes dans les chapitres 4 et 5, c'est-à-dire dans le contexte de leur activation.

En outre, nous verrons dans le chapitre suivant la réalisation de l'opération globale $\gg_\tau$ destinée à représenter l'interprétation intensionnelle des passerelles, ainsi que des opérations destinées à représenter les conditions intensionnelles des propriétés d'exclusivité et d'exhaustivité des taxonomies de classes.

3.5.3 Architecture générale

Nous représentons sur la figure 3.8 l'architecture logicielle de METÉO, en illustrant les trois niveaux de définitions d'opérations : mécanismes globaux, mécanismes spécifiques, et opérations locales.

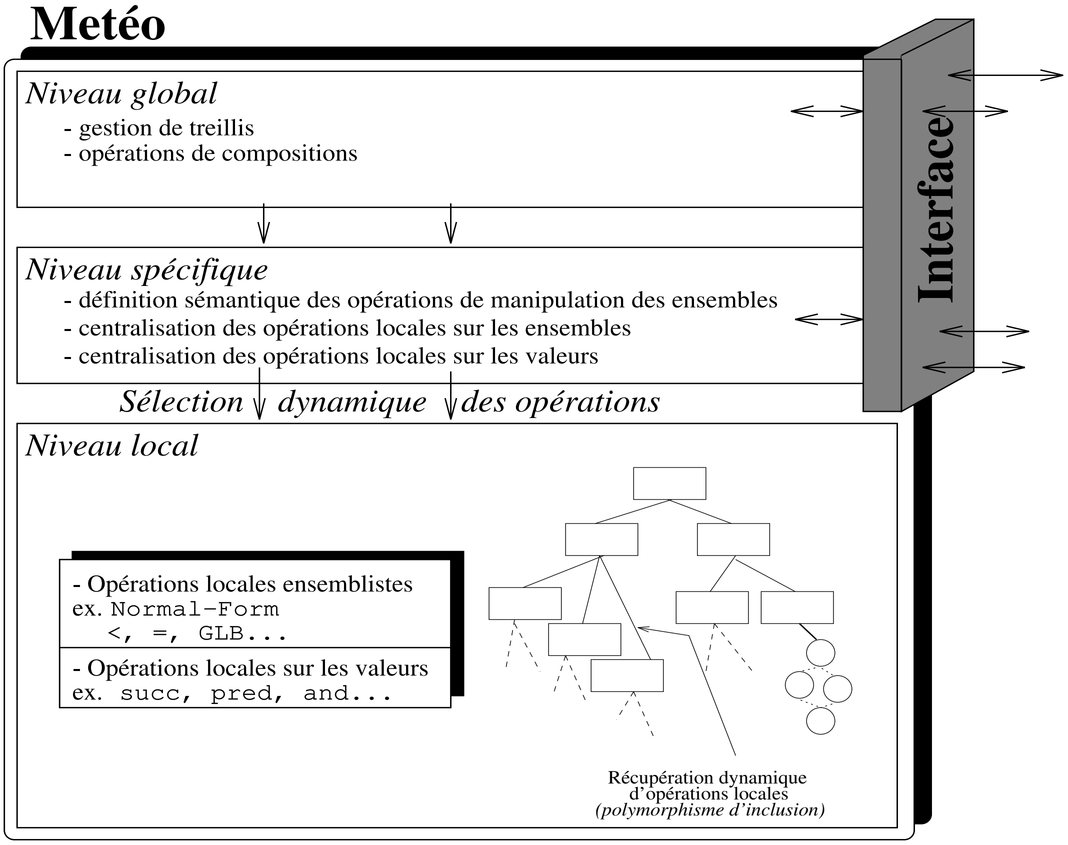


FIG. 3.8 - : Architecture logicielle de METÉO.

- *Le niveau global* contient des opérations génériques de manipulation des opérations définies sur les ensembles de valeurs dénotés par les C-types et δ-types (gestion de treillis, réalisation d'opérations de composition d'opérations spécifiques, etc.)
- *Le niveau spécifique* concerne l'attribution de la sémantique des opérations de manipulation directe des ensembles de valeurs dénotés par les C-types et δ-types. On peut associer une sémantique dénotationnelle à ces opérations spécifiques, indépendamment de la représentation syntaxique des sous-ensembles de valeurs.

– *Le niveau local* définit un ensemble d'opérations reprenant la sémantique des opérations du niveau supérieur, en tenant compte cette fois de la syntaxe de représentation des ensembles de valeurs.

Seuls les niveaux global et spécifique seront accessibles par l'interface entre METÉO et TROPES, le modèle de connaissances auquel METÉO est dédié.

3.6 Extensibilité de Metéo

La possibilité d'ajouter de nouveaux types de données, et ce de façon homogène, est une fonctionnalité nécessaire pour un système de types dédié à un langage dont les applications sont à la fois complexes et évolutives. Nous avons d'ailleurs abordé, dans la section 2.4.2, l'intérêt, dans un modèle de représentation des connaissances, de permettre la définition de nouvelles structures de données et de leurs opérations associées, indépendamment des connaissances représentées en termes du langage de représentation. Une telle possibilité, au sein d'un système de types dédié à un langage de représentation, a été prônée suite à deux observations faites sur les langages de représentation actuels.

– La plupart des langages de représentation (terminologiques, de *frames*, hybrides...), utilisent directement, à des fins de vérifications de cohérence, le système de types du langage hôte, rendant ainsi dépendante de ce langage l'extension du nombre de types de données disponibles pour les applications. On relèvera toutefois le cas particulier du système K-REP [MDW91] qui a défini son propre système de types, auquel il manque encore cette propriété d'extensibilité.

– Par ailleurs (et pour ne pas dire “en conséquence”), les langages de représentation étant confrontés aux exigences de leurs applications en matière de complexité de structures, ils se retrouvent face à la tentation d'adapter leur pouvoir expressif en vue de la programmation de telles structures. En particulier, les langages munis d'une sémantique dénotationnelle très adaptée à la déclarativité des représentations aimeraient pouvoir adapter (et complexifier inutilement) leur définition afin d'autoriser le développement de véritables types abstraits. Et ce n'est pas leur vocation.

METÉO a ainsi été conçu en partie pour répondre à cette nécessité de faciliter l'expression de types de données, qui se comportent, une fois définis, comme n'importe quel type de base, tout au moins au regard du modèle de connaissances. METÉO est en effet un système *extensible*. Il autorise et facilite, du fait de son organisation hiérarchique et modulaire, la création de nouveaux types de données ayant le statut de C-types.

3.6.1 Spécification d'un nouveau C-type

L'extensibilité de METÉO est réalisée par l'extensibilité homogène de sa hiérarchie de C-types. En effet, contrairement aux δ-types, les C-types permettent de spécifier, et leurs ensembles de valeurs, et les opérations applicables sur ces valeurs. C'est justement cet aspect comportemental qu'il est intéressant de considérer lorsque l'on parle d'ajouts de structures de données, dans la mesure où ces structures sont justement nécessaires du fait des opérations qu'elles supportent.

À l'instar des possibilités offertes par la théorie des algèbres multi-sortes ordonnées, la hiérarchie des C-types de METÉO, qui peut être considérée, du point de vue spécification, comme une hiérarchie de types abstraits de données, est un support à la création incrémentale de nouveaux types de données, c'est-à-dire qu'ils se définissent par référence à d'autres types existants.

La section 3.4.6 a présenté la définition minimale d'un C-type. Cette définition comporte deux parties, certes distinctes, mais non indépendantes. D'une part, le C-type s'attache à définir tout ce qui concerne ses valeurs, à savoir la construction de ces valeurs, d'après leurs propriétés, ainsi que les opérations applicables sur ces valeurs. D'autre part, le C-type doit fournir un mode de représentation de sous-ensembles de ses valeurs, la syntaxe et la sémantique d'une telle représentation dépendant justement des propriétés de ces valeurs.

La définition incrémentale d'un nouveau C-type T commence par attribuer à T un C-sur-type direct, T'. T "hérite" alors de T' les propriétés de ses valeurs, les prédicats et opérations sur ces valeurs, ainsi que le mode de représentation de ses δ-types. La distinction minimale à signifier entre T, le nouveau C-type, et T', son C-sur-type, réside dans la définition des prédicats $\in_T (v)$ et $=_T (v, v')$, et pour ce faire, les opérateurs de construction doivent éventuellement être redéfinis. Ensuite, deux cas se présentent : le nouveau C-type introduit, ou non, un nouveau mode de représentation de ses δ-types.

- Dans le premier cas, l'utilisateur doit donc fournir toutes les constantes et opérations qui définissent d'une part la structure des δ-types, et d'autre part, les mécanismes de construction et de manipulation de ces δ-types. Parmi ce qu'il y a à définir dans ce cas, on relève, bien entendu, la syntaxe de représentation d'un δ-type (ainsi que le cas particulier du δ-type maximum), mais aussi et surtout les opérations de δ-sous-typage, de SUP, INF, élimination, et l'appartenance d'une valeur à un δ-type.
- Dans le second cas, l'utilisateur, outre certains opérateurs de construction de valeurs, n'a rien à ajouter à son C-sur-type pour être pris en considération dans METÉO.
- Enfin, et ce dans les deux cas, l'utilisateur est libre de (re-)définir un certain nombre d'opérations ou de prédicats qui n'étaient pas présents dans le C-sur-type, ou qui, tout en ayant la même spécification quant à leur résultat, sont munis d'une réalisation différente.

La section suivante traite un exemple d'insertion de C-type.

Lorsque la définition d'un nouveau C-type est complète au regard des exigences de METÉO, il est intégré au système, et acquiert le même statut que tout autre C-type. En ce sens, les mécanismes globaux de METÉO, tels que la création, l'insertion, ou la suppression de δ-types, opèrent exactement de la même façon sur les C-types de base que sur des nouveaux C-types, mêmes complexes. En effet, les C-types étant accessibles par les mécanismes globaux de METÉO, le sont par des opérations nécessaires à la définition d'un C-type, justement paramétrées par ce C-type.

Au regard de l'extérieur, les C-types étant des types abstraits de données, gérés uniformément, l'introduction d'un nouveau C-type n'est finalement que l'apport d'une nouvelle structure de données avec ses opérations et ses techniques de gestion des δ-types, qui répondent à une sémantique commune à tous les C-types.

3.6.2 Exemple d'insertion d'un nouveau C-type

Un agent extérieur souhaite créer une structure de données permettant l'expression *dates*, c'est à dire de triplets (jour,mois,année). Il va donc s'orienter vers la création du C-type `Date`. Il existe, en tout premier lieu, au moins deux possibilités de spécification de ce C-type :

- il s'agit d'un type ordonné, isomorphe au type `Entier`, et l'on souhaite représenter cet ordre, et en conséquence le C-type `Date` est déclaré initialement comme étant un C-sous-type de `Discret`,
- il s'agit d'un C-type construit, `Date` = `Entier` × `Entier` × `Entier`, par exemple, et en conséquence `Date` est déclaré comme C-sous-type de `Record` ou `Construit`.

Dans le premier cas, le mode de représentation des δ-types, ainsi que la spécification des opérations sur les valeurs, ne sont pas modifiés. Dans le second cas, le mode de représentation est affiné, en conséquence les opérations de définitions et de manipulation de cette représentation sont à affiner. Nous choisissons de spécifier le type `date` par la première solution. On distingue alors deux nouvelles possibilités :

- ce type est prédéfini dans le langage hôte, et de ce fait, il est possible de définir les opérations requises pour le C-type `Date` à partir des primitives offertes pour ce type de données dans le langage hôte.
- le type `Date` n'est pas disponible dans le langage hôte, et les opérations requises doivent donc être entièrement définies.

Nous supposons par la suite que le langage hôte ne définit pas le type `Date`.

Le C-type `Date` est déclaré comme étant un C-sous-type de `Discret`. En conséquence, le mode de représentation de ses δ-types est la liste d'intervalles à bornes fermées, et il s'agit de définir pour `Date`, outre les prédicats d'égalité et d'appartenance, le prédicat d'ordre sur les valeurs, et les opérations `successeur(v)` et `prédécesseur(v)` qui sont requises pour la gestion des intervalles.

prédicat d'appartenance d'une valeur à `Date`

$$
\begin{array}{lll}
\in_{Date}(v) \iff & v = (j\ m\ a) & \\
& \text{et } \in_{Entier}(j) & \\
& \text{et } \in_{Entier}(m) & \\
& \text{et } \in_{Entier}(a) & \\
& \text{et } (& (j \in^{\delta}_{Entier} \langle Entier; [1;30] \rangle \\
& & \text{et } m \in^{\delta}_{Entier} \langle Entier; [4;4]+[6;6]+[9;9]+[11;11] \rangle) \\
& \quad \text{ou} & (j \in^{\delta}_{Entier} \langle Entier; [1;31] \rangle \\
& & \text{et } m \in^{\delta}_{Entier} \langle Entier; [1;1]+[3;3]+[5;5]+[7;8]+[10;10]+[12;12] \rangle) \\
& \quad \text{ou} & (j \in^{\delta}_{Entier} \langle Entier; [1;28] \rangle \\
& & \text{et } \neg bissextile(a) \\
& & \text{et } =_{Entier}(m, 2)) \\
& \quad \text{ou} & (j \in^{\delta}_{Entier} \langle Entier; [1;29] \rangle \\
& & \text{et } bissextile(a) \\
& & \text{et } =_{Entier}(m, 2)) \\
& \quad) &
\end{array}
$$

prédicat d'égalité entre deux valeurs de `Date`

$$=_{Date}(v_1, v_2) \iff v_1 = (j_1\ m_1\ a_1), v_2 = (j_2\ m_2\ a_2)$$

$$\begin{aligned}
&\text{et } =_{Entier} (j_1, j_2)\\
&\text{et } =_{Entier} (m_1, m_2)\\
&\text{et } =_{Entier} (a_1, a_2)
\end{aligned}$$

prédicat d'ordre entre deux valeurs de `Date`

$$\begin{array}{lllll}
\leq_{Date} (v_1, v_2) \Longleftrightarrow & v_1 = (j_1\ m_1\ a_1), v_2 = (j_2\ m_2\ a_2) & & & \\
& \text{et } (& <_{Entier} (a_1, a_2) & & \\
& \quad \text{ou} & =_{Entier} (a_1, a_2) \text{ et } (& & <_{Entier} (m_1, m_2) \\
& & & \text{ou} & (=_{Entier} (m_1, m_2) \text{ et } \leq_{Entier} (j_1, j_2)) \\
& & \quad) & & \\
& \quad) & & &
\end{array}$$

Opérations annexes Il s'agit des opérations `successeur(v)` et `prédécesseur(v)` dont la définition précise, fort intuitive pour quiconque, n'a pas à être rappelée. En outre, ces opérations comportent la définition de la fonction *bissextile*(a) qui, à partir d'une année (un entier), détermine s'il s'agit d'une année bissextile ou non.

Une fois ces définitions données (et compilées), le C-type `Date` peut se comporter comme n'importe quel autre C-type, au regard des mécanismes globaux de METÉO qui le manipule, ainsi qu'au niveau de son statut externe de type abstrait[8].

3.6.3 Cas particulier des constructeurs unaires

La spécification d'un nouveau C-type T correspondant à un constructeur unaire, donc C-sous-type de `Multivalués`, entraîne automatiquement la création de tous les C-sous-types de $T(X)$ qui correspondent à l'application de ce constructeur à tous les C-types de base (c'est-à-dire que X est successivement remplacé par les C-types non construits). Dualement, lorsqu'un nouveau C-type est ajouté dans METÉO, s'il n'est pas un constructeur, alors sont explicitement créés tous les C-types correspondant à l'application des constructeurs unaires au C-type ajouté. Cela signifie, dans l'exemple précédent, qu'une fois `Date` créé, le sont aussi `Liste(Date)` et `Ensemble(Date)`.

Cette création automatique consiste, en théorie, à spécialiser le prédicat d'appartenance de $T(X)$ pour la prise en compte du prédicat particulier correspondant au C-type référencé, de même en ce qui concerne le prédicat d'égalité entre deux valeurs de $T(X)$. En pratique, les deux prédicats définis pour T le sont génériquement, avec comme paramètre instanciable le C-type référencé:

$$\in_{T(X)} (v) \Longleftrightarrow f(\in_X, v)$$

$$=_{T(X)} (v, v') \Longleftrightarrow f(=_X, v, v')$$

De ce fait, les prédicats d'appartenance et d'égalité n'ont pas à être explicitement définis dans les nouveaux C-types, la sélection des prédicats propres au type référencé s'effectuera dynamiquement.

Cette propriété s'applique de la même façon aux cinq opérations de manipulation des δ-types. Il s'agit là d'une illustration de la propriété de polymorphisme de METÉO.

[8] Lorsque le C-type à définir utilise entièrement des primitives du langage hôte, METÉO fournit un processus d'insertion du C-type qui évite la compilation.

3.6.4 Isomorphismes de C-types

Outre la possibilité d'ajouter de nouveaux C-types, METÉO permet la définition d'isomorphismes de C-types, c'est-à-dire l'écriture d'applications bijectives de la forme $b : \mathcal{T} \mapsto \mathcal{T}$. Par exemple, il est possible de réaliser l'application b :`Date` $\mapsto$ `Entier`.

Soit $b : T_1 \mapsto T_2$. b doit être tel que :

- à toute valeur de T_1 est associée une valeur unique de T_2, et vice-versa,
- à tout δ-type issu de T_1 peut être calculé un δ-type unique issu de T_2, et vice-versa,
- à toute opération impliquant des *valeurs* de T_1 doit être associée une opération reflétant le même comportement, impliquant des *valeurs* de T_2, et vice-versa.
- à toute opération de base sur les δ-types, à savoir le δ-sous-typage, l'élimination, le SUP, le INF et l'appartenance d'une valeur à un δ-type, définie dans T_1, il doit correspondre une unique opération équivalente dans T_2.

Bien évidemment, on n'impose pas, dans un tel isomorphisme, la mise en correspondance d'opérations de manipulations directes des modes de représentation des δ-types. En réalité, on ne cherche à établir l'équivalence qu'au niveau des opérations des C-types qui sont exportées et/ou utilisées par les mécanismes globaux de METÉO.

3.6.5 Limites de l'extensibilité

Aucune vérification n'est faite par METÉO lors de l'insertion d'un nouveau C-type, en ce qui concerne, d'une part, la cohérence des opérations au regard la sémantique qui leur est accordée dans METÉO, et d'autre part, vis-à-vis du respect de la sémantique du C-sous-typage. De la même façon, il n'est pas réalisé de vérification quant à la cohérence de la définition d'un isomorphisme de C-types.

Cette absence de vérification de cohérence lors de l'ajout d'un C-type nous incite à relativiser la propriété d'extensibilité de METÉO. Cependant, il s'agit ici d'un problème de vérification de spécifications de types abstraits, selon la théorie des algèbres multi-sortes ordonnées [GJM85], ce qui n'est pas du ressort de notre travail. On peut cependant affirmer que le développement d'un langage de spécification de types abstraits, inspiré de OBJ2 [FGJM85] basé sur les algèbres de sortes ordonnées, ASL [SW83] ou encore Larch [GHW85], permettrait de lever ce problème, à condition qu'un tel langage s'adapte à la double fonction des C-types : spécifications propres au comportement des valeurs, et spécifications propres au comportement des δ-types. En effet, ces langages réalisent des spécifications de types abstraits de données, et dans le cas de sortes ordonnées, ils assurent le respect de la sémantique de la relation d'ordre entre les types (sortes).

La propriété d'extensibilité de METÉO n'est en fait qu'une conséquence de l'organisation hiérarchique des C-types, et du soucis d'homogénéité dans le traitement de ces C-types.

3.7 Conclusion

Nous avons présenté dans ce chapitre les deux niveaux de types de METÉO. Le **premier de ces niveaux** correspond à une hiérarchie arborescente de *C-types*, un C-type étant l'implémentation d'un type abstrait de données, augmenté des structures et opérations permettant la représentation du **second niveau de typage**, à savoir les δ-types. Les δ-types sont des expressions de types bien formées, qui sont exprimés dans la syntaxe définie par le C-type duquel ils sont issus : les δ-types correspondent à l'expression de sous-ensembles arbitraires de l'ensemble dénoté par leur C-type de création. Les δ-types d'un même C-type sont organisés en treillis, ordonnés par la relation de δ-sous-typage. La composition de l'ordre sur les C-types et de l'ordre sur les δ-types donne lieu à la définition du γ-sous-typage, traduisant l'inclusion ensembliste des ensembles dénotés par les δ-types (issus de C-type éventuellement différents) mis en relation.

Nous pouvons d'ores et déjà émettre de nombreuses perspectives quant à l'enrichissement théorique et pratique de METÉO. Parmi ceux-ci, nous pouvons citer :

- développer des techniques de codage des sous-typages, [AKBLN89] [HN94] [Fal95] [Ell93], dans la perspective d'augmenter la rapidité des accès aux types ; ces codages sont pourtant destinés à n'être qu'appliqués à des ordres figés (une modification de l'ensemble ordonné pouvant mener à un recalcul intégral du codage),
- développer un système de réalisation automatique de C-types à partir d'un langage de spécification de types abstraits de données [ST88],
- transformer le C-sous-typage simple (représentation arborescente) en un C-sous-typage multiple (représentation en graphe) : cela permettrait par exemple la définition de C-type à la fois constructeurs et à la fois ordonnés,
- développer au sein de METÉO une technique de résolution des expressions cycliques de δ-types [Neb91] [AC91], les structures de treillis se prêtant à l'application des sémantiques des points fixes.

Notre objectif immédiat n'est pourtant pas dans la résolution de toutes ces extensions, le but de notre travail concerne l'étude de l'impact du développement d'un système de types pour un modèle de connaissances à objets, en particulier au regard de la représentation de l'interprétation intensionnelle des entités de représentation du modèle, et en conséquence des ordres, relations et autres mécanismes définis par le modèle sur ces entités.

Les deux niveaux de typage de METÉO sont destinés, pour le premier, à recevoir des définitions de structures de données et des opérations sur ces structures, et pour le second, à représenter l'interprétation intensionnelle des entités de représentation du modèle de connaissances auquel est dédié METÉO : c'est à ce typage qu'est consacré le chapitre suivant. Par ailleurs, le γ-sous-typage est destiné à représenter l'interprétation intensionnelle de la spécialisation, et nous verrons qu'une combinaison du γ-sous-typage et de l'opération de INF permet la représentation l'interprétation intensionnelle des passerelles.

La comparaison de METÉO avec d'autres systèmes de types, notamment ceux rencontrés en programmation, n'est pas très pertinente, dans la mesure où les objectifs sont différents. Plus exactement, un des objectifs servis par METÉO concerne la représentation des interprétations intensionnelles des composantes d'un modèle de connaissances à objets. Nous retiendrons toutefois la propriété de polymorphisme de METÉO, qui est un gage de son extensibilité. Nous verrons, dans les trois chapitres suivants, que METÉO est particulièrement adapté pour être faiblement couplé avec un modèle de connaissances à objets.

Chapitre 4

Typage des bases de connaissances

La section 2.4.1 a introduit le rôle que doivent jouer les types des entités de représentation des connaissances, et plus généralement la raison d'être d'un système de types dédié à un modèle de connaissances à objets. Informellement, le type d'une entité est la représentation de son intension indépendamment de son extension. Plus formellement, cela signifie que le type d'une entité correspond à la réécriture de la description de cette entité en termes du langage de représentation des connaissances vers un terme du langage d'expressions de types. Pourtant, il est bien évident que le typage d'une entité n'est pas une simple réécriture syntaxique. En effet, la création de ces types est motivée par le besoin de déléguer au système de types les processus qui opèrent sur les intensions des entités, notamment les vérifications/inférences de liens d'instanciation intensionnelle, de spécialisation intensionnelle, ainsi que la vérification de l'interprétation intensionnelle des passerelles. En conséquence, l'organisation du système de types doit être telle qu'il permette la mise en œuvre de processus équivalents sur les types des entités. Cette équivalence est nécessaire si le système de types se voit confier l'exécution des tâches opérant sur les descriptions des entités de représentation.

Soit $\mathcal{A}_L = \langle \Sigma_L ; \mathcal{D}_I \rangle$ l'algèbre dont la signature est issue de la syntaxe du langage de représentation des connaissances, et dont le domaine est celui de l'interprétation intensionnelle, les termes de cette algèbre représentent des entités de représentation. Soit $\mathcal{A}_T = \langle \Sigma_T ; \mathcal{V} \rangle$ l'algèbre dont la syntaxe est celle du système de types et dont le domaine est le domaine récursif de Scott, les termes de cette algèbre sont des types. L'équivalence entre les opérations intensionnelles du modèle et les opérations du système de types se traduit par l'existence d'un isomorphisme entre $\mathcal{A}_L$ et $\mathcal{A}_T$ que nous allons chercher à définir.

Ce chapitre est dédié, dans un premier temps, à utiliser les caractéristiques de MetéO pour l'élaboration de la signature de $\mathcal{A}_T$, et dans un second temps, à élaborer l'isomorphisme requis entre $\mathcal{A}_L$ et $\mathcal{A}_T$. Les structures de données proposées par MetéO ont été étudiées en vue de permettre la traduction des descriptions des termes du langage de représentation, de même que les relations établies entre types et valeurs (appartenance, sous-typage) ont été élaborées dans l'optique de représenter l'interprétation intensionnelle des relations clés du modèle de connaissances (instanciation, spécialisation). En particulier, l'organisation hiérarchique des termes de MetéO, due au sous-typage, a été conçue pour assurer l'équivalence entre la spécialisation intensionnelle et le sous-typage entre types de classes (figure 4.1). De la même façon, l'instanciation intensionnelle pourra être traduite par l'appartenance d'une valeur à un type.

La première section de ce chapitre met en évidence quelles sont exactement les structures sous-jacentes aux concepts, classes, attributs et instances d'après leur description, et en déduit les propriétés des types qui doivent les représenter. Les sections 4.2 et 4.3 s'attachent à montrer de

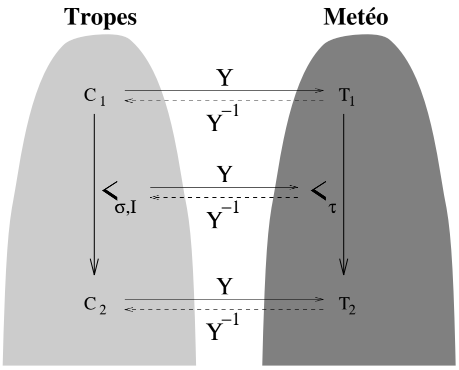


FIG. 4.1 - : Υ est l'application de $\mathcal{A}_L$ vers $\mathcal{A}_T$. Ce chapitre montrera que par cette application, la spécialisation intensionnelle définie dans TROPES est équivalente au sous-typage tel que défini par METÉO.

quelle façon le sous-typage peut représenter les interprétations intensionnelles de la spécialisation et des passerelles. La section 4.4 développe deux algorithmes de typage, brut et incrémental, des entités d'une base de connaissances. Ces deux algorithmes sont partiellement fondés sur la normalisation des δ-types réalisée dans METÉO, et illustrent les étapes de l'interprétation des entités de représentation vers un ensemble de valeurs, effectuées par METÉO lors du typage. La conclusion de ce chapitre comporte, entre autres, la mise en évidence de l'isomorphisme entre $\mathcal{A}_L$ et $\mathcal{A}_T$.

Il est important de noter que dans ce chapitre, les contraintes MICRO posées sur les entités d'une base de connaissances ne sont pas prises en compte dans l'intension de ces entités. Leur intégration sera développée dans le chapitre suivant.

4.1 Typage des entités de représentation

À chaque ensemble d'éléments E_L (instances ou valeurs élémentaires), dénoté par une entité, est associé un ensemble de valeurs E_V représenté par le type de cette entité. Le but de cette section est de caractériser cette application, en deux phases :

- tout d'abord, en établissant la correspondance qui existe entre les éléments du domaine d'interprétation intensionnelle des entités et les valeurs du domaine d'interprétation des types (domaine de Scott),
- ensuite, après analyse des structures des ensembles d'éléments E_L, et ce en fonction de la catégorie de généralité de l'entité qui le dénote (concept, classe, attribut), en associant à ces ensembles des structures de données adéquates définies dans METÉO.

Ceci constitue la première étape de l'identification de l'application Υ à mettre en évidence entre $\mathcal{A}_L$ et $\mathcal{A}_T$.

4.1.1 Identification des valeurs

Le modèle de connaissances manipule généralement des instances et des valeurs élémentaires à un même niveau d'interprétation, notamment lorsqu'il s'agit de valeurs d'attributs. Ainsi, instances et valeurs élémentaires sont des éléments du domaine de l'interprétation intensionnelle. Pourtant, une instance dans ce domaine n'est plus une identité, elle n'y véhicule que sa description, sa structure d'agrégation d'autres éléments.

En ce sens, quelle que soit la nature de l'élément, il sera représenté dans le système de types par une valeur, éventuellement construite à partir d'autres valeurs, à partir des constructeurs disponibles dans MÉTÉO pour répondre à l'existence des constructions similaires de TROPES.

La fonction `compute-value(e)` associe à un élément du domaine d'interprétation intensionnelle sa valeur dans le domaine de Scott. Il s'agit de l'application qui désigne l'homomorphisme Υ entre $\mathcal{A}_L$ et $\mathcal{A}_T$ appliqué aux opérateurs constants ($\forall e \in \mathcal{D}_I$, `compute-value(`$e$`)` $= \Upsilon(e) \in \mathcal{V}$). Il s'agit de l'identité dans le cas de valeurs élémentaires (appartenant à des types de base). Dans le cas d'une valeur construite sur d'autres éléments, cette fonction applique l'équivalence des opérateurs de construction dans MÉTÉO aux valeurs associées aux éléments de construction de la valeur initiale : soit $e =< const > (e_1, \cdots, e_n)$, `compute-value(`$e$`)` $= \Upsilon(< const >)($`compute-value(`e_1`)`$, \cdots,$ `compute-value(`e_n`))`. Le cas particulier des éléments qui sont des instances est traité dans la section 4.1.2.

4.1.2 Identification des types

Il s'agit dans cette section d'analyser, pour chaque catégorie d'entités, les caractéristiques que doivent avoir les types représentant leurs intensions, d'après la structuration des ensembles d'éléments dénotés par la description de ces entités.

Types de concepts

L'intension d'un concept correspond à l'agrégation de ses attributs, elle peut donc être représentée par un *type record*, où les étiquettes sont les identificateurs d'attributs, et les types associés aux étiquettes sont naturellement l'expression des domaines de valeurs des attributs. Le domaine de valeurs d'un attribut de concept est connu d'après son type principal.

Types de classes

De même qu'un concept, l'intension d'une classe est l'agrégation de ses attributs, qui peut aussi se représenter par un type *record*. Les types associés aux identificateurs d'attributs sont les domaines de valeurs associés aux attributs de classes. Le domaine de valeurs d'un attribut de classe est issu d'une restriction du domaine de valeurs de l'attribut de concept correspondant.

Les types de classes, tout comme les types de concepts, sont des types *records*. Ces derniers ont d'ailleurs été introduits pour les langages à objets, dans l'optique de représenter les types des classes.

La définition d'un type *record* associé à une classe (ou un concept) de représentation nécessite le typage des attributs décrivant cette classe (ce concept).

Types d'attributs

Le typage d'un attribut est nécessaire au typage des classes et concepts, puisque ces derniers sont des *records* de types d'attributs.

En ce qui concerne les attributs, la notion d'intension n'est pertinente que dans le cas des attributs complexes : l'intension d'un attribut complexe est une (éventuelle) restriction de l'intension de l'entité à laquelle il fait référence. Le type d'un attribut complexe sera donc issu du type de l'entité référencée. Le type d'un attribut élémentaire, quant à lui, sera simplement la représentation en termes du langage d'expressions de types, du domaine de valeurs normalisé spécifié pour cet attribut par les descripteurs de typage.

Le typage des attributs peut se diviser en deux groupes provenant de leur statut déclaratif, c'est-à-dire du niveau de généralité auquel ils sont déclarés.

- *Les attributs de concept* ne sont spécifiés que par la donnée de leur type principal (descripteurs `$un, $liste-de` et `$ens-de`). Ainsi, si ce type existe déjà dans le système de types, le typage de l'attribut est immédiat, c'est le cas des attributs élémentaires. Quant aux attributs complexes, leur typage consiste à typer le concept référencé.
- *Les attributs de classe* sont spécifiés par les descripteurs de restriction de domaines, leur type est donc une restriction du type principal de l'attribut de concept correspondant, soit un sous-type du type de l'attribut de concept. Le typage d'un attribut de classe revient donc à interpréter les descripteurs de restriction comme des constructeurs de sous-ensembles issus de l'ensemble représenté par le type principal.

Qu'il s'agisse de types d'attributs de classe ou d'attributs de concepts, ils doivent cependant être exprimés en termes du même langage, interprétables dans le même univers de valeurs, car ils représentent des ensembles de valeurs comparables. Le type principal est choisi pour le comportement de ses valeurs, lorsqu'il est élémentaire, et pour la signification générale de ses individus lorsqu'il est complexe. Mais quelle que soit la nature du type principal, un affinement effectué par un attribut de classe ne relève pas obligatoirement de la volonté d'accéder à un comportement plus précis des valeurs.

Valeurs et types d'instances

De même qu'à un concept ou à une classe est associé un type *record*, à une instance peut être associée une *valeur record*, qui donne pour toute étiquette la valeur donnée par l'instance pour l'attribut dont l'identificateur correspond à cette étiquette, que cet attribut soit complexe ou élémentaire. La fonction `compute-value(i)` rend la valeur *record* associée à l'instance en paramètre. Autrement dit, la valeur associée à l'instance résulte de l'application des constructeurs de valeurs *records* aux couples (attribut = élément) qui composent la description de l'instance, et qui ont été préalablement transformés par Υ (Υ est l'identité lorsqu'elle s'applique aux identificateurs d'attributs). L'application de Υ aux instances termine puisque les définitions récursives dans TROPES sont interdites, et a fortiori le sont les objets cycliques.

Une instance appartient initialement à un concept, puis elle est ensuite attachée à un ensemble de classes, une par point de vue. Il est alors pertinent de parler du type d'une instance pour désigner le type *record* issu de l'intersection des types *records* de ses classes d'appartenance. Cela représente en fait le type (l'intension) de la classe qui, structurellement, correspondrait à la plus grande sous-classe commune des classes d'appartenance de l'instance (figure 4.2).

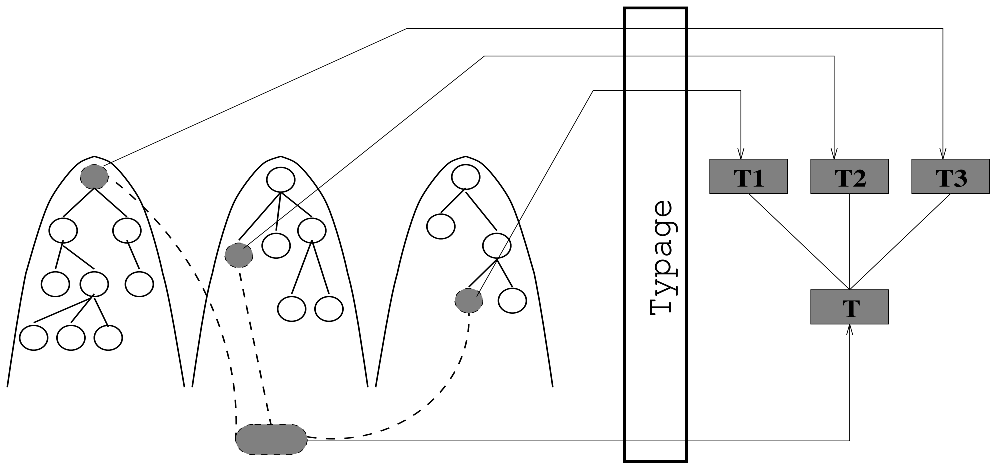


FIG. 4.2 - : Le type de l'instance est calculé par l'intersection des types de ses classes d'appartenance.

4.1.3 Représentation des types issus de la base de connaissances

Les types issus des entités de représentation des connaissances sont représentés par des δ-types, issus d'un C-type caractérisé par la nature de l'entité typée. En effet, ces entités ne peuvent être typées par des C-types car ils sont trop grossiers vis-à-vis des ensembles qu'ils dénotent. De plus, tous les types des entités doivent être homogènes dans leur expression, dans la mesure où ils seront amenés à être comparés indépendamment de ce à quoi ils correspondent. Ainsi, même si par exemple le C-type `Entier` dénote toutes les valeurs entières et qu'il s'agit du type d'un attribut de concept, ce dernier sera représenté par le δ-type maximum de `Entier`, de façon à ce qu'il partage les mêmes opérations de manipulation que tout autre δ-type issu de `Entier`, telles que la conjonction de δ-types ou le test de δ-sous-typage.

Le C-type duquel sont issus les δ-types correspondant au typage des classes, concepts et instances, est le C-type `Record`. Un δ-type issu de `Record` est ainsi un type *record* particulier, correspondant à un type de concept, de classe ou d'instance, dont les identificateurs correspondent aux attributs décrivant l'entité typée, et les types associés à ces identificateurs sont ceux des attributs.

Le C-type duquel est issu le δ-type correspondant au typage d'un attribut est celui spécifié par son type principal. S'il s'agit d'un attribut complexe, le C-type est `Record`, modulo l'application des constructeurs `Liste` ou `Ensemble` à `Record`. S'il s'agit d'un attribut élémentaire, le C-type est simplement obtenu par le descripteur de typage principal (il s'agit éventuellement d'un type construit).

Un type d'attribut de classe sera un δ-type issu du C-type spécifié par l'attribut de concept correspondant. Ce δ-type est calculé à partir de l'interprétation des descripteurs de restriction vers l'ensemble des valeurs dénoté par le C-type.

Dans MÉTÉO, seuls les δ-types issus du typage d'une base de connaissances seront explicitement exprimés, ainsi que des δ-types intermédiaires.

4.1.4 Gestion des liens dynamiques

Les δ-types issus du typage d'une base de connaissances contiennent, dans leur structure, l'information relative à leur existence. MÉTÉO gère ainsi les liens qui unissent les δ-types aux entités de représentation, ainsi que les liens de dépendances entre δ-types.

Dépendances Tropes/Metéo

Une des deux raisons qui ont amené la création d'un δ-type est qu'il correspond au résultat du typage d'une entité de représentation. À chaque classe, concept, instance ou attribut est associé un unique δ-type, mais un même δ-type peut correspondre à plusieurs entités de représentation. METÉO et TROPES gèrent de pair chaque lien bidirectionnel, qui est stocké d'une part dans le schéma des entités et d'autre part dans la structure du δ-type. Ces liens sont appelés `type` (entité vers δ-type) et `owners` (δ-type vers entités) ; leur sont associées quatre opérations élémentaires de lecture/écriture :

- `get-type(e)` qui renvoie le type stocké pour l'entité `e`,
- `put-type(e,t)` qui stocke dans le schéma de `e` le type `t`,
- `get-owners(t)` qui renvoie l'ensemble des entités typées par `t` et stockée dans la description de `t`,
- `put-owner(e,t)` qui ajoute l'entité `e` à l'ensemble stocké par `t` qui contient les entités qu'il type.

En réalité, la définition et l'exécution des opérations `get-type` et `put-type` sont confiées à TROPES qui doit garder le contrôle sur la modification et la consultation des schémas de ses entités. Les deux autres opérations sont définies et exécutées par METÉO.

Dépendances de δ-types

Un δ-type explicitement créé n'est pas obligatoirement associé à une entité de représentation, son existence peut être nécessaire à la représentation d'un δ-type issu d'un C-type correspondant à un constructeur. En effet, le mode de représentation d'un δ-type de construction δt nécessite la donnée de tous les δ-types sur lesquels δt se construit. Lors du typage d'une entité dont le type principal est construit, la création de son δ-type nécessite la création des δ-types sur lesquels il est construit. Si parmi ceux-ci certains n'existent pas encore, ils sont créés sans obligatoirement être directement associés à une entité de connaissance. Par exemple, un attribut spécifié par `$liste-de Entier $parmi [1..4]` va nécessiter la création du δ-type $\langle Entier; [1;4] \rangle$, qu'il corresponde ou non à une entité de représentation de la base de connaissances.

Il s'agit dans ce cas d'un *lien de dépendance* entre deux δ-types, qui peut mener à plusieurs degrés de profondeur (un δ-type amorce la création d'un autre δ-type qui lui même nécessite la création d'un troisième δ-type, etc.). Ces créations en chaîne ne bouclent pas car les définitions récursives sont interdites dans TROPES, de même qu'est impossible dans le langage de TROPES d'exprimer une application infinie de constructeurs[1].

METÉO maintient en interne les liens de dépendance entre δ-types, selon le même principe qu'est maintenue la relation qui lie les entités de représentation aux δ-types qui en sont issus. L'information sur ce lien est contenue dans la structure du δ-type référencé (le lien inverse étant donné par le champ `type-ref` du δ-type d'origine), il est appelé `depend` et géré par deux opérations définies et exécutées par METÉO :

- `get-depend(t)` qui renvoie la liste des δ-types qui le référencent, stockée dans la structure de `t`,

[1] Ceci est garanti par la syntaxe de TROPES.

– `put-depend(t1,t2)` qui ajoute `t2` dans la liste des δ-types référençant `t1`, liste stockée dans la structure de `t1`.

Il est important de noter que le lien `depend` contient des *ensembles* de δ-types, et non pas un seul d'entre eux, du fait qu'un δ-type puisse dépendre de la création de plusieurs autres : toutes ces informations doivent être stockée.

En outre, les δ-types d'un C-type devant être organisés en treillis, certains δ-types doivent être créés par MeTÉO afin que toutes les propriétés des treillis soient préservées (figure 4.3). Cette information qui traduit une des raisons d'être d'un δ-type est maintenue dans la définition même de ce δ-type, de la même façon qu'elle contient ses liens de dépendances, par le lien `lattice` qui est un booléen. `lattice` est `true` si le type existe car sa création provient de la nécessité de maintenir la structure de treillis du C-type dont il est issu. `lattice` est `false` pour un δ-type si les liens `depend` et `owner` sont vides et si ce δ-type est nécessaire à la préservation de la sructure de treillis. De même que les deux précédents liens, `lattice` est géré par deux opérations de lecture/écriture : `get-lattice(t)` et `put-lattice(t,b)` où `b` est un booléen.

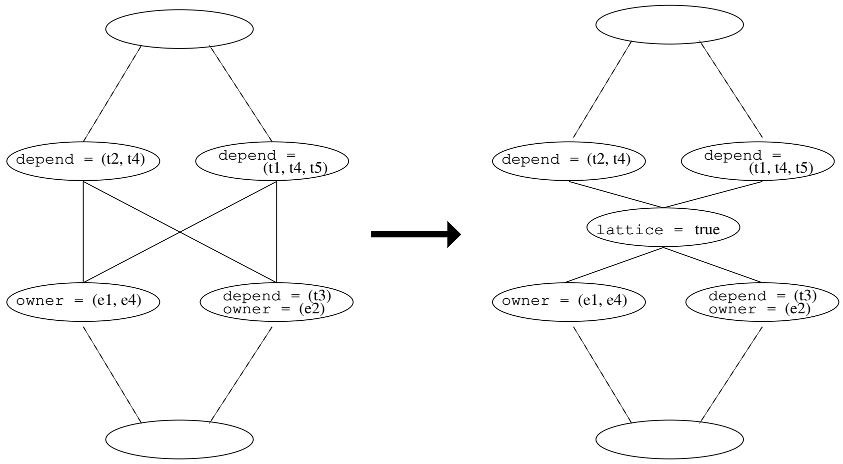


FIG. 4.3 - : Pour préserver l'existence et l'unicité de l'INF (ou SUP) de deux δ-types dans un treillis, un δ-type doit être créé.

Maintien du δ-sous-typage

Outre les liens de dépendances, les δ-types possèdent dans leur structure deux champs contenant l'information sur le δ-sous-typage ordonnant $\widehat{\mathcal{G}(T)}$. On rappelle que $\widehat{\mathcal{G}(T)}$ est le treillis des δ-types issus du C-type T.

– **`sous-types`**, dont la valeur, pour chaque δ-type δt, est la liste des δ-sous-types directs de δt dans $\widehat{\mathcal{G}(T)}$,

– **`sur-types`**, dont la valeur, pour chaque δ-type δt, est la liste des δ-sur-types directs de δt dans $\widehat{\mathcal{G}(T)}$.

Ces deux champs sont maintenus pour chaque δ-type par MeTÉO par quatre opérations élémentaires de lecture/écriture, dont les signatures sont les suivantes :

– `get-sur-types(`t`)`, qui a pour résultat la liste des sur-types directs de t,

- `put-sur-types(`t`,`t'`)`, qui a pour effet d'ajouter t' à la liste des sur-types de t,
- `get-sous-types(`t`)`, qui a pour résultat la liste des sous-types directs de t,
- `put-sous-types(`t`,`t'`)`, qui a pour effet d'ajouter t' à la liste des sous-types directs de t.

Résumé

Chaque δ-type voit augmenter sa structure informatique de deux informations relatives à la raison de l'existence de ce δ-type : un δ-type existe soit parce qu'il est directement issu du typage d'une entité de représentation (lien `owner`), soit parce que la création d'un autre δ-type lui faisait référence (lien `depend`), ou soit parce que son existence est nécessaire à la préservation de la structure du treillis de δ-types auquel il appartient.

En outre, les liens de δ-sous-typage sont maintenus à l'intérieur même des structures de δ-types.

4.1.5 Typage et instanciation

Le test d'appartenance d'une valeur à un δ-type *record* telle que définie dans MetÉo (page 112) permet la vérification de l'instanciation stricte et de l'instanciation large.

Appartenance d'une instance I à un concept K

Le principe du test de cette appartenance est d'associer à I sa valeur *record* r, et de tester l'appartenance de r au type $t(K)$ du concept, c'est-à-dire tester la validité de $r \in^{\delta}_{Record} t(K)$. Un δ-type *record* est de la forme $\langle Record; [R; D_1; D_2]\rangle$. Le type d'un concept n'est que l'agrégation des types de ses attributs, donc pour tout concept K, son type est de la forme $T(K) = \langle Record; [R; all; nothing]\rangle$. Ainsi, la définition de l'appartenance d'une valeur à un type de concept est simplifiée : soient $R = \langle\langle e_1 : t_1, \cdots, e_n : t_n\rangle\rangle$ et $\forall i, t_i = \langle T_i; E(T_i)\rangle$:

$$r = \langle e_1 = v_1, \cdots, e_p = v_p\rangle \in^{\delta}_{Record} \langle Record; [R; all; nothing]\rangle \Longleftrightarrow p \leq n \text{ et } \forall i \in [1..p], v_i \in^{\delta}_{T_i} t_i$$

Cette définition est déduite de la définition inductive de $\in^{\delta}_{Record}$[2].

Cette définition est exactement équivalente à la définition de l'instanciation intensionnelle (définition 4, section 2.2.2), modulo la rigidité des conditions (appartenance stricte ou large). MetÉo définissant aussi un test d'appartenance large d'une valeur à un δ-type, quel que soit ce δ-type (l'appartenance est validée si la valeur est inconnue), les deux variantes de l'instanciation intensionnelle peuvent être correctement et complètement validées par les tests d'appartenance d'une valeur *record* à un δ-type *record*.

Appartenance d'une instance I à une classe C

Elle s'effectue selon le même principe que précédemment. Toutefois, la vérification de la correction d'une telle appartenance doit être précédée de la vérification de l'appartenance de I au concept K de C. En effet, l'intension de C n'est pas incluse dans celle de K (au sens classique des

[2] L'induction est remplacée par la condition $p \leq n$ et par le déploiement de la structure des valeurs et types *records*. Les conditions sur D_1 et D_2 ont disparu car sont devenues des tautologies dans le cas de $D_1 = all$ et $D_2 = nothing$.

records) puisque C ne considère qu'un sous-ensemble des propriétés décrivant K, donc l'affinement par les propriétés n'est pas effectif (même si les propriétés de K reprises par C sont affinées). Afin d'éviter ce double test systématique, qui s'effectuera sur les types de la classe et du concept, nous proposons une révision du type d'une classe, en complétant son δ-type associé par les couples (identificateur / type) qui correspondent aux attributs de K non repris par C. Le type de la classe devient alors le type résultant de la conjonction entre le type du concept et le type original de la classe, d'après la définition de la conjonction sur les types *records* définie par Luca Cardelli [Car84] et reprise dans la section 3.4.6 (opération INF) pour l'adapter à la définition étendue des δ-types *record*. Cette nouvelle définition du type d'une classe permet de vérifier, sans redondance de tests, l'appartenance de la valeur *record* au type du concept et au type de sa classe (figure 4.4).

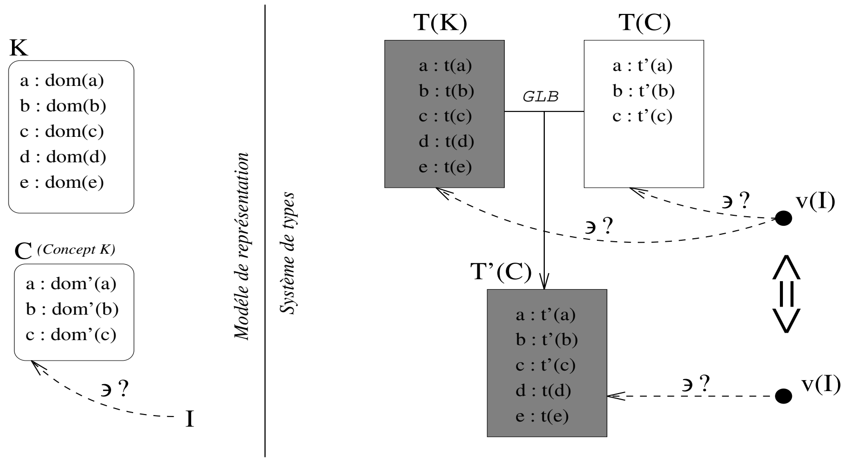


FIG. 4.4 - : Une valeur *record* qui appartient à deux types *records* appartient à leur conjonction (INF), et vice-versa. La vérification de l'appartenance intensionnelle d'une instance I à une classe C d'un concept K est donc totalement équivalente à l'appartenance de la valeur *record* de I ($v(I) =$ `compute-value(I)`) à la conjonction des types *records* de C et K ($T(C) \sqcap_{Record} T(K) = T'(C)$, $T'(C)$ est le type stocké pour C).

Conclusion

Qu'il s'agisse de l'instanciation d'une classe ou d'un concept, sa validation intensionnelle est équivalente à l'évaluation du prédicat d'appartenance d'une valeur *record* à un δ-type *record*.

Propriété 4 (Instanciation intensionnelle et appartenance à un type *record*)

$$\forall O = C \text{ ou } K, I \in_I \|O\|^I \iff \textit{compute-value}(I) \in^{\delta}_{Record} \textit{get-type}(O)$$

La preuve de cette propriété dépend de la preuve de correction du typage des attributs, puisque la vérification de l'appartenance d'une valeur *record* r à un type *record* t exige la vérification de l'appartenance de chaque valeur étiquetée de r au type correspondant au même identificateur (étiquette) dans t (cette vérification est sémantiquement équivalente à la vérification de l'appartenance d'une valeur au domaine d'un attribut). Ainsi, dans la mesure où pour chaque C-type, ce test d'appartenance est correct vis-à-vis de la sémantique des descripteurs de typage (c'est-à-dire si $\forall e \in \mathcal{D}_I, \forall d = \bigwedge(< desc >_i < dom >_i)$ une combinaison de descripteurs de typage, $e \in d \iff \Upsilon(e) \in_\tau \Upsilon(d)$), le test d'appartenance d'une valeur *record* à un δ-type *record* est correcte vis-à-vis de l'instanciation, à condition cette fois que le déploiement[3] des *records* termine. Cette

[3] On appelle *déploiement* d'une valeur (type) *record* l'opération qui consiste à substituer toute valeur (type) étiquetée issue d'un type construit par les valeurs (types) sur lesquelles elle est obtenue par application du constructeur, jusqu'à ne plus avoir que des valeurs (types) élémentaires.

deuxième condition est vérifiée puisque les définitions récursives dans TROPES sont interdites, et que la syntaxe de TROPES interdit l'application successive de deux constructeurs.

4.2 Spécialisation et sous-typage

Le typage des entités de représentation a pour but une représentation effective et adaptée de leur intension, indépendante de leur signification dans le monde modélisé (extension), afin de déléguer au système de types l'exécution des processus opérant sur les structures (intensions) des entités de représentation. Parmi ces processus, outre la vérification de l'instanciation qui a été vue dans la section précédente, la spécialisation de classes joue un rôle prédominant dans un système de représentation par objets.

4.2.1 Spécialisation de classes

De la même façon que l'instanciation, la spécialisation intensionnelle entre classes est complètement et correctement validée par le test de δ-sous-typage entre des types *records* issus du typage des classes. Vis-à-vis de l'application Υ, cela signifie que $\Upsilon(\leq_{\sigma,I}) = \leq^{\delta}_{Record}$.

Le principe du test de spécialisation d'une classe C_1 par une classe C_2 du même point de vue consiste à considérer les types $t(C_1)$ et $t(C_2)$ de ces deux classes, et à tester la relation de δ-sous-typage sur ces deux δ-types *records*, c'est-à-dire évaluer $t(C_2) \leq^{\delta}_{Record} t(C_1)$. Le type d'une classe C n'étant que l'agrégation des types de ses attributs, son type est de la forme $T(C) = \langle Record; [R; all; nothing] \rangle$, ce qui amène à simplifier la définition du sous-typage sur les δ-types *record*. Soient $R_1 = \langle\langle e_1 : t_1, \cdots, e_p : t_p \rangle\rangle, R_2 = \langle\langle e_1 : t'_1, \cdots, e_n = t'_n \rangle\rangle$ et $\forall i, t_i = \langle T_i; E(T_i) \rangle$ et $t'_i = \langle T'_i; E(T'_i) \rangle$

$$\langle Record; [R_2; all; nothing] \rangle \leq^{\delta}_{Record} \langle Record; [R_1; all; nothing] \rangle \Longleftrightarrow p \leq n \text{ et } \forall i \in [1..p], t'_i \leq_{\gamma} t_i$$

Cette définition, déduite de celle du sous-typage sur les δ-types *records* (page 3.4.6), correspond exactement à la définition 6 de la spécialisation intensionnelle.

Propriété 5 (Spécialisation intensionnelle et δ-sous-typage entre *records*)

$$C_1 \leq_{\sigma,I} C_2 \Longleftrightarrow \textit{get-type}(C_1) \leq^{\delta}_{Record} \textit{get-type}(C_2)$$

Cette propriété est elle aussi dépendante de la correction du sous-typage entre types d'attributs élémentaires, vis-à-vis de la sémantique intensionnelle de l'affinement des propriétés. À son tour, la correction du sous-typage entre δ-types élémentaires est complètement tributaire de la correction du typage, dans la mesure où le sous-typage dans METÉO est homogènement défini à partir du prédicat d'appartenance d'une valeur à un δ-type. La terminaison du test de sous-typage entre deux types *records* correspondants à des classes est assurée car TROPES interdit les définitions récursives.

Vérifications des propriétés : exhaustivité et exclusivité

La vérification de la spécialisation doit aussi assurer le respect des propriétés des taxonomies des points de vue, à savoir l'exclusivité et l'exhaustivité.

- L'exhaustivité se vérifie grâce à l'opération SUP (Least Upper Bound, ou disjonction) définie pour les δ-types *records*, mais aussi, par déploiement, par cette opération définie pour tous les autres C-types. Un arbre de spécialisation $\mathcal{A} = \langle \mathcal{C}, \mathcal{S} \rangle$ (où $\mathcal{C}$ est l'ensemble des classes de l'arbre de spécialisation, et $\mathcal{S}$ est l'ensemble des arêtes) est exhaustif si et seulement si $\forall C \in \mathcal{C}, C_1, \cdots, C_n$ telles que $\forall i \in [1..n], C_i \leq_\sigma C$

  $$\bigvee_{i=[1..n]} T(C_i) = T(C)$$

  où $\vee$ est l'opération SUP. Cette condition exprime le fait que le type associé à chaque attribut de la super-classe doit être entièrement couvert par les types associés à ce même attribut dans les sous-classes.

- L'exclusivité se vérifie grâce à l'opération de INF (Great Lower Bound, ou conjonction) définie pour les δ-types *records*, mais aussi, par déploiement, par cette opérations définie pour tous les autres C-types. Un arbre de spécialisation $\mathcal{A} = \langle \mathcal{C}, \mathcal{S} \rangle$ est exclusif si et seulement si $\forall C \in \mathcal{C}, \forall C_1, C_2$ telles que $\forall i \in [1..2], C_i \leq_\sigma C$, $R = \langle\langle e_1 : t_1, \cdots, e_p : t_p \rangle\rangle$:

  $$T(C_1) \sqcap_{Record} T(C_2) = \langle Record; [R; all; nothing] \rangle, \text{ et } \exists k \in [1..p] \text{ tel que } t_k = \bot$$

Ces conditions respectent les spécifications intensionnelles de ces propriétés, mais doivent être considérées en toute relativité. En particulier, les contraintes portant sur l'exhaustivité ne peuvent pas être considérées comme nécessaires (et certainement pas suffisantes). Leur non-respect, doublé du respect de l'interprétation extensionnelle, est cependant un bon indicateur, par exemple, d'une mauvaise spécification des propriétés de la super-classe.

Treillis des types de classes

Les δ-types issus du C-type `Record` explicitement exprimés dans MÉTÉO sont ceux issus du typage des concepts, classes et instances de la base de connaissances, plus certains δ-types intermédiaires. MÉTÉO maintient le treillis de tous ces δ-types, qui possède, vis-à-vis de la base de connaissances, la propriété nécessaire bien que triviale suivante :

Propriété 6 (plongement) *L'arbre de spécialisation de tout point de vue d'un concept est* plongé *dans le treillis des δ-types* records *issus du typage de la base de connaissances dans laquelle est défini le concept.*

Notons en outre que dans le treillis des δ-types *records*, le type d'une instance est nécessairement un sous-type de ceux de ses classes d'appartenance, puisqu'il est le résultat de l'opération INF entre ces types de classes.

4.2.2 Affinement des propriétés descriptives des classes

La spécialisation interprétée par le δ-sous-typage entre types *records* est tributaire du δ-sous-typage entre les types d'attributs. De même que les types de concepts, classes et instances, forment un ensemble partiellement ordonné, dont le treillis est maintenu par METÉO, les types d'un même attribut, ordonnés par le δ-sous-typage défini pour leur C-type d'appartenance, sont plongés dans le treillis des δ-types de ce C-type. Soit $\mathcal{T}(T, a)$ le treillis des δ-types issus du C-type T et correspondant aux typages de l'attribut a (donc extrait à partir de $\Delta(T)$). Ce treillis illustre tous les affinements de a dans le concept, il correspond à la projection fermée[4] du treillis des types de classes et instances selon a.

Les propriétés structurelles du treillis des δ-types d'un attribut sont pertinentes vis-à-vis du rôle que joue cet attribut dans la description de la modélisation.

- La profondeur de ce treillis, relativement à la profondeur du treillis des types *records*, indique l'importance du rôle que la propriété représentée par l'attribut joue dans le processus de modélisation incrémental. Autrement dit la profondeur du treillis reflète la pertinence de l'attribut dans les critères d'affinement (spécialisation) de la connaissance.
- La largeur relative de ce treillis reflète la pertinence de la propriété représentée par l'attribut en ce qui concerne le découpage d'une catégorie d'individus en plusieurs sous-catégories exclusives : l'attribut est important dans le *partitionnement* d'une classe en plusieurs sous-classes.

En résumé, nous dirons que le treillis des δ-types d'un attribut contient une information quant à la capacité discriminante de cet attribut.

La figure 4.5 illustre un exemple de treillis de δ-types, avec les liens de références entre δ-types ainsi que les liens de typage, δ-sous-typage et spécialisation.

4.3 Passerelles, INF et sous-typage

Tout comme l'instanciation et la spécialisation, à partir du sous-typage peut être construit un processus de vérification de l'interprétation intensionnelle des passerelles. Pour cela, nous transposons la définition en intension d'une passerelle en termes d'opérations du système de types, en nous rappelant que les conditions intensionnelles portent essentiellement sur les domaines des attributs partagés par plusieurs classes parmi la source et la destination.

Ainsi, on vérifie que pour chaque attribut défini par au moins une de ces classes, l'intersection des domaines alloués à cet attribut dans chacune des classes sources le possédant ne doit pas être vide, et doit de plus être incluse dans le domaine de cet attribut dans la classe destination, si elle le définit. Pourtant, la notion d'"attribut existant" dans une classe n'a pas d'équivalent au niveau du type d'une classe ; en effet, lorsque, dans la base de connaissances, un attribut de concept n'est pas considéré par une classe, il reste présent dans le type de cette classe, le type de cet attribut dans le type de la classe est alors celui qui lui est associé dans le type du concept. En conséquence, tous les tests effectués sur les attributs communs à plusieurs classes impliquées dans une passerelle, le sont, par défaut, sur tous les attributs du concept.

[4] La projection fermée d'un treillis de types *records* selon une étiquette est le treillis obtenu après deux étapes de calcul :

1. parmi les types *records* possédant cette étiquette, seuls les types associés à cette étiquette sont retenus, et les liens de sous-typage sont recopiés ou obtenus par transitivité,
2. les éventuels liens de sous-typage entre les types de l'étiquette (ceux qui n'étaient pas reflétés dans le treillis des types *records*) sont calculés.

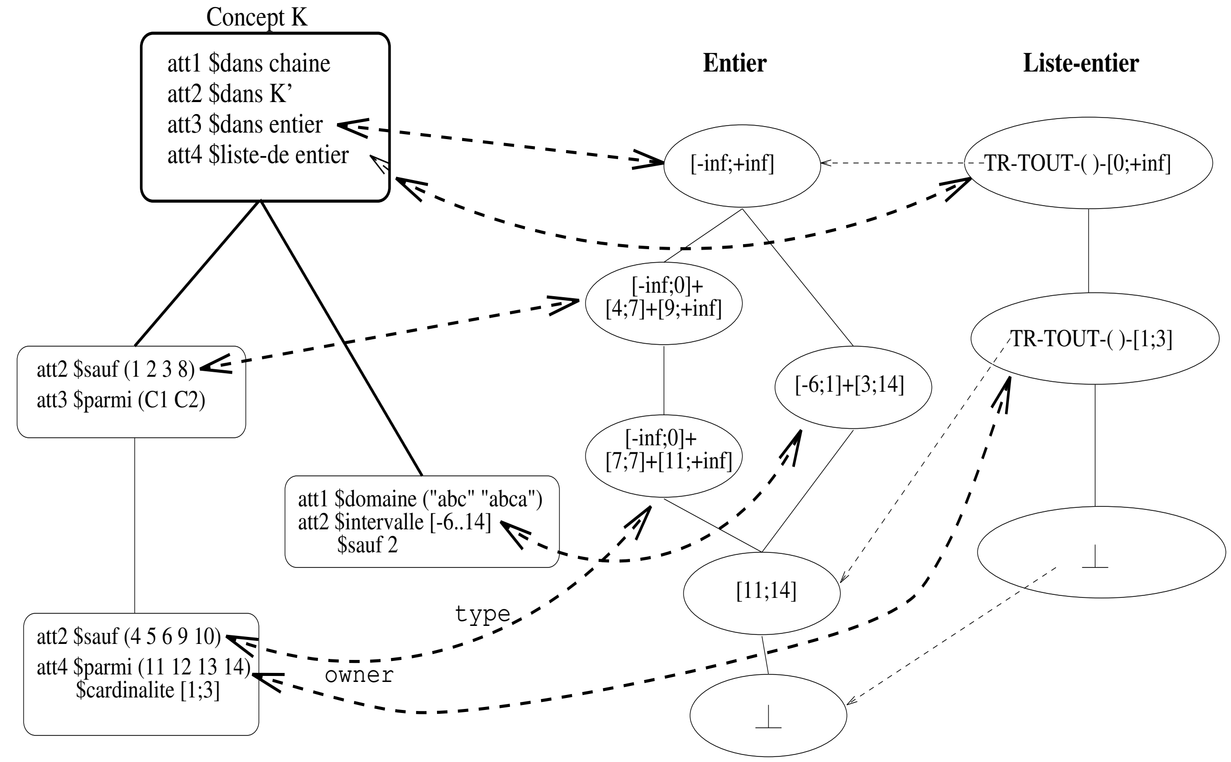


FIG. 4.5 - : Deux hiérarchies de spécialisation sont issues du concept `K`. Cette figure illustre pour deux des attributs de ce concept les treillis de δ-types. On remarque la liaison entre les deux treillis, du fait de la dépendance du C-type `Liste-entier` vers le C-type `Entier`. Le δ-sous-typage sur les entiers reflète la spécialisation.

Un attribut défini dans deux classes de la source mène systématiquement au calcul de l'intersection de leurs domaines respectifs. On introduit alors le calcul du type de la source d'une passerelle, qui doit refléter la prise en compte simultanée, par intersection, de tous les domaines des attributs du concept spécifiés dans les descriptions des classes de la source. L'intersection des domaines est représentée dans MetéO par l'opération INF, définie aussi au niveau des types *records*. Le type d'une source est ainsi défini comme suit :

$$T_p = T(\{C_1, \cdots, C_n\}) = \sqcap_{Record}(T(C_1), \cdots, T(C_n))$$

(l'opération INF étant associative, nous l'avons, par souci de clarté, étendue d'une opération binaire vers une opération n-aire). On note que le type d'une source est en fait le plus petit sur-type commun à ceux de toutes les instances appartenant simultanément à toutes les classes de la source.

Nous définissons maintenant l'opération $\gg_\tau$ destinée à être l'équivalente de $\gg_I$, qui reprend donc les deux critères posés sur les domaines des attributs partagés.

Définition 27 (passerelle entre types *records* $\gg_\tau$) *Soit un ensemble de types records* $\{T_1, \cdots, T_n\}$*, et soit* $T_p = \sqcap_{Record}(T_1, \cdots, T_n)$*. Soit* T *un type* record*. On définit une passerelle entre* δ*-types* records *de la façon suivante :*

$$\{T_1, \cdots, T_n\} \gg_\tau T \Longleftrightarrow T_p \neq \bot \ \ et \ \ T_p \leq_{Record} \Big(\prod_{e \in Etiq(T_p)} T\Big)$$

où $\boldsymbol{Etiq}(\langle Record; [R; D_1; D_2]\rangle)$ *est l'ensemble des étiquettes du* record R.

La propriété 7 est alors immédiate :

Propriété 7 (Passerelles, INF et δ-sous-typage) $\forall C_1, \cdots, C_n, \forall C$,

$$\{C_1, \cdots, C_n\} \gg_I C \Longleftrightarrow \{\texttt{get-type}(C_i)\}_{i=1..n} \gg_\tau \texttt{get-type}(C)$$

Il est à noter ici que la base de connaissances doit signifier à MÉTÉO quels sont les attributs des classes sources pour la vérification de $\gg_\tau$. En effet, la projection sur les étiquettes du type de la source n'est pas très pertinente puisque le type de chaque classe comporte les étiquettes de tous les attributs du concept, même lorsqu'elle ne les possède pas tous. Ainsi, la projection devra être effectuée sur l'ensemble des attributs qui apparaissent au moins dans une des classes sources.

4.4 Algorithmes de typage

Le typage d'une entité de représentation consiste à déterminer l'expression de δ-type dont l'ensemble de valeurs correspond à l'intension de cette entité. Dans le cas d'un attribut élémentaire, le typage n'est que l'expression d'une forme normalisée du domaine de valeurs spécifié par les descripteurs. Dans les deux cas, le typage délivre une expression normalisée de la description d'une entité de représentation.

4.4.1 Principes du typage

Les δ-types issus du typage d'une base de connaissances doivent être sous *forme normale*, de façon à ce que toute comparaison soit immédiate. Il est d'ailleurs important de noter que les types des entités sont tous exprimés à un même niveau d'interprétation, celui des δ-types, qu'il s'agisse de types de classes ou d'attributs. La raison première de ce choix d'expression des types d'entités provient du fait que tous les δ-types issus de la base de connaissances sont amenés à être comparés, tous sont exploités par les mêmes mécanismes de manipulation et de vérification. Cette possibilité d'homogénéité dans les traitements est l'illustration du fait que les types ne sont que les expressions des intensions, des descriptions de ces entités, *indépendamment de leur interprétation dans le monde modélisé*: s'il n'y a aucune conséquence à comparer les types d'un attribut et d'une classe, cela n'est pas vrai d'une comparaison similaire au niveau du modèle de connaissances : classes et attributs ont deux statuts complètement différents dans un modèle de connaissances, du fait de leur interprétation extensionnelle.

L'algorithme de typage est constitué de trois étapes principales.

1. Traduction du langage de représentation des connaissances vers EOLE. Dans le cas des attributs, cela revient à interpréter les descripteurs de typage vers les champs exprimant un δ-type, en fonction du C-type. Dans le cas des concepts et classes, cela nécessite l'interprétation de la description d'un concept (classe) en termes d'attributs spécifiés.

2. Normalisation des champs du δ-type, à partir de la procédure de normalisation définie pour chaque C-type.

3. La phase de typage d'une entité est éventuellement suivie de l'insertion du δ-type t obtenu dans le treillis issu de son C-type, c'est-à-dire de la recherche de sa position et de la mise à jour des liens de δ-sous-typage. Cette étape n'est effectuée qu'en cas de *validation* de l'entité typée dans la base de connaissances, c'est-à-dire de la reconnaissance de l'identité de cette entité. Cela reflète la relativisation de l'intension vis-à-vis de l'extension d'une entité.

Le typage d'une entité est effectué à la création de cette entité dans la base de connaissances, qu'elle soit ou non déjà explicitement liée à d'autres entités. En particulier, si une erreur de type survient durant cette phase, le rattachement de l'entité à celles acceptées dans la base est refusé. Le typage d'une classe ou d'un concept est suivi de la validation de ce type, conséquence de l'acceptation de l'entité dans la base de connaissances (reconnaissance d'une extension). La validation d'un type T est appelée par le modèle de connaissances, elle est notée `Valide(O,T)`.

MÉTÉO propose deux algorithmes de typage différents, mais qui contiennent tous deux les trois étapes principales données précédemment. Le premier algorithme est celui du *typage brut*, dont le principe est de typer une entité o sans prendre en compte les types des autres entités auxquelles o est liée. Le second algorithme est le *typage incrémental*, qui, par définition, ré-utilise les types des entités déjà typées pour déterminer celui de l'entité courante. Chacun de ces deux algorithmes distingue le typage des classes et concepts, le typage des attributs, et le typage des instances, justement du fait de leurs statuts extensionnels distincts.

Ces deux algorithmes de typage seront utilisés dans deux contextes différents : le typage incrémental d'une entité sera activé lorsque la position de cette entité dans la hiérarchie est connue, alors que le typage brut sera mis en œuvre pour une entité dont on ne connaît pas (encore) la position au regard de la spécialisation.

4.4.2 Typage brut

Le principe du typage brut d'une entité est que l'*élaboration* de son type est effectuée *indépendamment de l'existence de tout autre type*, et en conséquence, en ne faisant cas d'aucune des hiérarchies effectives de δ-types. La procédure de typage brut d'une entité `O` est appelée `Typing-b(O)`.

Typage brut des concepts et des classes

Le typage brut d'un concept, comme celui d'une classe, est constitué de deux phases, la première concernant la traduction de la description de l'entité en termes du langage de représentation vers EOLE, la seconde correspondant à la mise sous forme normale du type obtenu à l'issue de la première phase. L'algorithme de `Typing-b(O)`, où `O` est un concept ou une classe, est décrit ci-dessous. Il utilise les notations suivantes : A(`O`) désigne l'ensemble des attributs décrivant `O`; $id(a)$ dénote l'identificateur de l'attribut a.

1 Initialisation de T(`O`), le δ-type correspondant à `O`

1.1. Création d'un δ-type issu du C-type `Record`
$T(\mathtt{O}) \longleftarrow \langle Record; [R; D_1; D_2] \rangle$
1.2. Initialisation des champs d'énumération
$D_1 \longleftarrow$ `all`
$D_2 \longleftarrow$ `nothing`
1.3. Initialisation du champ contenant le *record* des types d'attributs
Soit $A(\mathtt{O}) = (a_1, \cdots, a_n)$
Début
$R_0 \longleftarrow \langle\langle\ \rangle\rangle$
$i \longleftarrow 1$
`erreur` $\longleftarrow$ `false`
Tant que (`Erreur` = `false`) et ($i \leq n$) faire

t =Typing-b(a_i)
Si $t = \perp$ Alors Erreur $\longleftarrow$ True
Sinon $R_i \longleftarrow \langle\langle R_{i-1} \mid id(a_i) : t\rangle\rangle$
$i \longleftarrow i+1$
FinSi
FinTantque
$R \longleftarrow$ (Erreur$\vee R_n$)
Fin

L'étape suivante ne s'exécute qu'à la condition que l'initialisation de $T(0)$ n'ait pas mené à une erreur.

2 Normalisation de $T(0)$: $T(0) \longleftarrow NormalForm_{Record}(\langle Record; [R;\texttt{all};\texttt{nothing}]\rangle)$

Le processus de validation qui peut être activé à l'issue du typage, marquant ainsi l'acceptation de la création de l'entité maintenant typée, concerne dans Metéo l'insertion du type dans son treillis et la mise à jour des différents liens qui attachent le δ-type t créé à tous les autres. L'étude de la procédure de validation est le sujet de la section 4.4.7.

Typage brut d'un attribut

Ce qui différencie le typage d'un attribut du typage d'une classe, c'est principalement le mode de description d'un attribut. La description d'une classe ou d'un concept est son agrégation d'attributs, ce qui a amené une interprétation naturelle de la description d'une telle entité vers les types *records*. La desription d'un attribut, quant à elle, est définie par les descripteurs de typage qui confèrent à l'attribut un domaine de valeurs autorisées. Le typage d'un attribut relève donc d'une transformation des descripteurs vers les champs du C-type donné pour cet attribut. Cette transformation constitue en partie la première phase du typage, la suivante étant similaire à celle identifiée pour le typage d'un concept.

L'interprétation des descripteurs vers les champs d'un C-type est dépendante de ce C-type. D'une part, les C-types sont regroupés en catégories reflétant leur mode de représentation des δ-types. Donc, d'une catégorie à l'autre, les champs d'expression des δ-types diffèrent (section 3.4.2). D'autre part, tous les descripteurs ne sont pas considérés par toutes les catégories de C-types (section 2.1.5). Il s'agit alors d'examiner, pour chaque catégorie de C-type, quels sont les descripteurs pris en compte, et quelle est l'interprétation qui en est faite.

La phase d'élaboration du δ-type d'un attribut, qui précède la normalisation, est alors constituée de cinq étapes séquentielles : récupération de la spécification complète de l'attribut (prise en compte de tous les descripteurs qui participent à la définition du domaine de valeurs de l'attribut), détermination du C-type de l'attribut, initialisation du δ-type, réécriture syntaxique de l'information véhiculée par les descripteurs (syntaxe Tropes vers syntaxe Eole), intégration des descripteurs réécrits dans les champs du δ-type. Cette phase d'élaboration est décrite dans l'algorithme ci-dessous, il s'agit de la procédure `Elaborate-type-b(a)`, où `a` est l'attribut à typer. On note $t(a)$ son type en cours d'élaboration.

1 *Récupération des descripteurs de l'attribut*, par activation du mécanisme d'héritage sur a, et par recopie des descripteurs de typage principal donnés dans la description de l'attribut de concept correspondant. La description complète de a est alors de la forme

$$\langle < const > < id - type >; (< descr >_i < info >_i)\rangle$$

où $< const >$ est l'identificateur d'un constructeur ($un, $liste-de ou $ens-de), $< id - type >$ est un identificateur de type de base, d'un concept, ou une liste d'identificateurs de classes, $< descr >$ est l'identificateur d'un descripteur de restriction de domaine ($intervalle, $parmi, $valeur...), et $< info >$ est la donnée d'une restriction de domaine.

2 *Sélection du C-type*, à partir du descripteur de typage principal, par interprétation de l'application du constructeur au type spécifié, selon le schéma suivant :

$< const >$	$< id - type >$	$\Longrightarrow$	C-type
$un	Concept	$\Longrightarrow$	`Record`
$un	Liste de classes	$\Longrightarrow$	`Record`
$un	Identificateur de type de base T	$\Longrightarrow$	$ADT(T)$
$liste-de	Concept	$\Longrightarrow$	`Liste(Record)`
$liste-de	Liste de classes	$\Longrightarrow$	`Liste(Record)`
$liste-de	Identificateur de type de base T	$\Longrightarrow$	`Liste`($ADT(T)$)
$ens-de	Concept	$\Longrightarrow$	`Ensemble(Record)`
$ens-de	Liste de classes	$\Longrightarrow$	`Ensemble(Record)`
$ens-de	Identificateur de type de base T	$\Longrightarrow$	`Ensemble`($ADT(T)$)

où $ADT(T)$ est une fonction de l'interface entre TROPES et METÉO, qui, à un identificateur de type de TROPES associe son C-type de base dans METÉO. Lorsqu'un $< id - type >$ n'est pas adapté au constructeur, alors la procédure d'élaboration du type génère une `Erreur`.

3 *Initialisation du δ-type de* `a`, $t(a)$. Cette initialisation est dépendante du C-type, puisqu'elle consiste à initialiser les champs du δ-type nécessaires dans la catégorie du C-type. Le tableau ci-dessous indique, pour chaque catégorie de C-types, de quelle façon sont initialisés les champs d'expression de δ-type.

C-type T	domaine	comp-domaine	type-ref	card
Types de base	`all`	`nothing`	–	–
Ordonnés	$[\lfloor T \rfloor ; \lceil T \rceil]$	–	–	–
Énumération	`all`	`nothing`	–	–
Construits ($< const >$ $T_1, \cdots, T_n$)	`all`	`nothing`	$\top_{T_1}, \cdots, \top_{T_n}$	–
Multivalués ($< const >$ T')	`all`	`nothing`	$\top_{T'}$	$[0; \lceil Entier \rceil]$
Record ($[T_1, \cdots, T_n]$)	`all`	`nothing`	$\top_{T_1}, \cdots, \top_{T_n}$	–

Dans le cas où il s'agit d'un *record* issu d'une référence à un concept (une liste de classes), le champ **type-ref** s'initialise avec le type de ce concept (l'INF des types des classes).

4 *Réécriture des restrictions de domaines données par les descripteurs*, cette phase consiste donc, pour chaque identificateur de descripteur de restriction, à réécrire le domaine spécifié dans la syntaxe usuelle de METÉO, avec *interprétation des valeurs*. Le tableau ci-dessous indique le principe de cette réécriture, pour chaque descripteur et en fonction du C-type. Certains descripteurs ne sont pas pris en considération pour certains C-types : si un descripteur est associé à un attribut alors que le C-type ne peut l'interpréter, alors la procédure d'élaboration du δ-types génère une `Erreur`.

Syntaxe TROPES	C-type	Syntaxe METÉO
\$valeur v	Univers, T	$(CV_T(v))$
\$valeur v	Ordonnés, T	$[CV_T(v);CV_T(v)]$
\$domaine $(v_1, \cdots, v_n)$	Univers, T	$(CV_T(v_1), \cdots, CV_T(v_n))$
\$domaine $(v_1, \cdots, v_n)$	Ordonnés, T	$CR_T(CV_T(v_1), \cdots, CV_T(v_n))$
\$sauf $(v_1, \cdots, v_n)$	Univers, T	$(CV_T(v_1), \cdots, CV_T(v_n))$
\$sauf $(v_1, \cdots, v_n)$	Ordonnés, T	$CR_T(CV_T(v_1), \cdots, CV_T(v_n))$
\$intervalle $I_1, \cdots, I_n$	Ordonnés, T	$GR_T(I_1) + \cdots + GR_T(I_2)$
\$sauf-intervalle $I_1, \cdots, I_n$	Ordonnés, T	$GR_T(I_1) + \cdots + GR_T(I_2)$
\$parmi $C_1, \cdots, C_n$	Record	$\sqcap_{i=1..n}$`get-type`(C_i)
\$parmi $(v_1, \cdots, v_n)$	Multivalués $<$ *construct* $>$ T'	`Typing-b(`\$un T' \$domaine $(v_1, \cdots, v_n)$`)`
\$parmi $I_1, \cdots, I_n$	Multivalués $<$ *construct* $>$ T'	`Typing-b(`\$un T' \$intervalle $I_1, \cdots, I_n$`)`
\$interdit $C_1, \cdots, C_n$	Record	$\sqcup_{i=1..n}$`get-type`(C_i)
\$interdit $(v_1, \cdots, v_n)$	Multivalués $<$ *construct* $>$ T'	`Typing-b(`\$un T' \$domaine $(v_1, \cdots, v_n)$`)`
\$interdit $I_1, \cdots, I_n$	Multivalués $<$ *construct* $>$ T'	`Typing-b(`\$un T' \$intervalle $I_1, \cdots, I_n$`)`
\$cardinalité $[v_1; v_2]$	Multivalués	$[CV_{Entier}(v_1); CV_{Entier}(v_2)]$

Où :

- CV_T est l'abréviation de *compute-value*$_T(v)$ qui est la fonction d'interprétation des valeurs et instances de TROPES vers des valeurs de METÉO, définie pour chaque C-type par METÉO à partir des fonctions de constructions nécessairement présentes dans la définition de chaque C-type,
- CR_T est l'abréviation de *compute-range*$_T(D)$ qui est la fonction qui, à partir d'une liste de valeurs, en extrait la liste d'intervalles corrects. Cette fonction ne s'applique bien entendu qu'aux C-types ordonnés. Dans le cas des C-types `Discrets`, les intervalles résultant seront à bornes fermées, contrairement aux intervalles du C-type `Continus`.
- GR_T est l'abréviation de *ground-range*$_T(I)$ qui est une fonction qui, après interprétation des valeurs aux bornes de l'intervalle par *compute-value*$_T$, adapte les bornes de l'intervalle en fonction du mode de représentation de T.

Si une erreur est détectée par les opérations propres aux C-types, lors de la réécriture des descripteurs en syntaxe EOLE, alors cette erreur est récupérée par `Elaborate-type-b` qui produit une `Erreur`.

5 *Intégration des descripteurs réécrits dans les champs du δ-type.* Cette phase est une double interprétation, à la fois de la spécialisation (qui, sur les domaines de valeurs, revient à la conjonction de la spécification héritée et de la spécification re-définie), et de la sémantique des descripteurs. Cette intégration est décrite dans le tableau suivant, indépendamment de la syntaxe de représentation (qui a été unifiée dans la phase précédente).

Descripteur réécrit	Interprétation sur le δ-type issu du C-type T
\$valeur v	**domaine** $\longleftarrow$ **domaine** $inter_T$ v
\$intervalle (I_i)	**domaine** $\longleftarrow$ **domaine** $inter_T(I_i)$
\$sauf-intervalle (I_i)	**domaine** $\longleftarrow$ **domaine** $remove_T(I_i)$
\$domaine D	**domaine** $\longleftarrow$ **domaine** $inter_T$ D
\$sauf D	(Ordonnés) **domaine** $\longleftarrow$ **domaine** $remove_T$ D (Autres) **comp-dom** $\longleftarrow$ **comp-dom** $union_T$ D
\$cardinalité C	**card** $inter_{Entier}$ C
\$parmi D	**type-ref** $\longleftarrow$ **type-ref** $\sqcap_{Tref}$ D
\$interdit D	**type-ref** $\longleftarrow$ **type-ref** $\setminus_{Tref}$ D

Le mode de lecture du tableau est le suivant : chaque case représente une modification d'un champ du δ-type (noté en caractères gras), par intersection avec la restriction apportée par chaque descripteur. Ainsi, à un δ-type initial (seconde étape), est appliquée la liste $(< descr >_i < info >_i)$, couple par couple, jusqu'à ce que cette dernière soit vide. Bien entendu, l'application de la restriction n'est effectuée que dans le cas où elle a une répercussion sur l'un des champs du δ-type (ex. dans le tableau, le descripteur \$parmi n'a d'influence que sur le champ **type-ref**, donc ce descripteur n'est pas pris en compte lorsque le δ-type est issu d'`Entier`, ou, plus généralement, de tout autre C-type ne possédant pas le champ **type-ref**).

La procédure d'élaboration du type d'un attribut a pour résultat soit une première expression du δ-type, soit une erreur traduisant une incohérence. À l'issue de cette phase, la procédure de typage active la mise sous forme normale définie par le C-type duquel est issu le δ-type en résultat. Pour résumer, dans le cas d'un attribut, la procédure de typage brut `Typing-b(a)` est définie comme suit :

1. $t \longleftarrow$ `Elaborate-type-b(a)`
2. Si $t \neq$`Erreur` Alors $(t = \langle T; E(T)\rangle)$
   $t(a) \longleftarrow NormalForm_T(t)$
   FinSi

Le théorème 2 indique que la procédure de typage est correcte et complète au regard de l'interprétation intensionnelle des attributs. Ce théorème est prouvé en annexe B, en supposant correct et complet le processus de normalisation.

Théorème 2 (Correction et complétude du typage d'un attribut) *La procédure de typage brut d'un attribut est correcte et complète au regard de l'interprétation intensionnelle.*

$$\forall \tilde{a} = a : d, \exists t^N = \textit{Typing-b}(\tilde{a}) \textit{ et } ||\tilde{a}||^I = ||t^N||^\xi$$

La forme normale du δ-type correspondant à l'attribut sera ensuite éventuellement insérée dans le treillis de δ-types correspondant, sous réserve de la validation de cet attribut dans la base de connaissances, cette dernière résultant de la validation de la classe ou du concept qui contient cette description de l'attribut.

Typage brut d'une instance

Le type d'une instance est en réalité le type résultant de la combinaison des classes de rattachement de l'instance. Son calcul équivaut au typage d'une classe qui serait la plus grande sous-classe commune de celles de l'instance. C'est un moyen de représenter intensionnellement la spécialisation multiple, lorsqu'elle a lieu de l'être (figure 4.2).

Pourtant, toutes les combinaisons de classes de différents points de vue ne sont pas toutes prises en compte lors du typage, dans la mesure où la plupart d'entre elles n'ont pas de signification (lorsque sont réunies dans une combinaison des classes qui n'ont pas d'extension commune). Dans la mesure où le système ne connaît pas les vraies extensions des classes, mais seulement celles explicitement représentées dans la base de connaissances par instanciation, il se repose sur sa connaissance des extensions communes : lorsqu'une instance est attachée à plusieurs classes dans plusieurs points de vue, cela signifie que l'intersection des extensions de ces classes n'est pas vide et que calculer l'intension correspondante a un sens. C'est pourquoi le type d'une instance est calculé, comme l'intersection des types de ses classes de rattachement. La procédure de typage brut d'une instance `I`, notée `Typing-b(I)`, est la suivante. $t(I)$ désigne le type de l'instance, $C_1, \cdots, C_n$ sont ses classes de rattachement.

$$t(I) \longleftarrow \sqcap_{i=1..n} \texttt{Typing-b}(C_i)$$

Le calcul du type d'une instance revient à représenter l'intension de la classe, fictive, qui aurait comme super-classes directes celles auxquelles l'instance est attachée. On rejoint ici la raison d'être des *derived class* imaginées par Domenico Beneventano et Sonia Bergamaschi, qui sont la représentation explicite de la plus grande sous-classe commune à celles auxquelles est rattachée une instance [BB92a]. Les auteurs ont introduit cette notion dans le but d'accélérer le calcul de relations de subsomption intensionnelle entre description de concepts et d'individuels.

4.4.3 Typage incrémental

Le principe du typage brut est d'ignorer l'existence d'autres types issus de la base de connaissances. En particulier, cette procédure ne réutilise pas certains calculs déjà effectués.

Au contraire, le principe du *typage incrémental* est, pour le typage d'une entité O, de tirer parti des calculs déjà effectués sur les entités que spécialise O[5]. La différence majeure d'avec le typage brut est surtout visible lors du typage d'un attribut : le mécanisme d'héritage n'est pas activé pour la récupération des descripteurs, et à la place, l'initialisation du type d'un attribut revient à copier le type du plus petit sur-attribut. Autrement dit, la spécialisation intensionnelle n'est interprétée qu'au niveau du lien entre l'attribut courant et son sur-attribut, et non plus à tous les niveaux de spécialisation (figure 4.6).

La différence entre le typage brut et le typage incrémental se situe tout d'abord au moment de l'initialisation du δ-type. Puis, dans le cas du typage d'un attribut, le fait que seuls les descripteurs ajoutés soient considérés (les autres étant déjà pris en compte lors de l'initialisation) diminue considérablement le nombre de calculs. Notons enfin que le typage d'un concept est nécessairement brut, puisqu'il n'est pas concerné par la spécialisation (de même pour le typage des attributs de concept).

[5] Le typage incrémental pose comme condition que les types des entités soient toujours stockés et disponibles.

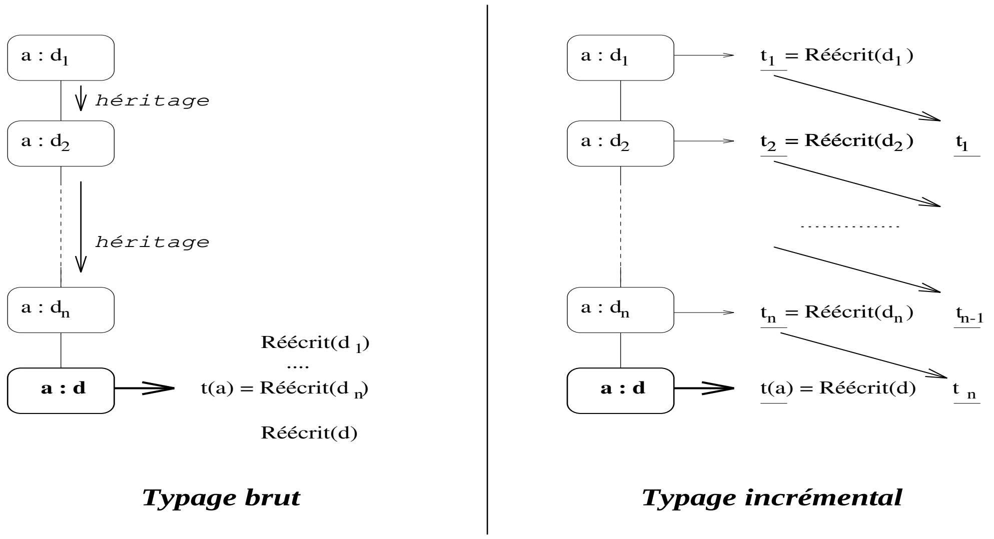


FIG. 4.6 - : La différence entre le typage incrémental et le typage brut est que le premier utilise les types déjà calculés et stockés alors que le second recalcule tout.

Le typage incrémental d'une entité de représentation `O` est effectué par la procédure `Typing-i(O)`. De même que le typage brut, le typage incrémental est éventuellement suivi d'une validation, qui mène alors à l'insertion du δ-type obtenu dans le treillis des δ-types correspondant, et à la mise à jour des liens de dépendances.

Typage incrémental d'une classe

Le typage incrémental d'une classe est effectué par une initialisation du type de sa sur-classe immédiate, puis par typage incrémental des attributs qu'elle affine, et par typage brut des attributs qu'elle ajoute. L'algorithme du typage incrémental d'une classe `C` est alors le suivant (la sur-classe immédiate de `C` est `sp(C)` :

1 Initialisation de $T(\texttt{C})$, le δ-type correspondant à `C`

1.1. Création d'un δ-type issu du C-type `Record`
  $T(\texttt{C}) \longleftarrow \langle Record; [R; D_1; D_2] \rangle$
1.2. Initialisation des champs d'énumération
  $T(\texttt{C}) \longleftarrow$ `get-type(sp(C))`
1.3. Calculs des types des attributs affinés et ajoutés
  Soit $A(\texttt{C}) = (a_1, \cdots, a_n)$ la description exacte de `C`, sans activation de l'héritage
  Début
    $R_0 \longleftarrow \langle\langle\ \rangle\rangle$
    $i \longleftarrow 1$
    `erreur` $\longleftarrow$ `false`
    Tant que (`Erreur` = `false`) et ($i \leq n$) faire
      Si a_i est un attribut ajouté Alors t =`Typing-b(`a_i`)`
        Sinon t =`Typing-i(`a_i`)`
      FinSi
      Si $t = \perp$ Alors `Erreur` $\longleftarrow$ `True`
        Sinon $R_i \longleftarrow \langle\langle R_{i-1} \mid id(a_i) : t \rangle\rangle$
          $i \longleftarrow i + 1$

FinSi
FinTantque
$R \longleftarrow$ (`Erreur`$\vee R_n$)
Fin

2 Normalisation de $T(\mathtt{0})$: $T(\mathtt{0}) \longleftarrow NormalForm_{Record}(\langle Record; [R; D_1; D_2]\rangle)$

Typage incrémental d'un attribut de classe

Le typage incrémental d'un attribut de classe consiste à récupérer le type de l'attribut correspondant dans la première super-classe où il apparaît, et d'appliquer à ce type les nouveaux descripteurs précédemment réécrits, selon la même sémantique que lors d'un typage brut. C'est donc la procédure d'élaboration du δ-type, notée `Elaborate-type-i(a)`, qui est modifiée par le typage incrémental.

L'algorithme de l'élaboration incrémentale d'un attribut `a` est alors le suivant (l'attribut correspondant à `a` dans la première super-classe où il apparaît est noté `sp(a)`) :

1 *Initialisation du δ-type de* `a` *noté $t(a)$*

$t(a) \longleftarrow$ `get-type(sp(a))`

2 *Réécriture des restrictions de domaines données par la description locale de* `a`, selon la même méthode que lors de l'élaboration brute (phase 4).

3 *Intégration des descripteurs réécrits dans les champs de $t(a)$*, selon la même sémantique que lors de l'élaboration brute (la différence étant que la liste des descripteurs est plus petite).

La procédure de typage incrémental d'un attribut de classe est alors similaire à celle du typage brut : `Typing-i(a)` =

1. $t \longleftarrow$ `Elaborate-type-i(a)`
2. Si $t \neq$`Erreur` Alors ($t = \langle T; E(T)\rangle$)
   $t(a) \longleftarrow NormalForm_T(t)$
   FinSi

Typage incrémental d'une instance

Le typage incrémental d'une instance consiste simplement à récupérer les types des classes de l'instance, à partir du moment où ils existent, et à calculer leur intersection.

$t(I) \longleftarrow \sqcap_{i=1..n}$ `get-type`(C_i)

4.4.4 Équivalence des deux procédures de typage

Du fait les deux procédures de typage sont équivalentes, excepté dans leur gestion des erreurs. En effet, le typage incrémental ne produit que des avertissements au regard de la correction de la spécialisation, alors que le typage brut déclenche une erreur lorsque la spécialisation n'est pas vérifiée.

Théorème 3 *Soit E une entité de représentation considérée par les deux procédures de typage (un attribut, une classe, un concept ou une instance). Si* `Typing-b(E)` $\neq \perp$ *alors :*

$$\texttt{Typing-b}(E) = \texttt{Typing-i}(E)$$

(il s'agit d'une équivalence syntaxique : les δ-types produits sont deux expressions syntaxiques de EOLE *égales).*

4.4.5 Typages et vérification de la spécialisation intensionnelle

La vérification de la spécialisation intensionnelle peut être effectuée à l'issue du typage d'une classe, et même plus directement à l'issue du typage d'un attribut. Il existe aussi, dans ce contexte, une différence importante entre le typage brut et le typage incrémental.

- Le typage brut permet d'effectuer une vérification de la spécialisation intensionnelle au fur et à mesure du typage des attributs, mais s'il s'avère que la spécialisation n'est pas vérifiée, l'acceptation ultérieure de l'attribut est impossible, a fortiori la validation de la description de la classe et donc son acceptation extensionnelle.
- À l'instar du typage brut, le typage incrémental peut être simultané à une vérification de la spécialisation intensionnelle au niveau des attributs, en analysant la validité de la restriction au moment de l'intégration des descripteurs réécrits. Pourtant, une éventuelle erreur de spécialisation peut être ignorée lors du typage puisque le type de l'attribut est forcément un sous-type de celui de son sur-attribut du fait de l'initialisation. Toutefois, une constatation d'erreur dans la spécialisation peut être levée par un avertissement (warning) à l'utilisateur, lui précisant que la description de son attribut, en termes du langage de représentation, ne vérifie pas la spécialisation intensionnelle, le δ-sous-typage étant cependant garanti par le système de types.

Un des avantages que l'on peut constater en faveur du typage incrémental est qu'une erreur de spécialisation intensionnelle détectée au niveau d'un attribut ne bloque pas le typage de la classe englobante : toutes les erreurs au niveau des attributs peuvent ainsi être signalée lors du même processus de typage de la classe. L'utilisateur est alors libre de modifier sa description pour qu'elle reflète réellement l'intension considérée par le modèle de connaissances, en l'occurrence celle calculée par le système de types.

4.4.6 Extensibilité de Metéo et typage

Quelle que soit la procédure de typage utilisée, elle comporte l'interprétation de la syntaxe de description de TROPES. En particulier, le typage doit considérer la sémantique associée à chacune des facettes de typage.

Nous avons vu dans le chapitre précédent que METÉO est extensible, à savoir qu'il permet la création de nouveaux C-types. Pourtant, si, comme nous l'avons vu, tous les C-types de base (non construits) sont tous accessibles dans TROPES par l'intermédiaire d'un identificateur, il n'en est pas de même des C-types construits, qui ne sont accessibles que par l'intermédiaire d'une facette de typage. De ce fait, la syntaxe de TROPES doit s'adapter à l'intégration de nouveaux constructeurs, par la création de facettes de typage adaptées. Cette création peut se faire automatiquement, elle mène à l'extension de la table située page 141.

Quelle que soit la nature d'un C-type ajouté, s'il introduit un nouveau mode de représentation de ses δ-types, les fonctions d'interprétation de la syntaxe TROPES vers la syntaxe étendue de EOLE doivent être précisées, au sein même du module de typage. En particulier, un nouveau mode de représentation des δ-types doit être accompagné de ses procédures de normalisation.

4.4.7 Validations

La validation d'un type commande son rattachement aux autres types de la base de connaissances. Dans le cas des types de classe, de concept ou d'instance, cette validation est commandée par le modèle de connaissances, de par l'acceptation de l'entité au sein de la base de connaissances[6].

La validation d'un type conduit à son insertion dans la base de δ-types, donc à la création d'un certain nombre de liens de dépendances (section 4.1.4), ainsi qu'à l'insertion du δ-type dans le treillis du C-type duquel il est issu.

La procédure de validation d'un type `t` issu d'une entité de représentation `O` est appelée `Validate(t,O)`. On distingue trois versions de cette procédure, la première étant réservée aux types de classes et concepts, la deuxième aux types d'instances et la dernière aux types d'attributs. Quelle que soit l'entité, si son δ-type est issu d'un type construit, la validation se répercute au niveau des δ-types référencés, de part la procédure `Propagate-valid(t)`. Et quelle que soit la version de la procédure de validation, elle se termine toujours par l'insertion du type validé dans son treillis, c'est-à-dire à la mise à jour de liens effectifs de δ-sous-typage.

Validation d'un type de classe ou de concept

Soit O une classe ou un concept, et $T(O) = \langle Record; [R; D_1; D_2] \rangle$ le type calculé pour O, avec $R = \langle\langle a_i : t_i \rangle\rangle_{i=1..n}$. `Validate`$(t, O)$, avec $t = T(O)$, se définit comme suit :

1 Propagation de la validation au niveau des attributs

Pour tout $a_i : t_i$ de R
  `Validate`(t_i, a_i)
  `put-depend`(t_i, t)

2 Attachement de l'entité à son type et du type à l'entité

`put-type`(O, t)
`put-owner`(t, O)

3 Insertion effective du type dans son treillis

`insert-new-type(Record,`t`)`

[6] L'acceptation d'une entité dans la base de connaissances traduit son acceptation extensionnelle, qui devient possible une fois que l'interprétation intensionnelle des relations a été vérifiée.

Validation d'un type d'attribut

La procédure de validation d'un type d'attribut est similaire à celle d'un type de classe ou de concept. Il s'agit de distinguer dans ce cas les types construits des types de base (simples), car la validation d'un δ-type issu d'un C-type construit nécessite la validation des δ-types sur lequel il est construit, activée par un appel à la procédure `Propagate-valid(`t`)` où t est le type référencé par t_a, le type de l'attribut à valider. `Validate(`t_a, a`)`, validation du type de l'attribut a, est définie ci-dessous. On note T_a le C-type duquel est issu t_a, $t_a = \langle T_a; E(T_a)\rangle$.

1 Propagation de la validation dans le cas d'un attribut de type construit

Si $T_a \leq_C$ `Construits` Alors $(E(T_a) = [type - ref; D_1; D_2; C^*])$
Pour tout $t_i \in type - ref$ faire
`put-depend(`t_i, t_a`)`
`Propagate-valid(`t_i`)`
FinSi

2 Attachement de l'attribut à son type et du type à l'attribut

`put-type(`a, t_a`)`
`put-owner(`t_a, a`)`

3 Insertion effective du type dans son treillis

`insert-new-type(`T_a, t_a`)`

La procédure de propagation de la validation est identique, mis à part le fait qu'il n'y a plus de lien à mettre à jour entre le type à valider et une quelconque entité de représentation. Soit t le type sur lequel la validation est propagée. Il s'agit d'un δ-type créé par MeTéo suite à la création d'un δ-type construit : l'existence de t n'est due qu'au δ-type créateur. Si $t = \langle T; E(T)\rangle$, la procédure de propagation de validation sur t, notée `Propagate-valid(`t`)` est définie ainsi :

1 Propagation de la validation dans le cas d'un type construit

Si $T \leq_C$ `Construits` Alors $(E(T) = [type - ref; D_1; D_2; C^*])$
Pour tout $t_i \in type - ref$ faire
`put-depend(`t_i, t`)`
`Propagate-valid(`t_i`)`
FinSi

2 Insertion effective du type dans son treillis

`insert-new-type(`T, t`)`

Il s'agit d'une procédure de propagation récursive qui est assurée de terminer puisque les définitions cycliques sont interdites.

Validation du type d'une instance

La validation du type d'une instance revient à celle du type d'une classe, pour ce qui est de la création du lien bidirectionnel entre l'instance et son type, ainsi qu'en ce qui concerne bien sûr l'insertion de ce type. Toutefois, dans la phase d'activation de la validation au niveau des types des attributs, ces derniers ne correspondant à aucune entité explicitement représentée dans la base de connaissances, ils ne peuvent être attachés à la base, et sont donc considérés comme des types intermédiaires : ce n'est donc pas la procédure `Validate` qui est appelée sur la décomposition de l'instance en attribut, mais la procédure `Propagate-valid`.

4.4.8 Insertion d'un δ-type dans un treillis

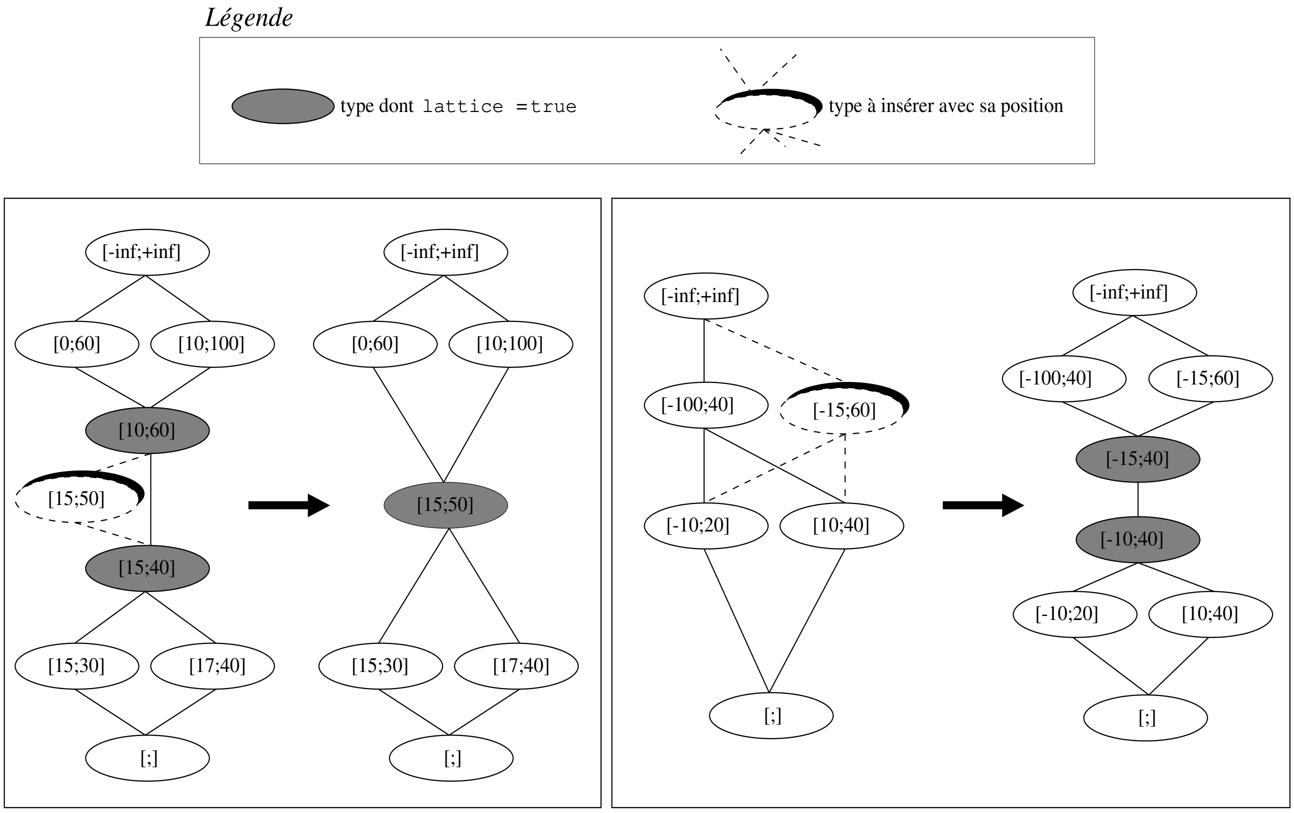
La validation d'un type $t = \langle T; E(T) \rangle$ entraîne nécessairement son insertion dans le treillis des δ-types issus de T, c'est-à-dire la recherche des liens de δ-sous-typage qui unissent t à tous les autres δ-types issus de T.

Avant le typage d'une base de connaissances, le treillis des δ-types de n'importe quel C-type T ne comporte que deux δ-types : $\top_T$ et $\perp_T$, le premier étant sur-type du second. L'insertion progressive de δ-types issus de la base de connaissances enrichit chaque treillis, de telle façon que ses propriétés soient respectées (section 3.4.4). Ainsi, la procédure d'insertion a pour effet de déterminer la position du type (procédure `search-type-position(`t`)`), et de réorganiser le treillis en vue de l'insertion effective de t. Ce remaniement du treillis comporte plusieurs étapes :

- mise à jour des liens `sur-type` et `sous-type` de t,
- mise à jour des liens `sur-type` (resp. `sous-type`) des sous-types (resp. sur-types) de t,
- réduction transitive du δ-sous-typage,
- introduction d'éventuels δ-types pour maintenir les propriétés structurelles du treillis,
- élimination d'éventuels δ-types qui étaient présents pour assurer la préservation des propriétés structurelles du treillis, et qui ne sont plus utiles du fait de l'introduction de t.

Ces deux dernières étapes de la réorganisation d'un treillis de δ-types sont illustrées par la figure 4.7. La dernière phase qui a pour effet d'éliminer des δ-types est une conséquence du fait qu'un treillis de δ-types ne doit comporter que des δ-types "utiles", c'est-à-dire dont l'existence provient d'une des trois raisons suivantes : il correspond au type d'une entité de représentation, il a été créé par un δ-type construit, ou il a été créé pour préserver la structure de treillis. Dans ce dernier cas, si l'insertion d'un nouveau δ-type remet en cause l'utilité d'un autre δ-type t, comme dans le cas (a) de la figure 4.7, t est supprimé du treillis. La section 5.5.4 détaille les conditions de suppression d'un δ-type et les conséquences qu'elle peut avoir.

La procédure `insert-new-type(`T, t`)` d'insertion effective d'un δ-type dans son treillis est fournie dans l'annexe C, ainsi que la procédure `search-type-position(`t`)` qu'elle utilise. Il est à noter que le type à insérer peut être déjà existant dans le treillis ; dans ce cas, seule la seconde étape est réalisée.



(a) insertion avec élimination de certains types *(b) insertion avec introduction d'un type supplémentaire*

FIG. 4.7 - : Remaniements de treillis de δ-types. La procédure d'insertion d'un δ-type doit, entre autres, examiner les conséquences de l'insertion au niveau des propriétés que la structure de treillis impose. Notons que lorsque la structure de treillis est maintenue par l'existence d'un δ-type non issu de la base de connaissances, sont calculés en réalité deux δ-types, l'un correspondant exactement au SUP des sous-types, l'autre à l'INF des sur-types ; en revanche, lorsqu'un δ-type issu de la base de connaissances permet de maintenir la structure de treillis, il n'est pas réalisé d'autre calcul de δ-type.

4.5 Un isomorphisme entre intensions et types

Les structures de données de METÉO et les opérations qui y sont définies permettent une représentation isomorphe de la vue intensionnelle d'une base de connaissances :

- à chaque élément (instance ou valeur) manipulé dans la base de connaissances, METÉO associe une valeur,
- à chaque entité descriptive (concept, classe, attribut), METÉO associe un δ-type, une forme normale de l'expression de l'ensemble de valeurs équivalent à celui dénoté par la description,
- à chaque constructeur d'ensemble d'éléments de TROPES, METÉO associe un constructeur de type,
- à chaque opération du modèle de connaissances traduisant l'interprétation des relations entre descriptions d'entités, METÉO associe une opération équivalente. En particulier, le sous-typage entre δ-types *records* (respectivement l'appartenance d'une valeur *record* à un δ-type *record*) est isomorphe à la spécialisation intensionnelle entre classes (respectivement à l'instanciation intensionnelle).

Plus formellement, nous avons établi un homomorphisme de l'algèbre $\mathcal{A}_L$ (modèle de connaissances) vers l'algèbre $\mathcal{A}_T$ (le système de types), et nous montrons que cet homomorphisme est bijectif.

L'algèbre intensionnelle est issue du formalisme de représentation des connaissances, elle permet l'interprétation intensionnelle des descriptions des entités d'une base de connaissances.

Définition 28 (Algèbre intensionnelle) *L'algèbre intensionnelle, notée $\mathcal{A}_L$, est définie à partir de sa signature*[7] *Σ_I et de son domaine d'interprétation $\mathcal{D}_I$, où*

- *$\mathcal{D}_I$ est le domaine de l'interprétation intensionnelle d'une base de connaissances,*
- *$\Sigma_I = \langle \mathcal{S}_I; \mathcal{F}_I \rangle$ avec :*
  - *$\mathcal{S}_I$ est un ensemble de sortes,*

$$\mathcal{S}_I = \{Entier_I, Reel_I, Symbole_I, Chaine_I, Liste_I, Booleen_I, Ensemble_I, Agregat_I\}$$

  - *$\mathcal{F}_I$ est un ensemble de fonctions,*

$$\mathcal{F}_I = \{d\text{-}entite, d\text{-}instance, \in_I, \leq_{\sigma,I}, \gg_I, inter\text{-}descr\}$$

Les sortes de cette algèbre sont celles à partir desquelles sont construites les entités de représentation, par application des opérations d-entite (description d'une entité) et d-instance (description d'une instance).

L'algèbre des types est issue du formalisme de MetéO, elle caractérise l'interprétation des termes du système de types dans le domaine de Scott.

Définition 29 (Algèbre des types) *L'algèbre des types, notée $\mathcal{A}_T$, est définie à partir de sa signature Σ_T et de son domaine d'interprétation $\mathcal{V}$, où :*

- *$\mathcal{V}$ est le domaine récursif de Scott,*
- *$\Sigma_T = \langle \mathcal{S}_T; \mathcal{F}_T \rangle$ avec :*
  - *$\mathcal{S}_T$ est un ensemble de sortes,*

$$\mathcal{S}_T = \{Entier_T, Reel_T, Symbole_T, Chaine_T, Liste_T, Booleen_T, Ensemble_T, Record_T\}$$

  - *$\mathcal{F}_T$ est un ensemble de fonctions,*

$$\mathcal{F}_T = \{\Delta T, VRecord, \in_\tau, \leq_\tau, \gg_\tau, INF\}$$

Les sortes de cette algèbre sont les C-types définis par MetéO, *sur lesquels sont construits des δ-types (par l'opérateur ΔT) et des valeurs* records *(opération $VRecord$).*

[7] *Une signature d'algèbre (sa syntaxe en quelque sorte) est une paire $\langle S; P \rangle$ où S est un ensemble (*de sortes*) et P est un ensemble (*de symboles de fonctions*) tel que P contient une application type : $P \rightarrow S^* \times S$. Pour chaque $f \in P$, la valeur type(f) est le* type *de f. Les constantes sont représentées par des symboles de fonctions d'arité nulle.*

Propriété 8 (Isomorphisme) *Il existe un isomorphisme $\Upsilon : \mathcal{A}_L \longmapsto \mathcal{A}_T$ défini comme suit :*

$$\Upsilon \begin{pmatrix} Entier_I \\ Reel_I \\ Symbole_I \\ Chaine_I \\ Booleen_I \\ Agregat_I \\ Liste_I \\ Ensemble_I \\ d\text{-}entite \\ d\text{-}instance \\ \in_I \\ \leq_{\sigma,I} \\ \gg_I \\ inter\text{-}descr \end{pmatrix} = \begin{matrix} Entier_T \\ Reel_T \\ Symbole_T \\ Chaine_T \\ Booleen_T \\ Record_T \\ Liste_T \\ Ensemble_T \\ \Delta T \\ V\,Record \\ \in_\tau \\ \leq_\tau \\ \gg_\tau \\ \sqcap_\delta \end{matrix}$$

En effet, nous avons montré dans ce chapitre que $\forall f_I^n \in \mathcal{F}_I$, d'arité n, $\exists f_T^n \in \mathcal{F}_T$ d'arité n tel que $\Upsilon(f_I^n) = f_T^n$. De plus, si $f_I^n : i_1, \cdots, i_n \rightarrow i \in \mathcal{D}_I$, alors :

$$\Upsilon(f_I^n(i_1, \cdots, i_n)) = f_T^n(\Upsilon(i_1), \cdots, \Upsilon(i_n))$$

Une conséquence directe de l'existence de cet isomorphisme est que l'interprétation intensionnelle d'une base de connaissances peut être déléguée au système de types, sur des structures de données plus adaptées. De ce fait, toutes les opérations qui sont effectuées sur les intensions des entités[8] pourront alors l'être sur les types/valeurs de ces entités, sans perte d'informations ni incohérence. La possibilité d'une telle délégation est une concrétisation des différences fondamentales qu'il existe entre intension et extension, et de leur coopération. En effet, si le modèle de connaissances est chargé de la définition des opérations extensionnelles et le système de types de la définition des opérations intensionnelles, la définition d'une opération mixte (ayant les deux composantes) se fait au niveau du modèle de connaissances, au moyen d'appels effectifs à des opérations du système de types.

La motivation de notre étude qui a abouti à la mise en évidence de l'isomorphisme Υ est, dans une certaine mesure, la même que celle qui a conduit Robert Dionne, Eric Mays et Frank Oles à introduire et formaliser l'*algèbre universelle des concepts* [DMO92][9]. Ils ont en effet réalisé une des premières tentatives de formalisation de la sémantique intensionnelle des systèmes à logique de descriptions, qui s'est avérée concluante, poursuivant les investigations de William Woods [Woo91] qui prônait une plus grande importance des interprétations intensionnelles. Le premier résultat de leur étude, reporté dans [DMO93], est d'avoir établi l'équivalence entre la subsomption structurelle (intensionnelle, calculée sur la base des descriptions abstraites des concepts) et la subsomption définie sur la théorie des modèles (extensionnelle, calculée sur la base de la donnée explicite des ensembles correspondant aux concepts). Une contribution majeure des résultats de leur étude est la prise en compte des cycles dans les définitions de concepts. Ce résultat reste pourtant propre aux hypothèses quant au statut des descriptions de concepts dans les langages terminologiques : la

[8] Ces opérations sont appelées *opérations intensionnelles.*

[9] Un terme de l'algèbre universelle des concepts correspond à la structure de construction des concepts et des rôles. Ces termes sont ordonnés par une abstraction de la subsomption structurelle telle que définie dans de nombreux systèmes à logique de descriptions.

description d'un concept, sauf s'il est primitif, est une définition de l'appartenance d'un individuel à ce concept, en conséquence, elle représente exactement ce que le concept dénote dans le monde modélisé.

L'objectif de cette étude, de même que la nôtre, est finalement de cerner les liens entretenus entre le raisonnement fondé sur les intensions et le raisonnement extensionnel, l'analyse de ces liens étant nécessaire à une définition précise de la sémantique des modèles de connaissances.

4.6 Conclusion

Nous avons montré dans ce chapitre qu'il existe un isomorphisme entre l'algèbre issue de l'interprétation intensionnelle du modèle de connaissances et l'algèbre issue de METÉO. En effet, le typage des entités de représentation respecte complètement l'interprétation intensionnelle des entités, de même que le γ-sous-typage correspond à la spécialisation, et que les conditions intensionnelles portées sur les passerelles sont traduites vers une opération de composition de sous-typage et de l'opération INF. Compte-tenu des résultats du chapitre précédent, à savoir que l'interprétation ensembliste de METÉO a pour co-domaine le domaine récursif de Scott, il existe, par transitivité, une fonction d'interprétation des composantes de représentation du modèle TROPES vers un univers de valeurs.

En conséquence, toute interprétation intensionnelle d'une entité ou d'une relation entre entités de représentation, peut être traduite vers METÉO, interprétée dans le domaine de Scott, et vice-versa. Nous allons, dans le chapitre suivant, montrer que cette correspondance facilite la mise en évidence de la coopération entre les interprétations intensionnelle et extensionnelle définies par le modèle, en illustrant le rôle joué par METÉO dans la réalisation de mécanismes d'exploitation des connaissances, et lors des mécanismes d'inférence : nous allons contribuer à analyser l'importance des inférences fondées sur les intensions par rapport aux inférences fondées sur les extensions.

Chapitre 5

Exploitations directes de Metéo

Par l'expression "exploitation directe de METÉO", nous entendons montrer l'utilisation faite, par le système de représentation, des δ*-types* en tant que *représentation des intensions des entités de représentation*, ainsi que du δ-sous-typage conçu pour être la relation isomorphe à la spécialisation intensionnelle.

Le développement de METÉO a été motivé en partie pour expliciter les différences qu'il existe entre intension et extension au niveau de la construction et de la gestion d'une base de connaissances à objets. Ce chapitre a pour objectif d'illustrer la coopération entre les deux interprétations des entités de représentation de la connaissance, au sein des principaux mécanismes d'un modèle de connaissances, par la mise en évidence, pour chacun d'eux de l'importance du rôle des types issus des entités de représentation, dans des mécanismes de raisonnement d'une part, et de vérification d'autre part. Cette étude passe donc naturellement par l'adaptation des mécanismes de vérification et d'inférence à l'existence de METÉO.

La première section de ce chapitre définit précisément l'interaction entre interprétations intensionnelle et extensionnelle, en fixant une sémantique de la coopération basée sur la révision. Les sections 5.2 et 5.3 s'attachent respectivement à étudier les mécanismes d'inférences que sont la classification d'instances et la classification de classes, dans le but de cerner précisément le rôle qu'y jouent les inférences intensionnelles. Dans le même ordre d'idées, la section 5.4 propose une exploitation du typage de *filtres* dans la conduite du mécanisme de filtrage. La section 5.5 concerne l'exploitation des inférences et vérifications de faits intensionnels dans le processus de gestion de la dynamique des représentations, pour lequel la recherche de cohérence et de correction est essentielle, dans la mesure où ce processus est essentiel dans la phase de construction d'une base de connaissances. Enfin, et avant de conclure, la section 5.6 définit le couplage entre le module de contraintes MICRO et METÉO, réalisé dans le but d'intégrer, au niveau des intensions donc des types, les informations descriptives véhiculées par la pose de contraintes.

5.1 Délégation des opérations intensionnelles

Concrètement, l'intégration de METÉO à TROPES se traduit par la délégation des opérations intensionnelles. On appelle *opération intensionnelle* toute opération ou processus dont l'exécution porte entièrement sur les descriptions des entités de représentation, indépendamment de la signification extensionnelle d'une telle manipulation. Par exemple, la vérification de la spécialisation intensionnelle est directement exécutable sur les types des entités, il s'agit d'une opération intensionnelle. Cependant, l'interprétation du résultat d'une opération intensionnelle faite par le modèle de connaissances traduit la coopération entre interprétations intensionnelle et extensionnelle.

5.1.1 Coopération intension/extension

Lorsque l'on considère qu'il est impossible de *définir*, en termes de propriétés arrêtées, des catégories de connaissances [Kay93], l'intension d'une catégorie ne peut être considérée comme équivalente à son extension. À partir de cette constatation, il s'agit pour un modèle de connaissances de définir la coopération des deux, à savoir la signification de l'une par rapport à l'autre.

Une mise en évidence de cette coopération est essentielle, ne serait-ce que pour tenter d'atteindre l'équivalence entre intension et extension lors du processus de construction d'une base de connaissances. Il s'agit d'informer l'utilisateur de l'importance que jouent les descriptions des entités, vis-à-vis de la caractérisation du monde modélisé. Il peut alors orienter ses descriptions pour que leurs dénotations se rapprochent le plus exactement possible du sens extensionnel des entités décrites[1].

La section 2.2.3 a défini pour TROPES la coopération entre interprétations intensionnelle et extensionnelle : la validation extensionnelle exige la validation intensionnelle. Pour qu'un fait puisse être établi dans TROPES, la validation extensionnelle est requise, et a fortiori la validation intensionnelle. Autrement dit, la cohérence d'une entité vis-à-vis de la sémantique de TROPES passe par la cohérence intensionnelle *et* extensionnelle.

- La cohérence intensionnelle est vérifiée automatiquement à partir des descriptions des entités, elle est donc déléguée à METÉO (de façon cohérente et complète, du fait de l'isomorphisme Υ).
- La cohérence extensionnelle est vérifiée par l'utilisateur : une assertion traduit la cohérence du fait "asserté" du point de vue extensionnel. Ensuite, certaines inférences cohérentes peuvent être faites à partir des extensions des relations entre entités. Par exemple, lorsqu'une instance I est rattachée à une classe C, cela signifie en premier lieu que I appartient à l'extension de C, et en conséquence, d'après l'interprétation extensionnelle de la spécialisation, que I appartient à l'extension de toutes les super-classes de C. Il est à noter que la validité de ces inférences passe par leur validité intensionnelle.

En ce qui concerne la vérification de la cohérence d'une base de connaissances, le fonctionnement du modèle de connaissances revient à l'exécution indépendante des deux processus (vérifications intensionnelle et extensionnelle), puis de la réunion des résultats, selon une sémantique guidée par la coopération établie entre les deux interprétations, et fondée sur la nécessité d'exploiter toute incohérence détectée.

La figure 5.1 illustre le schéma de fonctionnement du modèle vis-à-vis de la recherche simultanée de cohérence et de précision. Lors du test de cohérence d'une inférence ou de l'intégration d'une assertion, chacune des deux interprétations est prise en compte, le résultat de chacune est indiqué, dans le tableau, par un booléen. La considération simultanée des deux résultats détermine la validation définitive élaborée par le modèle ; elle est résumée dans les cases centrales du tableau. Lorsqu'il y a égalité des cohérences des deux interprétations, le modèle s'y accorde. Lorsqu'il n'y

[1] Il est d'ailleurs important de noter que l'expressivité d'un langage est sa capacité à permettre d'affiner les descriptions pour que, potentiellement, elles puissent être équivalentes à l'observation du monde modélisé, sans pour autant limiter la puissance des calculs effectués sur ces descriptions.

Consistance Intensionnelle

Consistance Extensionnelle	**0**	**1**
0	***Non-intégration*** *de la connaissance*	***Instauration d'un dialogue*** *proposition de modifications des descriptions d'une ou plusieurs des entités concernées* **But :** diminuer la distance entre intension et extension **Moyen :** éliminer la consistance intensionnelle
1	***Instauration d'un dialogue*** *proposition de modifications des descriptions d'une ou plusieurs des entités concernées* **But :** adapter l'intension à l'extension **Moyen :** obtenir la consistance intensionnelle	***Intégration*** *de la connaissance*

FIG. 5.1 - : Schéma de coopération intension/extension.

a pas égalité (désaccord reconnu entre intension et extension), le modèle tente d'aller en direction d'un accord, en proposant de modifier les descriptions des entités concernées. Dans le cas où il y a validation intensionnelle et pas validation extensionnelle, cela signifie qu'une description (ou plusieurs) est trop large, trop permissive (elle ne représente pas l'extension) : de façon globale, les descriptions ne "canalisent" pas suffisamment les extensions, il est alors proposé d'affiner les descriptions trop "laxistes". Dans le cas contraire, cela signifie qu'au moins une description est trop stricte, trop restrictive, qu'elle ne regroupe pas toute l'extension : il est alors proposé d'étendre les descriptions soupçonnées inadaptées.

5.1.2 Schéma général de la délégation

Tous les mécanismes élémentaires d'inférences et de vérification définis dans TROPES ont une composante intensionnelle (opérations intensionnelles) et une composante extensionnelle (opérations extensionnelles). Les opérations intensionnelles sont implémentées dans METÉO à partir des ses propres processus, qui opèrent directement sur les types et valeurs issus de la base de connaissances. L'exécution d'une opération intensionnelle est cependant appelée par le modèle de connaissances lui-même qui garde le contrôle sur la nécessité d'effectuer ou non des tests ou des inférences sur les intensions (figure 5.2).

Ce chapitre est consacré à illustrer, pour quelques mécanismes complexes de TROPES[2], parmi lesquels la classification d'instances, la classification de classes, le filtrage ou la gestion de contraintes, la séparation entre le raisonnement intensionnel et le raisonnement extensionnel, qui, habituellement, n'est pas très nette.

[2] On appelle mécanisme complexe tout processus qui est une compilation d'inférences et de vérifications plus élémentaires.

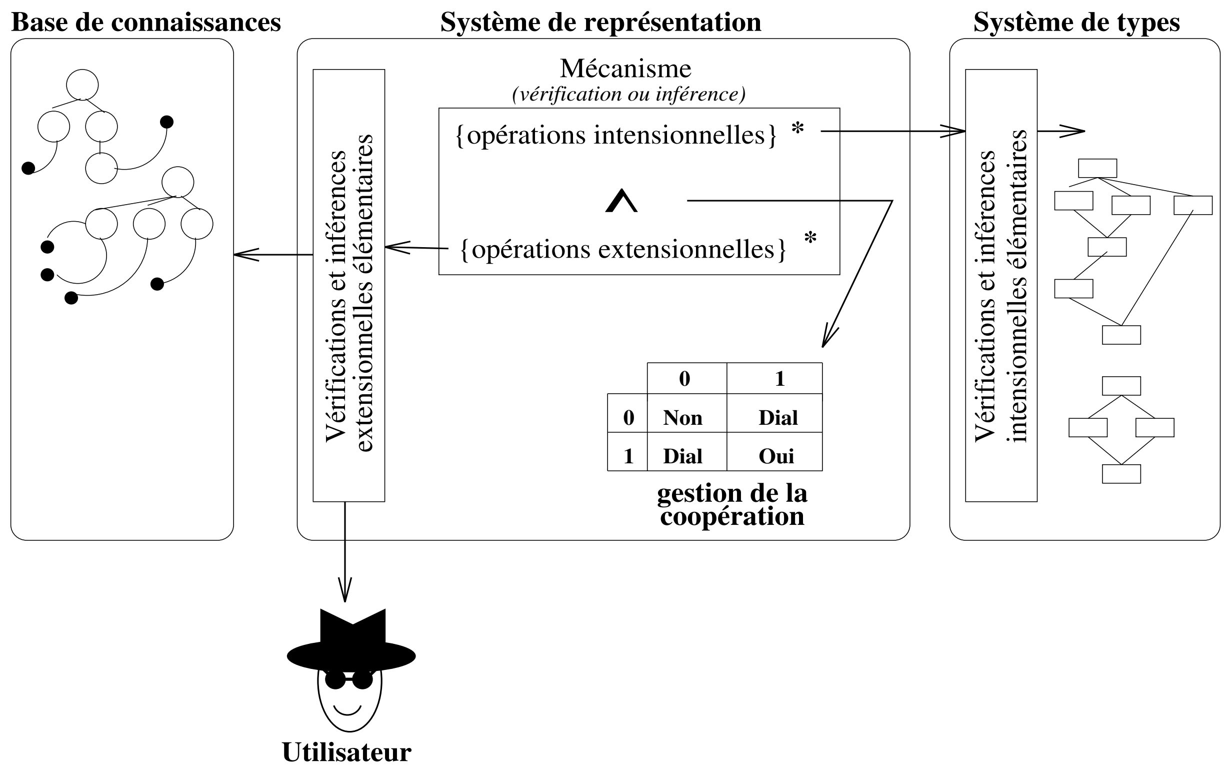


FIG. 5.2 - : Les processus intensionnels et extensionnels sont tout d'abord exécutés indépendamment, puis la sémantique de la coopération est considérée pour établir le résultat définitif du mécanisme (le symbole * traduit classiquement une séquence de cardinalité positive ou nulle).

5.2 Classification d'instances

Globalement, la classification d'instances est un mécanisme d'inférence de liens d'instanciation, qui peut avoir comme effet de bord la caractérisation précise de propriétés de l'instance inconnues avant l'activation du processus. La classification d'instances telle qu'elle a été définie par Olga Mariño dans TROPES comporte cinq phases (section 2.3.3) qui alternent des processus basés sur les intensions et d'autres basés sur le raisonnement extensionnel.

Le résultat final de la classification d'une instance doit être valide intensionnellement *et* extensionnellement. Soit `Classify-instance(K,I)` l'opération de classification d'une instance `I` dans un concept `K`. Outre un enrichissement éventuel de la base de connaissances, ce processus a pour résultat principal l'attachement de I à un ensemble de classes $C_1, \cdots, C_n$ de K (une par point de vue), tel que cet attachement soit cohérent selon les deux interprétations, c'est-à-dire :

$$\forall i, \|I\|^E \in \|C_i\|^E \text{ et } \|I\|^I \in \|C_i\|^I$$

Seule la vérification de la condition ci-dessus autorise la validation du résultat de la classification de l'instance I.

5.2.1 Interprétation mixte

Le rôle de la classification d'instances est d'inférer des liens d'instanciation intensionnelle les plus spécifiques possibles, susceptibles d'être validés extensionnellement. Ce n'est pourtant pas un simple positionnement de la description initiale de l'instance (donc un parcours de graphe), car la description de cette instance, éventuellement incomplète au départ, s'enrichit au fur et à mesure de la descente dans les arbres de spécialisation.

Actuellement, *au cours* de la classification d'une instance, l'intension et l'extension d'une même classe sont supposées équivalentes, c'est-à-dire qu'à partir du moment où l'instance est reconnue appartenir à l'intension d'une classe, il est décidé, par le système, qu'elle appartient à son extension, et réciproquement (la réciproque étant d'ailleurs toujours vraie), et ce uniquement durant le processus[3]. À l'issue du processus de classification, il sera demandé la validation extensionnelle pour fixer définitivement le lien d'instanciation. Cette hypothèse permet d'inférer des liens d'instanciation à partir de résultats intensionnels mais aussi extensionnels. Durant le processus de classification, plusieurs critères peuvent ainsi décider de l'appartenance d'une instance I à une classe C (ces critères sont résumés dans la figure 5.3) :

1. *appartenance intensionnelle stricte* ($\|I\|^I \in \|C\|^I$) : la description de I est en total accord avec celle de C. Ce résultat est généralement issu de la constatation d'une appartenance intensionnelle large suivie de l'obtention, par l'utilisateur, des informations manquantes qui sont à leur tour testées vis-à-vis de l'appartenance intensionnelle, jusqu'à ce que toutes soient fournies. Notons dans cette phase que les moyens d'obtention automatiques de valeurs, définis dans C (défauts, filtres ou activation de contraintes), ne peuvent être utilisés dans la mesure où leur utilisation nécessite l'établissement de l'appartenance de I à C[4]. Ce critère d'appartenance résulte d'une opération de vérification intensionnelle.

2. *appartenance de I à la source d'une passerelle menant à C ($I \in C_1, \cdots, C_n \gg C$)* : d'après la définition 7, cela signifie que I appartient à l'extension de C et a fortiori à son intension. Une telle inférence permet ensuite d'activer les moyens d'obtention de l'information définis dans C dans le but de compléter la connaissance sur I. Il s'agit d'un critère d'appartenance extensionnel dans la mesure où une passerelle porte initialement sur l'inclusion des extensions des classes mises en relation : les considérations intensionnelles ne portent que sur d'éventuels attributs communs à différents points de vue, ce qui confère à l'inférence intensionnelle de passerelles un caractère hasardeux et de ce fait trop coûteux.

3. *impossibilité de l'appartenance de I aux classes sœurs de C* : d'après les hypothèses d'exclusivité et d'exhaustivité (section 2.2.4), on déduit que $I \in C$. Il s'agit là aussi d'une déduction extensionnelle dans la mesure où ces propriétés ne peuvent qu'être établies extensionnellement : leur interprétation intensionnelle ne permet d'établir qu'une vérification très relative de la validité extensionnelle.

4. *appartenance de I à une sous-classe de C* : on en déduit, intensionnellement *et* extensionnellement que $I \in C$. Cependant, il s'agit plus ici d'une propagation que d'un réel apport d'information quant à la description de l'instance. Il est important de noter que de la même façon, l'établissement d'une non-appartenance de I à une classe C propage la non-appartenance de I à toutes les sous-classes de C, ce qui est un apport d'informations extensionnelles.

Au vu de ces quatre critères d'inférences de liens d'instanciation, il est immédiat de constater que le rôle des critères intensionnels se limite au test d'appariemment (strict ou large) d'une instance

[3] Cette hypothèse peut bien sûr être levée en demandant à l'utilisateur son accord (ou son désaccord) lors de chaque possibilité d'appartenance extensionnelle, fondée sur une appartenance intensionnelle. Cela permettrait, justement en cas de désaccord de l'utilisateur, de remettre en cause la précision relative de la description de la classe. Un tel processus répété sur une même classe permettrait au fur et à mesure d'affiner la description de la classe, avec l'aide de l'utilisateur, pour tendre vers une définition. Ceci n'est pourtant pas l'objet de mes travaux, mais pourrait être étudié dans le cadre des recherches sur la validation des connaissances.

[4] Le détachement procédural peut être utilisé si l'appartenance intensionnelle requise pour son activation est vérifiée (section ??).

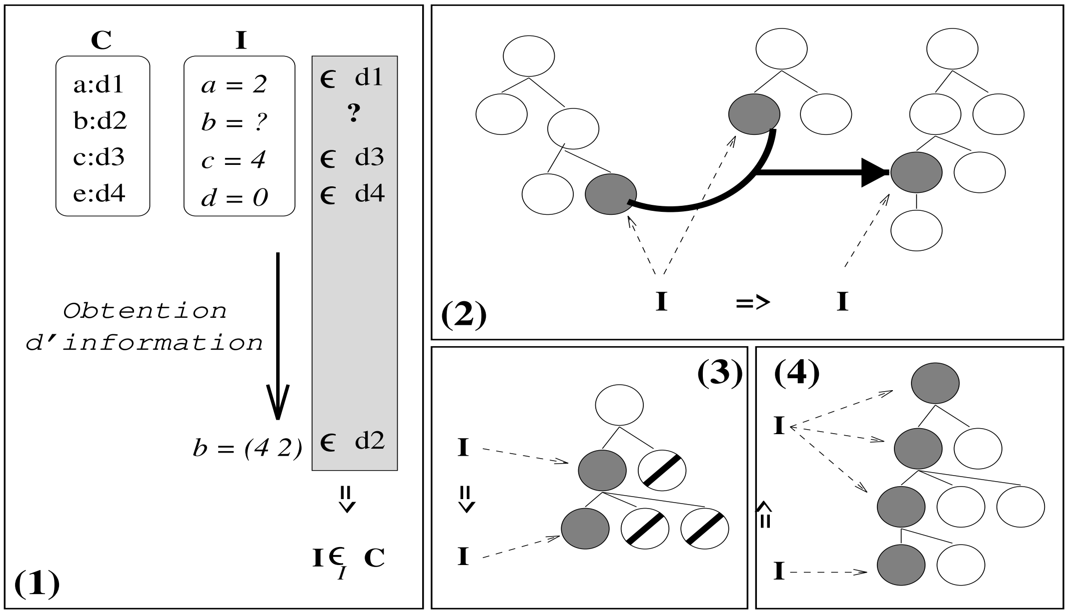


FIG. 5.3 - : Les quatre moyens de déduction d'un lien d'instanciation : appariemment (1), passerelle (2), exclusivité/exhaustivité (3), et spécialisation (4). Tous ces moyens d'inférences sont considérés lors de la classification d'une instance.

à une classe à partir de leurs descriptions respectives. En ce sens, la classification d'une instance ne peut se faire rapidement et complètement si l'on ne considère que les déductions sur les descriptions. En conséquence, il n'est pas envisageable de déléguer entièrement ce processus au système de types. *Seul l'appariemment peut l'être entièrement , en testant l'appartenance de la valeur* record *associée à I au type* record *associé à C*. Le test d'appariemment est une fonction qui rend un résultat correspondant à la possibilité d'instanciation d'une instance I à une classe C, pris parmi les trois valeurs 'vrai', 'faux' et 'possible'[5]. Cette fonction est intégrée au processus général de la façon suivante :

Test d'appariemment : $\|I\|^I \in \|C\|^I$ **?**

`Check-membership`(I, C)
Début
  Si `compute-value(I)` $\in^l_{Record}$ `get-type(C)` (appartenance large)
    Alors
      Début
        appartenance $\longleftarrow$ 'vrai'
        E $\longleftarrow$ {attributs de C non valués par I }
        Tant que (appartenance $\neq$ 'faux') et (E $\neq \emptyset$) Faire
          A $\longleftarrow$ head(E)
          V $\longleftarrow$ `Search-value`(A,I)
          Si V = *unknown*
            Alors appartenance $\longleftarrow$ 'possible'
            Sinon
              Si `compute-value`(V) $\in^s_\tau$ `get-type`(A) (appartenance sricte)
                Alors E $\longleftarrow$ tail(E)
                Sinon appartenance $\longleftarrow$ 'faux'

[5]Il s'agit d'une logique à trois valeurs telles que $b_1 \wedge b_2 = \mathrm{Min}(b_1, b_2)$, $b_1 \vee b_2 = \mathrm{Max}(b_1, b_2)$, à partir de l'ordre suivant : 'vrai' > 'possible' > 'faux'.

FinSi
FinSi
FinTantQue
Fin
Sinon appartenance ⟵ 'faux'
FinSi
⟶ appartenance
Fin

`Search-value`(A, I) est une fonction qui cherche à déterminer la valeur de l'attribut A pour l'instance I, en utilisant tous les moyens d'obtention possibles (simple récupération dans le schéma de l'instance, détachement procédural, sollicitation de l'utilisateur, valeur par défaut, etc.). Le critère intensionnel pour l'évaluation d'une instanciation est donc examiné par l'exécution de ce processus[6], dont le contrôle est assuré par le mécanisme général de classification d'instance.

Nous allons pourtant voir, dans ce qui suit, que le test d'appariemment intensionnel n'est pas toujours des plus adaptés, notammment en présence d'objets complexes dans les descriptions.

5.2.2 Classification d'objets complexes

La classification d'une instance I nécessite parfois, lors du test d'instanciation intensionnelle, de classer une autre instance I' dans un autre concept (I' est la valeur d'un attribut a de I) pour vérifier que I' respecte les conditions décrivant a. En particulier, ces conditions peuvent porter sur le test d'appartenance de I' à un ensemble de classes $C_1, \cdots, C_n$ imposées par a pour ses valeurs.

Une telle vérification revient, dans le cas où I' n'est pas une instance de $C_1, \cdots, C_n$, à descendre simultanément I vers ces classes, pour prouver ou réfuter l'appartenance requise (impliquant au niveau de I la réfutation de son appartenance à C ou la poursuite du processus d'appariemment de I à C). Il s'agit donc d'enclencher un autre processus de classification, de I' cette fois-ci, dans le contexte de la classification de I. Actuellement, cela génère, au niveau de l'algorithme de classification, un appel récursif de la classification sur la nouvelle instance. Pourtant, il ne s'agit pas exactement du même processus dans la mesure où dans ce cas, *une classe cible est fournie*. En ce sens, avant de déclencher le nouveau processus de classification, un test d'appariemment de I' aux classes requises est préalablement effectué : en cas d'appartenance stricte ou de non-appartenance, il n'est pas utile de lancer le processus de classification, sinon il doit l'être afin de trancher entre appartenance possible (large) et non-appartenance.

Par ailleurs, lorsque sont présents des objets complexes (ce qui est le cas en TROPES), un test d'appariemment exclusivement intensionnel n'est pas des plus efficaces, et de surcroît pas forcément complet : le test d'appariemment d'une instance i vers une classe C peut nécessiter le test d'appriemment d'une autre instance i' (valeur d'un attribut de i) vers une classe C' (type de cet attribut dans la description de C). D'une part, $i' \in_I C'$ peut ne pas être décidée (ou tout au moins seulement de façon large) du fait de l'éventuelle incomplétude de la description de i', alors que $i \in_E C'$ est établie : dans un tel cas, l'incomplétude du test intensionnel peut être levée par un test extensionnel. D'autre part, Il est inutile de tester un appariemment intensionnel de i' vers C' quand il est établi, extensionnellement, que $i \notin_E C'$: cette information peut être obtenue, par exemple, de par l'hypothèse d'exclusivité (i' est attachée à une classe sœur ou nièce de C').

[6]La phase d'appariemment a été ici largement simplifiée, dans le sens où elle ne rapporte pas les optimisations effectuées par le mécanisme général qui sont indépendants du test d'appartenance intensionnel en lui même.

De ce fait, il est pertinent, lors du test d'appariemment d'une instance vers une classe, d'introduire des tests sur les extensions des entités.

L'algorithme ci-dessus est modifié de façon à prendre en considération les attributs complexes pouvant mener à des classifications successives.

Test d'appariemment : présence d'objets complexes

`Check-membership`(I, C)
Début
Si (app-ext = `Check-ext-membership`(I, C)) $\neq$ 'possible'
  Alors appartenance $\longleftarrow$ app-ext
  Sinon
    Si `compute-value`(I) $\in^l_{Record}$ `get-type(C)` (appartenance large)
      Alors
        Début
          appartenance $\longleftarrow$ 'vrai'
          E $\longleftarrow$ {attributs de C non valués par I }
          Tant que (appartenance $\neq$ 'faux') et (E $\neq \emptyset$) Faire
            A $\longleftarrow$ head(E)
            V $\longleftarrow$ `Search-value`(A,I)
            Si V = *unknown*
              Alors appartenance $\longleftarrow$ 'possible'
              Sinon Si `compute-value`(V) $\in^s_\tau$ `get-type`(A) (appartenance sricte)
                Alors E $\longleftarrow$ tail(E)
                Sinon Si (`compute-value`(V) $\in^l_\tau$ `get-type`(A)) et (`instance-p`(V))
                  Alors appartenance $\longleftarrow$
                    `Classify-instance`(`dom`(A),V)
                    $\wedge$ (`compute-value`(V) $\in^s_\tau$ `get-type`(A))
                    (cette seconde condition est nécessaire car V doit non seulement appartenir aux classes spécifiées dans A mais aussi au type complet de A
                  Sinon appartenance $\longleftarrow$ 'faux'
                FinSi
              FinSi
            FinSi
          FinTantQue
        Fin
      Sinon appartenance $\longleftarrow$ 'faux'
    FinSi
FinSi
$\longrightarrow$ appartenance
Fin

Où :

- `dom`(A) désigne la spécification de A donnée par les descripteurs de typage *principal* (en particulier, dans le cas d'un attribut complexe, il s'agit d'un concept ou d'une liste de classes prises dans des points de vue distincts),
- `instance-p`(O) est une fonction qui teste le fait que l'entité de représentation O est une instance de concept.

Le fait de confier à MÉTÉO le test d'appartenance de la valeur de l'attribut à son domaine de valeur est un gain en efficacité, puisque des tests extensionnels sont effectués au préalables, et dans la mesure où les types des classes cibles sont sous forme normale (réduction du nombre de descripteurs et normalisation) et contiennent déjà les résultats du mécanisme d'héritage : le coût de l'héritage dynamique est donc nul, contrairement à un test d'appartenance effectué directement sur les structures des classes.

5.2.3 Classification d'instances incomplètes

Dans l'algorithme de classification actuel, seules les valeurs connues sont prises en compte dans la phase d'acceptation de l'appartenance intensionnelle à la classe testée (appartenance stricte), et lorsque des valeurs demeurent inconnues à l'issue de la phase d'obtention, le processus conclut sur une *possibilité* d'acceptation, et la descente dans l'arbre de spécialisation est stoppée momentanément même si elle peut s'avérer pertinente.

Une solution proposée pour éviter l'arrêt de la classification dû à une instance incomplète, est d'activer dans ce cas un raisonnement hypothétique, qui consiste à créer des versions de l'instance en fonction des valeurs possibles pour l'attribut manquant, de lancer la classification de ces versions hypothétiques, et de détecter simultanément des incohérences qui permettent d'eliminer des hypothèses [Gir95]. Malgré des techniques d'optimisation et des possibilités de parallélisation, ce type de solution peut mener à une explosion combinatoire.

Nous proposons une alternative qui consiste, en l'absence d'une valeur, à solliciter l'utilisateur pour qu'il fournisse, *au moins*, un domaine de valeurs plus restreint que celui connu pour un attribut de l'instance. Il s'agit d'une information qui demeure incomplète mais qui affine tout de même la connaissance que l'on a sur l'instance et qui *peut* suffire à poursuivre la classification.

Pour cela, et pour l'attribut dont on affine le domaine, le test d'appartenance est remplacé par un test d'inclusion, c'est-à-dire un test de sous-typage, voire une recherche de lien de spécialisation lorsqu'il s'agit d'un attribut complexe. Si le test s'avère positif, cela est naturellement interprété comme une appartenance, donc le processus de classification peut se poursuivre. Si le test mène à une réfutation, la disjonction entre les domaines de valeurs (celui requis et celui spécifié, il peut s'agir de classes) est testée : si elle est établie, cela signifie que la valeur de l'attribut pour l'instance n'appartient pas au domaine requis, la classe testée pour l'instance est alors marquée *impossible* et le processus de classification se poursuit. Si la disjonction des domaines n'est pas établie, l'apport d'information sur l'attribut d'instance n'est pas suffisant et le processus s'arrête, la classe est marquée *possible*.

En cas d'attribut complexe, le test de sous-typage (spécialisation intensionnelle) peut être complété par un test de spécialisation extensionnelle.

L'algorithme précédent est alors modifié pour intégrer ce traitement de l'information incomplète mais affinée : nous ne précisons ci-dessous que la phase d'obtention d'information et le traitement immédiat qui lui est associé. Soit I l'instance à classer, soit a l'identificateur de l'attribut dont la valeur est inconnue pour I, pour lequel l'utilisateur associe donc un domaine restreint à l'aide des descripteurs classiques de restriction de domaines. Dans l'algorithme ci-dessous, ce domaine restreint est noté $\tilde{a}$. Le principe est alors de créer un attribut virtuel A_I de domaine $\tilde{a}$, de typer cet attribut et de réaliser les tests intensionnels en le comparant au type de cet attribut dans la classe, noté A_C. Le résultat de la fonction ci-dessous est représenté par la variable "appartenance".

Étape de classification d'une instance à partir du domaine d'un de ses attributs

```
Check-instanceslot-adaptation(A_C, A_I)
Début
   t(A_I) ⟵ Typing-b(A_I)
   Choix Selon
         • t(A_I) ≤_τ get-type(A_C)
                        appartenance ⟵ 'vrai'
         • get-type(A_C) ∩t(A_I) = ⊥
              appartenance ⟵ 'faux'
         • A_C est un attribut complexe, soit dom-ext(A) les entités
         de représentation dans l'extension desquelles A prend sa valeur
              Choix Selon
                     • Check-extension-inclusion(dom-ext(A_I),dom-ext(A_C)) = true
                          appartenance ⟵ 'vrai'
                     • Check-extension-disjunction(dom-ext(A_I),dom-ext(A_C)) = true
                          appartenance ⟵ 'faux'
                     • Autres Cas
                          appartenance ⟵ 'possible'
              FinChoixSelon
         • Autres Cas
              appartenance ⟵ 'possible'
   FinChoixSelon
   ⟶ appartenance
Fin
```

Où `Check-extension-inclusion(`L_1`,`L_2`)` et `Check-extension-disjunction(`L_1`,`L_2`)` sont deux fonctions qui vérifient respectivement si l'extension commune des éléments de L_1 (des entités de représentation) est incluse dans (disjointe de) l'extension commune des éléments de L_2.

Cette fonction comporte une partie de raisonnement intensionnel, suivi éventuellement d'une recherche d'inférences extensionnelles dans le cas d'attributs complexes[7]. Le résultat de cette fonction ('vrai', 'faux' ou 'possible') est pris en compte, dans la procédure de test d'appariemment de l'instance à la classe testée, de la même façon que le résultat d'appartenance d'une valeur de l'instance au type de l'attribut correspondant. Il s'agit d'une fonction interne au processus de classification d'instance défini par le système de représentation.

Cet ajout dans la phase de traitement de l'information obtenue sur l'instance en cours de classification est un enrichissement du processus de classification, puisqu'il traite la notion d'information partielle, alternative entre information complète (la valeur est donnée) et information nulle (la valeur est inconnue), et ceci à moindre coût. Pour cela, nous avons introduit la notion d'inférence de lien de sous-typage et de spécialisation, qui sera plus largement abordée dans la section suivante.

Cet algorithme illustre clairement la séparation entre le raisonnement intensionnel (confié à MÉTÉO) et le raisonnement extensionnel (interne au modèle de connaissances), dont les conduites sont parfaitement indépendantes une fois que la coopération est établie par le modèle. En outre, les vérifications et tests qui sont faits sur les structures MÉTÉO issues des entités sont plus efficaces dans la mesure où les expressions de domaine sont déjà normalisées et que l'héritage a déjà été compilé lors de la phase de typage.

[7] Nous n'avons pas traité dans cet algorithme les inférences extensionnelles dans le cas d'attributs complexes multi-valués, et ce par souci de clarté : les tests de cardinalités compatibles s'effectuent sur les types de ces attributs.

5.3 Classification de classes

Lors de la construction des arbres de spécialisation d'une base de connaissances, réalisée par ses concepteurs, l'introduction d'une nouvelle classe d'individus peut s'avérer délicate même si sa description est connue : les contraintes intensionnelles de la spécialisation rendent difficile la recherche de la position d'une description de classe, et c'est pour cela que nous introduisons le processus de *classification d'une classe*, dont le but est de déterminer la position d'une classe dans une hiérarchie ordonnée par la relation de spécialisation, et ce à partir de la description *en termes d'attributs* de la classe.

5.3.1 Problématique

Le rôle de la classification de classes est similaire à celui de la classification de concepts définis, à savoir la recherche de liens de subsomption. Dans Tropes, il s'agit de déterminer la position d'une classe dans la hiérarchie de spécialisation. Comme tout processus opérant sur les entités de représentation, celui-ci a une interprétation mixte :

- déterminer des liens d'inclusion ensembliste entre l'*extension* de la classe et celles des classes existantes dans la hiérarchie,
- déterminer des liens d'inclusion ensembliste entre l'*intension* de la classe et celles des classes existantes, autrement dit inférer des liens de spécialisation intensionnelle, donc finalement des liens de sous-typage.

Dans Tropes, parler de la classification d'une *classe* est toutefois ambigu, car une classe traduit le regard porté sur un ensemble d'individus *sous un certain point de vue*. De ce fait, la classification d'une classe revêt des caractères différents selon qu'elle est effectuée dans un seul point de vue ou dans plusieurs simultanément.

- La classification d'une classe dans un seul point de vue est exactement la recherche de liens de spécialisation entre la classe et les autres du même point de vue, toutes étant caractérisées à partir d'un même ensemble de propriétés, celui du point de vue.
- La classification dans plusieurs points de vue n'est pas exactement la classification d'une *classe* en tant que telle, mais plutôt la classification d'un ensemble d'individus qui partagent certaines propriétés connues. Il s'agit donc de la caractérisation, pour cet ensemble d'individus, de la position de leurs classes communes d'appartenance *sous chaque point de vue*. Autrement dit, la classification d'une classe dans plusieurs points de vue équivaut, dans l'absolu, à classer autant de classes qu'il y a de points de vue dans le concept.

Nous proposons dans cette section d'étudier, pour ces deux catégories de classification, des algorithmes de positionnement basés principalement sur les interprétations intensionnelles des classes et des relations, de telle façon que la classification d'une classe soit effectuée par la classification de sa description. Les considérations extensionnelles interviennent dans les phases de validation, mais aussi dans certaines phases d'inférences, plus particulièrement dans la classification multi-points de vue.

5.3.2 Classification d'une description de classe dans un point de vue

La classification d'un élément dans un graphe revient au parcours d'un ensemble ordonné par l'ordre sous-jacent au graphe. Nous pouvons imaginer trois schémas de classification d'une description de classe dans l'arbre de spécialisation d'un point de vue, qui se distinguent par l'ensemble ordonné sur lequel est effectué ce parcours :

- la classification est effectuée par un parcours de l'arbre de spécialisation, selon le même principe que la classification d'une instance, lors duquel la spécialisation intensionnelle (le sous-typage) est systématiquement testée entre la classe courante et celle à classer,
- la classification est effectuée par un parcours de la hiérarchie des types de classes, lors duquel les liens de sous-typage entre le type courant et le type de la classe à tester sont testés,
- la classification s'effectue par un parcours simultané des différents treillis de types des attributs de la classe, menant à l'insertion dans chacun de ces treillis du type de l'attribut de la classe à insérer, puis par intersection des positions de la classe relatives à chaque attribut [CC93].

Les deux premières solutions sont algorithmiquement proches, même si, dans la seconde, le processus est entièrement conduit par MetÉo, c'est-à-dire à partir de considérations essentiellement intensionnelles, alors que dans la première, il oscille entre des consultations de connaissances extensionnelles *et* intensionnelles. Parmi ces deux solutions, la seconde est donc plus adaptée dans la mesure où toutes les considérations extensionnelles liées à la spécialisation sont représentées intensionnellement[8]. En particulier, la spécialisation extensionnelle est complètement représentée par le sous-typage entre les types de classe, de même que la propriété d'exclusivité.

Toutefois, nous retiendrons la troisième solution pour la classification d'une classe dans un point de vue, parce qu'elle permet la mise en évidence de liens de spécialisation intensionnelle projetés sur un ou plusieurs attributs (résultat partiel de classification), menant à une interactivité volontaire dont le but est d'aider le concepteur à corriger le plus tôt possible la spécification d'un attribut, lorsqu'il se rend compte qu'un résultat partiel n'est pas cohérent avec ce qu'il attendait, c'est-à-dire que la spécialisation intensionnelle ne vérifie pas, ou ne reflète pas, la spécialisation extensionnelle. Une telle interactivité dans le processus de construction d'une base de connaissances est nécessaire à la justesse des connaissances acquises, que ce soit au niveau des connaissances individuelles ou catégorielles.

La solution que nous proposons consiste donc à inférer des liens de spécialisation intensionnelle à partir des liens de δ-sous-typage sur les types d'attributs. Le principe de cette méthode est, en ce sens, similaire à l'approche réalisée par Christiaan Thieme et Arno Siebes qui ont proposé, dans le domaine des bases de données à objets, un algorithme pour l'intégration de schémas de classes fondé sur la définition syntaxique des classes en termes d'attributs et de méthodes [TS93].

Principe de l'algorithme

Le schéma général de la classification d'une description de classe par les types de ses attributs consiste à considérer la description de la classe comme une séquence de descriptions d'attributs. Lors de la donnée de la description d'un attribut, le processus positionne la classe relativement à cet attribut, puis confronte cette position avec celles obtenues pour les autres attributs déjà décrits,

[8] mise à part la propriété d'exhaustivité qui ne peut toutefois pas être prise en compte puisque la classification d'une classe s'effectue lors de la construction d'une base de connaissances, qui est a fortiori incomplète.

afin d'émettre un résultat partiel, relatif à tous les attributs déjà spécifiés, indépendamment des autres à venir. À la vue d'un résultat partiel, le concepteur de la classe a deux options : continuer la description de la classe (il accepte le résultat partiel), ou revenir sur la description d'un attribut déjà considéré (généralement le dernier), lorsqu'il estime que le résultat partiel n'est pas satisfaisant.

Soit $A(C)$ l'ensemble des identificateurs d'attributs qui participent à la description de la classe C. Soit $\tilde{A}(C)$ l'ensemble des couples attribut/domaine donnés dans la description de la classe C. Dans le cas d'une description incrémentale de la classe, cet ensemble est initialisé par les attributs qui sont repris dans C par référence à d'autres descriptions. Dans le cas d'une classification "brute", c'est-à-dire lorsque la description de C est vide au départ, $A(C)$ est initialisé à $\emptyset$[9]. On note $i = id(\tilde{a}_i)$ l'identificateur de l'attribut $\tilde{a}_i$, Le résultat partiel de la classification est noté $\langle sp; ss\rangle$, où sp contient les super-classes et ss les sous-classes possibles. La notation $\langle sp_i; ss_i\rangle$ dénote le résultat partiel de la classification relativement à l'unique attribut identifié par i : ce résultat est élaboré par la fonction `search-slottype-position` qui est détaillée plus en avant. Le schéma de l'algorithme de classification d'une classe, défini au niveau du modèle de connaissances, est alors le suivant :

Calcul de la position structurelle d'une classe C dans un point de vue PV d'un concept K
Traitement principal

Début
  fini $\longleftarrow$ faux
  $i \longleftarrow 1$
  $A(C) \longleftarrow \emptyset$
  $\langle sp; ss\rangle \longleftarrow \langle$ `get-classes-pv(`K, PV`)`; `get-classes-pv(`K, PV`)` $\rangle$
  Tant que (non fini) faire
    $\tilde{a}_i \longleftarrow$ `get-specif(`i, C`)` (acquérir la description d'un attribut $\tilde{a}_i$)
    $\langle sp_i; ss_i\rangle \longleftarrow$ `search-slottype-position(`$\tilde{a}_i, PV$`)`
    $\langle sp; ss\rangle \longleftarrow \langle sp \cap sp_i; ss \cap ss_i\rangle$
    Si $\langle sp; ss\rangle$ est accepté par le concepteur
      Alors $i \longleftarrow i + 1$; $A(C) \longleftarrow A(C) \cup \{i\}$
      Sinon
        modification $\longleftarrow$ true
        Tant que (modification = true) faire
          $k \longleftarrow$ `undo-slot()`
          Si $(k \neq 0)$ et $(k \in A(C))$
            Alors $A(C) \longleftarrow A(C) \backslash \{k\}$
              $\langle sp; ss\rangle \longleftarrow \langle(\bigcap_{j \in A(C)} sp_j); (\bigcap_{j \in A(C)} ss_j)\rangle$
            Sinon $k \longleftarrow$ `integrate-slot()`
              Si $(k \neq 0)$ et $(k \notin A(C))$
                Alors $\langle sp; ss\rangle \longleftarrow \langle sp \cap sp_k; ss \cap ss_k\rangle$
                  $A(C) \longleftarrow A(C) \cup \{k\}$
              FinSi
          FinSi
          modification $\longleftarrow$ `another-modif-to-do()`
        FinTantQue
    fini $\longleftarrow$ `end-classif?()`
  FinTantQue
  Fin

[9] C'est le cas que nous considérons dans l'algorithme ci-dessous par soucis de clarté, mais cela ne modifie en rien le principe de la classification de C.

Où :

- `get-classes-pv`(K, PV) est une fonction qui a pour résultat l'ensemble des classes du concept K sous le point de vue PV,
- `get-specif`(i, C) est une fonction qui a pour résultat la spécification de l'attribut d'identificateur i dans la classe C (fonction interactive),
- `undo-slot()` est un fonction qui a pour résultat l'identificateur d'un attribut à enlever de la description de C (fonction interactive)[10],
- `integrate-slot()` est une fonction qui a pour résultat l'identificateur d'un attribut à rajouter dans la description de C, la spécification de l'attribut possédant cet identificateur ayant déjà été donnée pour la classe en cours de classification (fonction interactive),
- `another-modif-to-do()` est une fonction qui à pour résultat un booléen, vrai si l'utilisateur souhaite encore modifier $A(C)$, faux sinon (fonction interactive).
- `search-slottype-position`(a, PV) a pour résultat un couple dont le premier élément est *l'ensemble des classes du point de vue PV qui possèdent ou héritent un attribut d'identificateur* $id(\tilde{a})$, *dont le type est supérieur à celui de* $\tilde{a}$, et le second élément est symétriquement *l'ensemble des classes du point de vue PV qui possèdent ou héritent un attribut d'identificateur* $id(\tilde{a}$, *dont le type est inférieur à celui de* $\tilde{a}$. Sa réalisation comporte quatre étapes :
  1. typage de la nouvelle spécification de l'attribut dans C, soit $t = \langle T; E(T) \rangle$,
  2. recherche de la position de t dans le treillis des attributs de T, soit $P = \langle spt; sst \rangle$, couple formé de l'ensemble de tous les super-types et de l'ensemble de tous les sous-types de t (ceci est effectué par un processus dont l'algorithme est similaire à celui de l'insertion d'un δ-type dans son treillis, exception faite de l'insertion effective menant à la mise à jour des liens de dépendances),
  3. réduction des deux éléments de P, qui consiste à en ôter les types qui ne correspondent pas à un typage de $id(\tilde{a})$ (par analyse du contenu du lien `owner` de chaque δ-type), et élaboration simultanée du résultat final, selon le procédé qui consiste à associer à chaque type t de P la (les) classes du point de vue PV dont la description contient $id(\tilde{a})$ auquel le type t est associé. Dans l'algorithme ci-dessous, L_{sp} et L_{ss} désignent la position de l'attribut $\tilde{a}$ donnée par les listes des attributs de même identificateur dont le type est respectivement un sur-type et sous-type de celui calculé pour $\tilde{a}$; on en déduit sp et ss qui désignent la position de $\tilde{a}$ donnée par les listes de classes qui définissent les attributs de L_{sp} et L_{ss}.
  4. complétion de sp (la liste des super-classes possibles), à laquelle on ajoute toutes les classes qui ne décrivent ni n'héritent a.

Recherche de la position d'un type d'attribut, `search-slottype-position`($\tilde{a}, PV$)
Début

[10] Dans le cas où lors de la classification d'une classe, l'utilisateur demande à enlever la spécification d'un attribut, nous pouvons imaginer que généralement il s'agit du dernier attribut spécifié (aucun autre n'a été (*ré*)-intégré entre temps), c'est-à-dire celui dont l'introduction a modifié la position partielle. Dans ce cas, l'algorithme principal peut être, du point de vue de la complexité-temps, amélioré : le calcul de la position suite au retirement du dernier attribut spécifié est le suivant : $\langle sp; ss \rangle \longleftarrow \langle sp_i^p; ss_i^p \rangle$, où $\langle sp_i^p; ss_i^p \rangle$ est la position partielle de la classe avant l'introduction de l'attribut identifié par i, sauvegardée juste après la prise en compte de la position relative à $\tilde{a}_i$, par intersection avec la position partielle. On évite ainsi le recalcul de toutes les intersections des positions relatives à chacun des attributs.

$t(a) \longleftarrow$ `Typing-b`$(\tilde{a})$
$\langle spt; sst\rangle \longleftarrow$ `search-type-position`$(t(a))$
$sp \longleftarrow \emptyset$
$ss \longleftarrow \emptyset$
$L_{sp} \longleftarrow \emptyset$
$L_{ss} \longleftarrow \emptyset$
Pour tout $t \in spt$
    $L_{sp} \longleftarrow L_{sp} \bigcup \{\tilde{a_{sp}} \in \texttt{get-owner}(t) \mid id(\tilde{a_{sp}}) = id(\tilde{a})\}$
    Pour tout $\tilde{a_{sp}} \in L_{sp}$
        $sp \longleftarrow sp \bigcup \{C_{sp} \mid \texttt{get-classslot}(id(\tilde{a_{sp}}), C_{sp}) = \tilde{a_{sp}}\ \}$
    FinPour
FinPour
Pour tout $t \in sst$
    $L_{ss} \longleftarrow L_{ss} \bigcup \{\tilde{a_{ss}} \in \texttt{owner}(t) \mid id(\tilde{a_{ss}}) = id(\tilde{a})\}$
    Pour tout $a_{ss} \in L_{ss}$
        $ss \longleftarrow ss \bigcup \{C_{ss} \mid \texttt{get-classslot}(id(\tilde{a_{ss}}), C_{ss}) = \tilde{a_{ss}}\ \}$
    FinPour
FinPour
$sp \longleftarrow sp \cup$ `search-class-without-slot`$(\tilde{a}, PV)$
$\longrightarrow \langle sp; ss\rangle$
Fin

Où :

- `search-type-position`(t) est une fonction de MÉTÉO qui a pour résultat un couple formé de l'ensemble des sur-types et de l'ensemble des sous-types du type t dans le treillis de δ-types correspondant,
- `get-classslot`(i, C) est une fonction de TROPES qui a pour résultat la spécification de l'attribut d'identificateur i dans la classe C,
- `search-class-without-slot`$(\tilde{a}, PV)$ est une fonction de TROPES dont le résultat est l'ensemble des classes du point de vue PV qui ne définissent ni n'héritent l'attribut $\tilde{a}$.

Le résultat obtenu à l'issue de cette procédure est la position de la classe C qui décrit a dans la hiérarchie de spécialisation, *relativement à l'attribut* a. Cette procédure fait partie de l'interface entre TROPES et MÉTÉO.

À l'issue du traitement principal, le résultat final est un couple formé de l'ensemble des super-classes possibles et de l'ensemble des sous-classes possibles, *indépendamment de tout autre attribut non décrit par* C. Ces deux ensembles peuvent avoir un élément en commun, quand une classe existante a la même description que C lorsque projetée sur $A(C)$. Ce résultat doit alors être affiné par trois processus successifs, constituant un *post-traitement*, afin d'obtenir une solution correcte et optimale[11] :

1. élimination, parmi l'ensemble des super-classes, de celles qui possèdent un attribut que ne possède pas C (cette phase est nécessaire car le processus de classification tel qu'il a été conçu opère en réalité sur la projection des classes de la base de connaissances selon les attributs

[11] L'ordre d'application de ces deux processus n'aaucune importance en ce qui concerne la correction du résultat ; seules des considérations liées à la complexité peuvent préférer un sens d'application. Cependant, et pour les besoins des preuves à venir, nous supposerons que ces trois phases sont réalisées dans l'ordre.

décrits par C : il s'agit alors de revenir à la desription complète des classes afin de garantir que la solution respecte la propriété d'héritage complet),

2. réduction transitive de la spécialisation sur les deux ensembles du couple (on ne garde que les plus petites super-classes et les plus grandes sous-classes, en accord avec la spécialisation intensionnelle),

3. élimination, parmi toutes les super-classes proposées, de celles dont les filles directes n'appartenant pas à ssc_C ne sont pas disjointes avec C : cette phase est nécessaire pour le maintien de la propriété d'exclusivité, elle se fonde sur le test d'exclusivité intensionnelle, donc il s'agit de procéder à l'intersection du type de C avec celui de chacune des filles des super-classes possibles. Lorsque que l'intersection de $t(C)$ avec $t(C_f)$ (où C_f est une classe fille d'une super-classe proposée C_s) ne mène pas à $\perp$, alors C_s est enlevée de spc_C. Notons que pour l'élimination de certains calculs redondants, il est préférable d'effectuer cette phase après la réduction transitive.

Cette dernière phase du post-traitement n'est justifiée que lorsque l'hypothèse d'exclusivité intensionnelle est faite par le modèle. Les deux premières phases, quant à elles, sont liées au respect de la spécialisation intensionnelle.

Correction et complétude

Nous pouvons d'ores et déjà mettre en évidence deux propriétés de la solution proposée, qui traduisent qu'outre le fait que le δ-sous-typage sur les attributs est vérifié, la propriété d'héritage complet est respectée, ce qui montre la correction des liens de spécialisation intensionnelle inférés. Soit $P_C = \langle spc_C; ssc_C \rangle$ la position intensionnellement possible de C dans le point de vue PV. Nous supposons que la fonction `Search-type-position(`t`)` qui recherche la position d'un δ-type t dans son treillis est correcte vis-à-vis du δ-sous-typage (la preuve est en annexe B).

Propriété 9 *Tout attribut a de toute super-classe proposée pour C dans spc_C existe aussi dans C, et le type de a dans C est un sous-type du type de a dans la super-classe :*

$$\forall C' \in spc_C, \forall a \in A(C'), \exists a \in A(C) \text{ et } t(a_C) \leq_\tau t(a_{C'})$$

Propriété 10 *Tout attribut a de C existe aussi dans les sous-classes proposées pour C par ssc_C, et le type de a dans C est un sur-type du type de a dans les sous-classes proposées :*

$$\forall a \in A(C), \forall C' \in ssc_C, a \in A(C') \text{ et } t(a_{C'}) \leq_\tau t(a_C)$$

À ce stade, nous pouvons affirmer que les liens de spécialisation inférés sont corrects vis-à-vis de la définition en intension de la spécialisation.

Lemme 1 (Correction de la spécialisation)

$$\forall C' \in spc_C, C \leq_{\sigma,I} C' \text{ et } \forall C'' \in ssc_C, C'' \leq_{\sigma,I} C$$

Par ailleurs, nous pouvons montrer que tous les liens de spécialisation intensionnelle sont découverts par le processus de classification par types d'attributs, et ce à l'issue du traitement principal et de la phase 1 du post-traitement. Pour cela, nous supposons que la fonction `Search-type-position(`t`)` qui recherche la position d'un δ-type t dans son treillis est complète, c'est-à-dire que tous les liens de δ-sous-typage sont trouvés.

Lemme 2 (Complétude de la spécialisation) *Le processus de classification d'une classe C par les types de ses attributs identifie toutes les classes et sous-classes possibles pour C, modulo la réduction transitive de la spécialisation, et ce avant que l'élimination de super-classes potentielles menant à la violation de l'exclusivité soit effectuée :*

$$\forall C', C' \leq_{\sigma,I} C \Longrightarrow C' \in ssc_C$$

$$\forall C'', C \leq_{\sigma,I} C'' \Longrightarrow C'' \in spc_C$$

La position d'une classe dans un arbre de spécialisation doit non seulement respecter l'interprétation intensionnelle de la spécialisation, mais doit aussi préserver la disjonction des classes sœurs (propriété d'exclusivité des arbres de spécialisation). Pour montrer ce dernier résultat, nous exhibons trois propriétés immédiates concernant les sous-ensembles de classes spc_C et ssc_C constituant la solution au positionnement de la classe C. Notons ici que ces propriétés ne sont vérifiées qu'à partir du moment où l'exclusivité intensionnelle est une hypothèse du modèle de connaissances.

Propriété 11 *Il existe au plus une classe intensionnellement candidate à être super-classe de C.*

Propriété 12 *Pour toute classe intensionnellement candidate à être sous-classe de C, sa super-classe avant insertion de C est celle proposée pour être super-classe de C.*

Une conséquence immédiate de cette propriété est que les sous-classes proposées pour C sont nécessairement disjointes. En outre, grâce à la phase 3 du post-traitement, la propriété suivante est vérifiée.

Propriété 13 *Toutes les sœurs proposées pour C sont disjointes entre elles et disjointes à C. Soit C' l'unique super-classe proposée pour C dans spc_C.*

$$\forall C'' \leq_\sigma C' \text{ et } C'' \notin ssc_C, \|C''\|^I \cap \|C\|^I = \emptyset$$

Les trois propriétés précédentes nous amènent à considérer le lemme 3 :

Lemme 3 (Correction et cohérence de l'exclusivité intensionnelle) *Le processus de classification d'une classe C par les types de ses attributs garantit le respect des conditions d'exclusion intensionnelle portées sur la hiérarchie de spécialisation, à partir du moment où ces conditions étaient respectées avant la classification de C :*

$$\forall C_1, C_2 \in spc_C, \|C_1\|^I \cap \|C_2\|^I = \emptyset$$

$$\forall C'_1, C'_2 \in ssc_C, \|C'_1\|^I \cap \|C'_2\|^I = \emptyset$$

$$\forall C'' \leq_\sigma C' \text{ et } C'' \notin ssc_C, \|C''\|^I \cap \|C\|^I = \emptyset$$

Les lemmes 1, 2 et 3 assurent donc que la spécialisation intensionnelle est respectée (elle n'est pas remise en cause lors de la phase 3 du post-traitement), de même que les contraintes intensionnelles sur l'exclusivité, à partir du moment où elles le sont avant la classification de la classe ; le processus de classification des classes par les types de ses attributs est donc correct et complet vis-à-vis des interprétations intensionnelles. En conclusion, le processus de classification d'une classe par les types de ses attributs est complet et cohérent du point de vue intensionnel :

Théorème 4 *La classification d'une classe dans un point de vue, réalisée par la classification de types de ses attributs, est complète et cohérente au regard de l'interprétation intensionnelle.*

C'est alors le concepteur qui finalement accepte la position proposée par le raisonnement intensionnel, ce qui constitue la validation extensionnelle du processus. Outre les modifications des schémas des sous-classes de C et l'éventuel repositionnement des instances de la super-classe de C (qui sont des modifications effectuées directement dans la base de connaissances, cette validation entraîne la validation du type de la classe, donc entre autres son insertion dans le treillis des δ-types *records* (section 4.4.7).

En absence de l'hypothèse d'exclusivité intensionnelle entre deux classes sœurs de la hiérarchie d'un point de vue, le processus de classification par les types d'une classe peut mener à plusieurs solutions proposées (la propriété 11 n'est plus vérifiée), parmi lesquelles l'utilisateur doit faire son choix. Cette nécessaire intervention de l'utilisateur illustre l'importance de l'interprétation extensionnelle face à un mécanisme intensionnel souffrant d'une trop grande largesse des descriptions d'entités par rapport à ce qu'elles cherchent à dénoter dans le monde modélisé (cette largesse expliquant, et nous tournons en rond, l'impossibilité de l'hypothèse d'exclusion intensionnelle).

Exemple

L'exemple ci-dessous illustre la classification d'une classe dans un point de vue par les types des attributs de la classe, à partir de la donnée de la description complète de la classe. Nous supposons que les concepts ne possèdent qu'un seul point de vue. Nous avons délibérément choisi de ne pas considérer la possibilité de révision interactive de la description dans le but de simplifier l'exemple.

La base de connaissances considérée contient deux concepts K_1 et K_2, dont les définitions n'introduisent pas de cycles. L'objectif est de classer une classe C dans l'arbre de spécialisation de l'unique point de vue de K_1, à partir de la description de C.

La hiérarchie de spécialisation des points de vue uniques de K_1 et K_2 est donnée sur la figure 5.4. Nous n'y représentons que les liens de spécialisation établis, les descriptions des classes sont données par la suite. Le point de vue K_1 définit pour ses instances l'ensemble des attributs de la table 5.1, qui sont redéfinis par certaines des classes de K_1 comme indiqué dans la table 5.2.

a_1	\$un `Entier`	a_6	\$un K_2
a_2	\$un `Entier`	a_7	\$ens-de `Caractère`
a_3	\$un `Booléen`	a_8	\$un `Entier`
a_4	\$liste-de `Entier`	a_9	\$liste-de `Chaîne`
a_5	\$un `Chaîne`	a_{10}	\$un K_2

TAB. 5.1 - : Descriptions minimales des attributs du concept K_1.

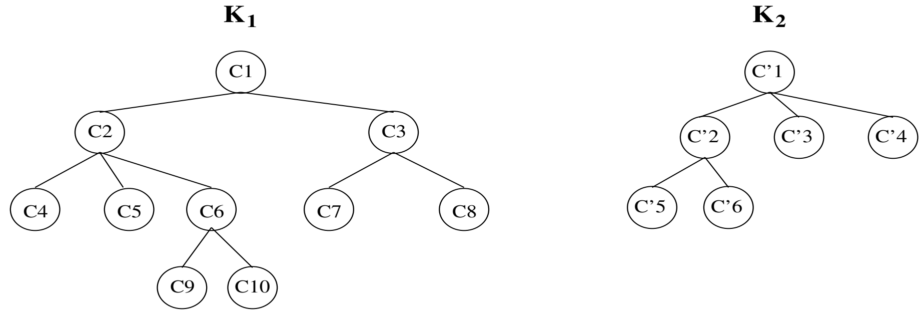


FIG. 5.4 - : Arbres de spécialisation des concepts K_1 et K_2.

Classe C_1 (racine) a_1 a_2	**Classe** C_2 a_1 $intervalle $[0;100]$ a_3 a_4 $card $[0;6]$ a_6
Classe C_3 a_1 $intervalle $[120;160]$ a_2 $intervalle $[0;100]$	**Classe** C_4 a_1 $intervalle $[40;80]$ a_3 $valeur `true` a_4 $card $[2;3]$ a_6 $parmi C'_6 a_8 $intervalle $[9;20]$
Classe C_5 a_1 $intervalle $[25;61]$ a_3 $valeur `true` a_4 $interdit $[18;20]$ $card $[4;6]$ a_6 $parmi C'_2 a_7	**Classe** C_6 a_1 $intervalle $[0;12]$ a_2 $intervalle $[-20;20]$ a_4 $card $[2;3]$ $parmi $[20;23]$ a_5
Classe C_7 a_2 $intervalle $[41;62]$ a_9 $card $[2;8]$	**Classe** C_8 a_2 $intervalle $[0;20]$ a_{10}
Classe C_9 a_3 $valeur `true`	**Classe** C_{10} a_3 $valeur `false`

TAB. 5.2 - : Descriptions des classes de K_1 (ne sont indiqués, dans une description de classe, que les attributs qu'elle affine ou introduit ; les attributs hérités ne sont pas précisés).

On cherche alors à déterminer la position d'une nouvelle classe C dans l'arbre de spécialisation de l'unique point de vue de K_1. Avant l'activation du processus de classification de classe, on vérifie que la spécialisation intensionnelle est cohérente, et que la propriété d'exclusivité est intensionnellement vérifiée. C possède la description complète donnée dans la table 5.3.

La classe C ne définit que les attributs a_1, a_2, a_3, a_4 et a_6, dont les treillis correspondant à leurs types sont donnés dans les figures 5.5 et 5.6[12]. Elles montrent les formes normalisées des domaines de valeurs de chaque attribut, exprimées en termes de EOLE[13]. Elles illustrent de plus

[12] Dans ces deux figures, par soucis de clarté, tous les liens de dépendances ne sont pas précisés, ni la description syntaxique des types *records*.

[13] Dans la figure 5.6, cependant, le δ-type correspondant à l'attribut a_4 dans C_6 (noté `(a4,C6)` sur la figure) n'est

Classe C		
	a_1	\$intervalle $[22;80]$
	a_2	
	a_3	\$sauf `false`
	a_4	\$card $[2;6]$
	a_6	\$parmi C'_2

TAB. 5.3 - : Description complète de C, la classe à positionner.

la position de chaque type d'attribut de C dans son treillis, après l'activation de la procédure `Search-type-position(`t`)`.

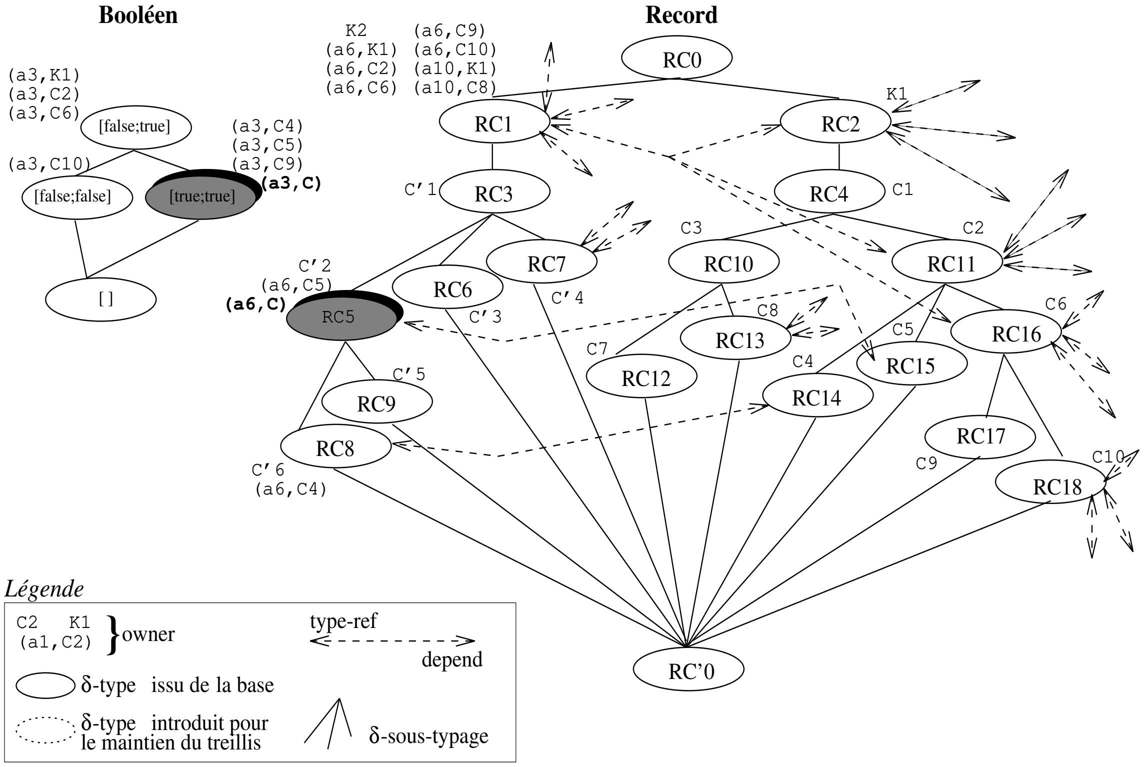


FIG. 5.5 - : Treillis des δ-types `Booléen` et `Record` (les types issus des attributs de K_2 ne sont pas indiqués). Les types grisés sont ceux qui correspondent aux attributs de C, pour lesquels on indique la position.

La position du type de chaque attribut de C dans son treillis, obtenue grâce à la fonction `Search-type-position(`t`)`, permet d'élaborer la position de la classe C relativement à chacun, par activation de la fonction `Search-slottype-position(`a, PV`)` dont le résultat dans notre exemple est donnée par la table 5.4.

L'intersection des positions relatives nous fournit la première version de la position de C dans l'arbre de spécialisation de K_1, la seconde version étant obtenue par réduction transitive selon la relation de spécialisation, appliquée à la première solution. La solution intermédiaire et la solution finale sont données dans la figure 5.7, qui illustre de plus la position finale de C dans l'arbre de

pas sous sa forme normalisée, pour une question de clarté dans la figure. Sa véritable forme normalisée est la suivante : $[\langle Entier;[20;23]\rangle$; (20 20) (20 21) (20 22) (20 23) (21 20) (21 21) (21 22) (21 23) (22 20) (22 21) (22 22) (22 23) (23 20) (23 21) (23 22) (23 23) ; `nothing` ; $[2;2]]$.

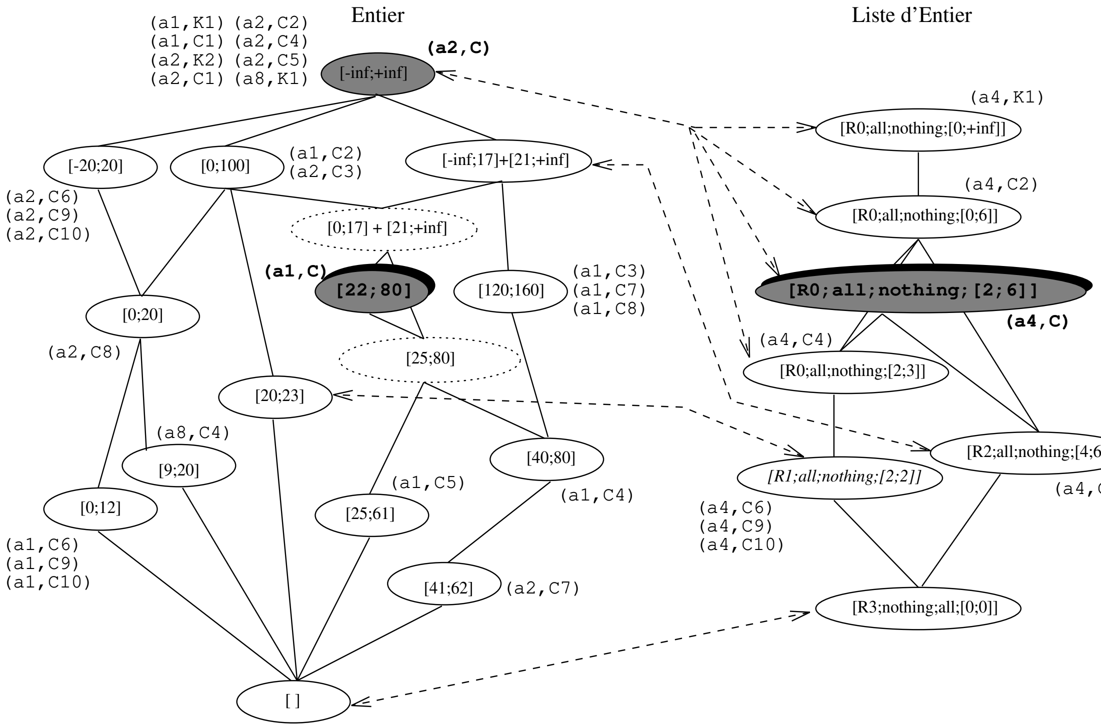


FIG. 5.6 - : Treillis des δ-types `Entier` et `Liste d'Entiers` (les types issus des attributs de K_2 ne sont pas indiqués). Les types grisés sont ceux qui correspondent aux attributs de C, pour lesquels on indique la position.

Attribut	**Super-classes**	**Sous-classes**
a_1	C_1, C_2	C_4, C_5
a_2	C_1, C_2, C_4, C_5	C_1, C_2, C_3, C_4, C_5 C_6, C_7, C_8, C_9, C_{10}
a_3	C_1, C_2, C_3, C_4, C_5 C_6, C_7, C_8, C_9	C_4, C_5, C_9
a_4	C_1, C_2, C_3, C_7, C_8	C_4, C_5, C_6, C_9, C_{10}
a_6	C_1, C_2, C_3, C_5, C_6 C_7, C_8, C_9, C_{10}	C_4, C_5

TAB. 5.4 - : Positions de C relatives à chacun de ses attributs.

spécialisation. On vérifie par ailleurs que C et C_6 sont intensionnellement disjointes, du fait des types disjoints de l'attribut a_1 dans ces deux classes.

La solution ainsi proposée par le système de types est une solution intensionnelle, qui doit être validée extensionnellement afin d'être définitivement acceptée dans le modèle.

Conclusion

La classification d'une classe C dans un point de vue est réalisée par la classification du type de chaque attribut de C dans son treillis, puis par intersection des positions relatives à chacun de ces attributs. Cette méthode, qui a été étudiée pour permettre l'élaboration de résultats partiels

Position intermédiaire	
Super-classes	**Sous-classes**
C_1, C_2	C_4, C_5

Position finale	
Super-classes	**Sous-classes**
C_2	C_4, C_5

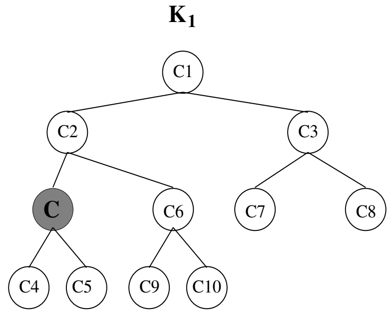


FIG. 5.7 - : Position de la classe C après classification. Le tableau de gauche illustre la position de la classe avant et après réduction transitive de la spécialisation. La figure de gauche donne la position finale et optimale de C.

est correcte et complète, et illustre le rôle prépondérant de l'interprétation intensionnelle en ce qui concerne l'inférence de liens de spécialisation.

Cet algorithme se rapproche conceptuellement de celui élaboré par Timothy Finin pour la classification de concepts dans une hiérarchie de subsomption, qu'il qualifie de *classification par profil d'attributs*, et qui consiste aussi à observer l'évolution du positionnement relativement à chaque attribut [Fin86].

Les parcours des treillis de δ-types pour l'insertion des types des attributs ne bénéficient pas de l'hypothèse d'exclusivité car elle est testée selon l'ensemble des attributs d'une classe, et n'a donc pas d'influence sur l'étude des types d'un seul attribut, indépendamment des autres. En ce sens, cette procédure de classification d'une classe C qui consiste à projeter la classification sur les attributs de C n'est pas la moins coûteuse : un parcours direct sur le treillis des types de classes est plus avantageux, certes, mais ne permet pas le calcul immédiat de résultats partiels, et rend de plus la correction interactive d'une description d'un attribut coûteuse dans la mesure où la classification doit reprendre au début.

5.3.3 Classification multi-points de vue

Nous présentons dans cette section un algorithme de classification de description catégorielle multi-points de vue. Il ne s'agit pas proprement dit de la classification d'une classe (au sens TROPES), mais de la classification d'une description résultant de la combinaison des descriptions d'une classe sous chaque point de vue, donc de la description complète d'un ensemble d'individus : *l'algorithme a pour objectif de classer tout un ensemble d'individus en projetant la description de cet ensemble sous chaque point de vue du concept.*

L'algorithme proposé relève du même principe que celui de la classification dans un seul point de vue, c'est-à-dire par projections successives selon les descriptions d'attributs. Le principe est alors d'élaborer, au fur et à mesure de la spécification de la description, un résultat partiel par point de vue. Les variations observées par rapport à l'algorithme mono-point de vue sont alors élémentaires, même si certaines considérations imposées par la raison d'être des points de vue doivent être faites.

- La récupération de l'information obtenue par classification d'un type d'attribut doit prendre en considération tous les points de vue du concept. Autrement dit, la procédure `search-slottype-position` est remplacée par une procédure semblable, dont le seul paramètre est l'attribut, et qui élabore un *ensemble de* couples de positions, qui, conjointement, représentent la position partielle de la description à classer, selon chaque point de vue. Cependant, les couples qui correspondent à la position de la description dans un point de vue particulier, relativement à un seul attribut non pertinent dans ce point de vue, ne sont pas considérés dans l'élaboration du résultat partiel de la position dans le point de vue correspondant[14]. La précision concernant la pertinence d'un attribut dans un point de vue est nécessaire à la complétude des liens de spécialisation intensionnelle inférés vis-à-vis de la spécialisation extensionnelle. En effet, lors de la classification d'une classe dans un point de vue, l'introduction d'un attribut non pertinent dans les critères intensionnels risque d'écarter des liens de spécialisation de la classe vers des sous-classes potentielles (ces dernières ne possédant pas l'attribut en question, puisqu'il n'est pas pertinent, ne seront donc pas retenues du fait de la définition en intension de la spécialisation qui exige qu'une sous-classe soit décrite par au moins tous les attributs de sa super-classe).

– Ensuite, tous les traitements effectués dans la classification mono-point de vue, qui opèrent sur la position finale de la description dans le but de l'affiner, sont itérés sur les positions dans chacun des points de vue (réduction transitive de la spécialisation, élimination des super-classes qui violent la propriété d'héritage complet et élimination des super-classes dont les sous-classes risquent de violer la propriété d'exclusivité).

À l'issue du traitement opéré sous chacun des points de vue, la position de la catégorie sous chaque point de vue respecte le théorème 4. Mais l'interprétation intensionnelle des passerelles ne peut être garantie par cet algorithme, qui n'exploite pas la coopération entre points de vue traduite par l'existence de passerelles. Plus formellement, cet algorithme ne garantit pas la propriété 14 que l'on est pourtant en droit d'attendre. En effet, pour que l'interprétation extensionnelle de la passerelle soit toujours vérifiée en présence des descriptions de classes composant C, il est nécessaire que l'interprétation intensionnelle le soit.

Propriété 14 *Soit la catégorie C classée sous n points de vue, projetée vers les classes $C_1, \cdots, C_n$. Soit $\langle sp_i ; ss_i \rangle$ la position de chaque C_i dans le point de vue PV_i. Nous savons que sous l'hypothèse d'exclusivité, sp_i est un singleton, on note $sp_i = \{C_{s,i}\}$. La définition en intension des passerelles impose que :*

$$Si\ \exists k < n\ t.q.\ \{C_{s,1}, \cdots, C_{s,k}\} \gg C_d,\ où\ C_d \in PV_{k+1}\ alors\ C_d \geq_{\sigma,I} C_{s,k+1}$$

Le non-respect de cette propriété par l'algorithme de classification multi points de vue s'explique par le fait que C peut définir un attribut a, uniquement présent dans PV_{k+1}, en lui donnant un type qui ne soit pas sous-type de celui donné par C_d, sans que cela ne soit détectable dans les autres points de vue, puisque cet attribut n'apparaît que dans PV_k. S'assurer, à l'issue de la classification de C dans tous les points de vue, que cette propriété est respectée, peut se faire très simplement à partir des types de $C_{s,k}$ et de C_d : si $t(C_{s,k}) \not\leq_\tau t(C_d)$, alors la description de C est incohérente avec les descriptions présentes dans la base. À ce stade, seul l'utilisateur, éventuellement assisté par le système, peut déterminer où se trouve cette incohérence parmi les descriptions des classes.

Cette condition intensionnelle sur les passerelles nous amène toutefois à proposer, au même titre que lors de la classification d'instances multi-points de vue, une inférence sur la description de C : si C a été classée sous la source d'une passerelle dans plusieurs points de vue, alors la

[14] L'information relative à la pertinence est fournie par l'utilisateur ou inférée plus ou moins arbitrairement par le système d'après la présence ou non de cet attribut dans d'autres classes du point de vue. Un tel critère est très subjectif, il n'est pas recevable dans la mesure où la base de connaissances est justement en cours de construction. Le système peut toutefois déclarer systématiquement pertinent un attribut dans un point de vue à partir du moment où cet attribut fait partie de la description d'au moins une classe de ce point de vue.

description de C dans le point de vue destination peut être inférée : en reprenant les notations de la propriété précédente, il s'agit de l'intersection entre les intensions des C_i sources et de l'intension de la classe destination C_d. Autrement dit, nous utilisons la propriété 14 à des fins d'inférence. Mais la description ainsi obtenue pour $C_{s,k+1}$ n'est qu'une maximisation de sa description possible, car la description de $C_{s,k+1}$ peut encore être affinée par rapport à celle de C_d par affinement des attributs spécifiques au point de vue PV_{k+1} présents dans C_d[15]. Ce recours aux passerelles signifie l'intégration, durant le processus de classification, de déclarations extensionnelles de la base, puisque les passerelles ont une signification qui implique avant tout les ensembles des individus dénotés, quels que soient leurs descriptions, et donc l'extension des classes en jeu.

En l'absence de l'hypothèse d'exclusivité, il peut y avoir, dans chacun des points de vue, plusieurs solutions possibles, et la seule inférence qui peut être faite en se basant sur les descriptions, en présence de passerelles, est l'élimination d'un ensemble de super-classes possibles (une par point de vue), lorsque ces super-classes sont sources d'une passerelle menant vers une destination qui n'est pas une super-classe possible pour la catégorie à classer vue sous PV_{k+1}. On élimine bien ici la solution conjointe de cet ensemble de super-classes sources, et non pas chaque super-classe possible individuellement : une classe participant à une telle source de passerelle pourra toujours être associée à une autre super-classe d'un autre point de vue qui n'appartient pas à cette même source.

Par ailleurs, il est à noter que si la catégorie C est classée sous n points de vue, donnant lieu à la création de n classes $C_1, \cdots, C_n$, alors le modèle peut inférer $\frac{n*(n-1)}{2}$ passerelles bidirectionnelles (une entre chaque paire de classes $(C_i, C_j), i \neq j$). Ce sont pourtant les seules inférences de passerelles qu'il soit possible d'effectuer, et encore, ici, ce n'est pas une inférence dûe à la description de la catégorie ni à la projection de cette description selon les différents points de vue, mais il s'agit bien d'une inférence dûe au statut extensionnel de cette catégorie : si sa description est décomposée sous chacun des points de vue, son extension ne l'est pas.

Cet algorithme de classification d'une description catégorielle n'est pourtant valide que dans la mesure où le concepteur connaît la description complète, dans tous les points de vue, modulo les inférences pouvant être faites sur la description de la catégorie sous un point de vue qui admet une destination de passerelle, inférences pouvant alors se soustraire aux renseignements fournis par l'utilisateur. On se rapproche ici de la classification d'instances multi points de vue réalisé par Olga Mariño. L'algorithme de classification de classes, contrairement à l'algorithme de classification d'instances, participe à la phase de construction de points de vue, et devrait donc intégrer la possibilité de gérer des descriptions incohérentes, avec comme option la création de nouveaux points de vue.

Les propriétés issues de la composition des passerelles et de la spécialisation sont nombreuses, et se décomposent elles aussi sous les deux interprétations intension/extension. Nous n'avons pas formalisé ces propriétés dans le chapitre 2, et c'est pour cette raison que nous n'abordons, dans cette section, que les aspects intuitifs liés à la classification d'une description catégorielle simultanément dans plusieurs points de vue. Une étude plus approfondie de ce mécanisme est en cours, mais dépasse le cadre de ce rapport.

[15] Une variante d'une telle inférence consiste à réaliser successivement – et non pas simultanément – dans chaque point de vue la classification par projection sur les attributs : à l'issue de chacune, le mécanisme active les inférences sur les passerelles afin d'initialiser la description dans le point de vue suivant.

5.3.4 Conclusion

La classification d'une description catégorielle confère au raisonnement intensionnel une plus grande importance que la classification d'une description individuelle, car la première n'est ici pas considérée comme étant un processus relevant de l'identification, donc ne nécessite pas la coopération régulière des deux raisonnements (afin de bénéficier de toutes les inférences possibles), alors que la seconde l'est.

Lorsque l'on considère la classification de description catégorielle comme un processus d'identification intensionnelle d'un groupe d'individus, ce qui implique que la description de ce groupe n'est pas obligatoirement connue entièrement, l'algorithme idéal à mettre en œuvre se rapproche de celui établi pour la classification d'instances, la différence étant le critère de comparaison des descriptions (spécialisation intensionnelle, donc sous-typage, dans le premier cas, et dans le second cas, instanciation intensionnelle, donc appartenance d'une valeur à un type)

Jérôme Euzenat a généralisé cette notion de classification d'une description (individuelle ou catégorielle) par une paramétrisation du critère de classification. La formalisation qu'il a établie en ce sens fait abstraction de la syntaxe descriptive des entités classées, pour se concentrer sur la nature du processus de classification et sur sa sémantique [Euz94] [CEG95]. Ce travail illustre entre autres l'importance du raisonnement intensionnel dans la classification. L'intérêt majeur de la notion de *système classificatoire* telle qu'il l'a introduite est qu'elle constitue une base formelle pour la comparaison des multiples processus de classification et de catégorisation développés dans de nombreux systèmes, quelle que soit leur syntaxe. On retrouve dans ce travail la distinction extension/intension dont nous traitons ici et qui est inhérente aux objets de représentation, respectivement sous les appellations *interprétations réelle/abstraite*, plus éloquentes d'un point de vue sémantique. La théorie des systèmes classificatoires est, entre autres, un support adapté à la comparaison des modèles de connaissances à objets et des logiques de descriptions, qui sont tous fondés sur le raisonnement taxonomique[16], autrement dit sur la classification [Att91] [Mac91a] [HBFC+91].

Une comparaison des statuts de la classification dans TROPES d'une part, et dans les logiques de description d'autre part, est abordée dans [Duc95] qui, comme il fallait s'y attendre, est basée sur la différence de statuts des classes dans TROPES – qui sont des descriptions – et des concepts dans les logiques de description – qui sont des définitions. Roland Ducournau a d'ailleurs basé cette étude sur une extension de la théorie des systèmes classificatoires de Jérôme Euzenat.

Par ailleurs, TROPES met l'accent sur la distinction classe/instance, aux niveaux syntaxique et sémantique, ce qui explique en partie l'appellation “modèle *à objets*”. En conséquence, les processus de classification de classe et d'instance diffèrent par leurs objectifs et par le critère de comparaison intensionnel considéré. Les logiques de description, quant à elles, ne différencient ni sémantiquement, ni syntaxiquement, les concepts individuels et définis, lorsqu'ils sont exprimés dans la T-Box. La différence de statut y est traduite par l'existence, dans la A-Box (composante assertionnelle), de liens des identificateurs d'individuels vers des identificateurs de concepts définis. Une des conséquences de ce choix de représentation se situe au niveau de la mise en œuvre de la classification qui, mis à part quelques systèmes comme KRIS [BH91], correspond à la recherche de liens de subsomption intensionnelle. Cette recherche s'effectue directement sur les termes de la T-Box, dont les descriptions sont comparées relativement à la subsomption intensionnelle, quel que soit le statut extensionnel de ces descriptions. Donc, contrairement à la classification d'instances et de classes dans TROPES, qui exploitent à la fois les inférences extensionnelles et intensionnelles, les classifieurs définis dans les systèmes à logiques de description ne reflètent pas les différences de statuts dénotationnels des individus et des catégories, même si elles sont explicitées par la A-Box.

[16] Taxonomic reasoning

5.4 Gestion des filtres

Un filtre (section 2.3.2) est un schéma de classe virtuel, dont la description est un ensemble de critères restrictifs portant sur les domaines de valeurs des attributs de l'entité sur lequel il est posé. Un filtre peut être utilisé de deux façons différentes :

- en tant que *requête* portant sur un ensemble d'instances (l'extension de l'entité sur laquelle est posé le filtre),
- associé à un attribut a (complexe) pour lequel il possède alors deux statuts :
  - un statut descriptif : le filtre représente, de par sa description en termes d'attributs contraints, une réduction de l'extension de a, et doit en conséquence être considéré dans le type de a,
  - un statut inférentiel : le filtre permet d'inférer un ensemble de valeurs possibles pour a, par activation du mécanisme de filtrage similaire à la résolution d'une requête.

Nous proposons dans cette section, dans un premier temps, de représenter par un type l'intension induite par un filtre, et dans un second temps, d'étudier le rôle de ce type dans le mécanisme de filtrage.

5.4.1 Typage d'un filtre statique

Un filtre est posé sur une entité, appelée *base du filtre*, qui est obligatoirement une classe, un concept ou un ensemble de classes issues du même concept et appartenant à des points de vue différents. Un filtre est ainsi la spécification d'une restriction de l'extension de sa base, donnée par une restriction de l'intension. En ce sens, le type d'un filtre désigne la restriction de l'intension de sa base.

La description d'un filtre consiste à poser des contraintes sur les valeurs des attributs de sa base. Il peut s'agir de contraintes inter ou intra-attributs. Une contrainte impliquant plusieurs attributs n'est pas prise en compte pour l'élaboration statique du type du filtre[17]. Lorsqu'un filtre est associé à un attribut d'une classe C, il peut en outre impliquer dans ses critères les autres attributs de C (figure 2.13) ; ce cas n'est pas non plus pris en compte dans le typage d'un filtre.

```
Classe Test-vue Sorte-de Examen-ophtalmo
      (concept  BILAN-SANTÉ Point de vue EXAMEN-MEDICAL)
Mèdecin $un Ophtalmologue
Client $liste-de Personne
        $sib-filtre
        {P = Personne
          P.indice-vue < 1.0}
...
```

FIG. 5.8 - : Un examen ophtalmologique concerne toutes les personnes dont la vue n'est pas parfaite. Le filtre posé sur l'attribut `Client` permet de retrouver ces personnes. Il ne contient qu'une seule contrainte qui concerne un attribut de la base, indépendante des autres attributs de la base : il s'agit d'un filtre *statique*.

Lors du typage d'un filtre par MÉTÉO, seule la partie statique est prise en compte, c'est-à-dire l'ensemble des restrictions apportées localement et individuellement sur les attributs de la base de

[17] Elles seront ultérieurement considérées par le module de gestion de contraintes MICRO (section 5.6)

ce filtre 5.8. Ces contraintes sur les attributs de la base sont exprimées, dans le filtre, au moyen de descripteurs de restriction de domaines, avec la même syntaxe et la même sémantique que la description des classes. Ainsi, le typage d'un filtre revient, en partie, à typer la classe virtuelle dont la description est équivalente à celle de la description statique de ce filtre.

Soit $F = \{B; [a_1 : d_1, \cdots, a_n : d_n]\}$ l'expression de la partie statique d'un filtre, telle que $\forall i \in [1..n], a_i$ est un identificateur d'attribut qui est présent dans la description de la base B, et d_i est le domaine de valeur associé à a_i, décrit à l'aide des descripteurs de restriction de domaines applicables aux attributs. Le typage brut d'un filtre est réalisé par la procédure suivante :

Typage brut d'un filtre F

1 Initialisation de $t(F)$, le δ-type correspondant à F

1.1. Création d'un δ-type issu du C-type `Record`
$t(F) \longleftarrow \langle Record; [R; D_1; D_2] \rangle$
1.2. Initialisation des champs d'énumération
$D_1 \longleftarrow$ `all`
$D_2 \longleftarrow$ `nothing`
1.3. Initialisation du champ contenant le *record* des types d'attributs
$R_0 \longleftarrow \langle\langle\ \rangle\rangle$
$i \longleftarrow 1$
Tant que $i \leq n$ faire
$t \longleftarrow$ `Typing-b(`$a_i : d_i$`)`
$R_i \longleftarrow \langle\langle R_{i-1} \mid a_i : t \rangle\rangle$
$i \longleftarrow i + 1$
FinTantQue
$R \longleftarrow R_n$

2 Prise en compte de la base du filtre $t(F) \longleftarrow$ `get-type(`B`)` $\sqcap_{Record} \langle Record; [R; D_1; D_2] \rangle$

3 Normalisation de $t(F)$: $t(F) \longleftarrow NormalForm_{Record}(T(F))$

Le typage d'un filtre est activé par le mécanisme de résolution de requête ou lors du typage d'un attribut. Dans ce dernier cas, l'attribut en question est un attribut complexe a, son type est un δ-type $t(a)$ issu de `Record`, et la prise en compte du filtre dans l'intension de a s'effectue par conjonction du type du filtre $t(F)$ et du type de a : $t(a) \longleftarrow t(a) \sqcap_{Record} t(F)$. $t(F)$ n'est pas explicitement représenté dans MÉTÉO dans la mesure où il ne correspond pas à une entité de représentation, mais il représente la composante intensionnelle d'un mécanisme d'inférence, et sa représentation au niveau du système de types est un premier pas vers la traduction de toute la connaissance intensionnelle représentée au niveau de la description des attributs.

5.4.2 Mécanisme de filtrage et type d'un filtre

L'algorithme classique de filtrage consiste à récupérer toutes les instances, qui, dans un premier temps, sont extensionnellement connues pour appartenir à la base du filtre, et dans un second temps satisfont les critères énoncés dans la description de ce filtre. Au regard de l'interprétation intensionnelle, il s'agit de retrouver l'ensemble des valeurs qui appartiennent au type du filtre et qui correspondent à des instances de la base de connaissances, c'est-à-dire des valeurs *records* qui peuvent se voir associer indirectement une signification dans le monde modélisé.

Il existe alors deux procédés principaux à l'utilisation du type d'un filtre pour la récupération des instances qui le satisfont.

1. Le premier procédé consiste simplement à déléguer au système de types la satisfaction des critères du filtre par toutes les instances extensionnellement acceptables (donc préalablement sélectionnées selon des critères extensionnels).

2. Le second procédé est la mise en œuvre d'un processus de pré-filtrage, dont le but est de limiter le nombre de tests de satisfaction du filtre, lorsque le nombre d'instances extensionnellement candidates est élevé. Le principe de l'algorithme est alors de déterminer, parmi les sous-types de celui de la base du filtre, lesquels ne sont pas disjoints du type du filtre[18], et correspondent à des types d'instances. Les instances correspondant aux types trouvés sont alors récupérées (par le lien `owner`) et sont alors testées au regard de l'appartenance intensionnelle à la description du type, ce qui correspond aussi à une opération du système de types. Selon ce procédé, le mécanisme de filtrage est la succession d'une opération intensionnelle (récupération des types), d'une opération extensionnelle (sélection des instances correspondant aux valeurs de ces types), et d'une seconde opération intensionnelle (test d'appariement de chacune des instances à la description du filtre).

Le second procédé correspond majoritairement à une opération intensionnelle, donc globalement effectuée par METÉO, mais n'est recevable que dans la mesure où le rapport entre le nombre d'instances potentiellement candidates (celles de la base du filtre) et le nombre d'instances, est grand, c'est-à-dire lorsque le filtre correspond à un véritable affinement de l'extension de sa base.

5.5 Gestion intensionnelle de la dynamique des représentations

Nous appelons *dynamique des représentations* l'ensemble des processus qui autorisent et gèrent la modification d'une base de connaissances, quelle que soit la nature de cette modification. Il s'agit d'un domaine de recherche à part entière, notamment développé dans le cadre des systèmes de gestion de bases de données à objets[19], dont les deux thèmes principaux sont :

- l'élaboration d'une sémantique réaliste des modifications, en fonction de leur nature, qui garantisse la cohérence de la base d'objets,
- la gestion des répercussions d'une modification au niveau des structures des objets, de façon à ce que la cohérence de la base soit préservée, et de ce fait que la cohérence des relations soit maintenue.

5.5.1 Approches des systèmes de gestion de bases de données

La grande majorité des systèmes de gestion de bases de données, qui traitent de l'évolution de schémas, définissent des contraintes d'intégrité (appelées aussi invariants) dont le rôle principal est

[18] Quand le filtre est une requête, son type est calculé par la conjonction du type de sa base et du résultat du simple typage de sa description ; donc le type d'un filtre est nécessairement sous-type du type de la base.

[19] Communément désigné sous le nom *schema evolution* ou *schema integration*

de définir la cohérence des bases de données : le respect de ces contraintes garantit la préservation de la cohérence générale. Ces contraintes sont en général fixées par le système, quelles que soient les applications (Matisse [Mol93], NO^2 [SGD93]), excepté dans ConceptBase [JEG$^+$95] qui autorise et intègre la définition de contraintes fournies par l'utilisateur. Ces contraintes d'intégrité correspondent aussi bien à la sémantique des relations entre entités (héritage entre classes, entre attributs) qu'à la déclaration de la notion d'identité (persistence d'une instance).

Le respect des contraintes d'intégrité, lors d'une modification de la base de données, est assuré par un ensemble de règles élémentaires dont l'utilisation, la combinaison et l'ordonnancement sont préalablement définis pour chaque modification possible (ORION [BKKK87]). Ainsi, chaque modification se voit associer une sémantique fixe, c'est-à-dire un ensemble de règles de propagation dont l'application garantit le respect des invariants. Pourtant, certains systèmes proposent pour une même modification, plusieurs combinaisons possibles des règles (ConceptBase) mais cela reste limité notamment lorsqu'il existe plusieurs niveaux de profondeur dans une propagation.

À vrai dire, dans tous ces systèmes, le seul cas qui ne soit pas strictement contrôlé par les contraintes d'intégrité est la remise en cause de l'appartenance d'une instance à sa classe modifiée. Dans ce cas, deux grandes approches se distinguent. La première consiste à créer de nouvelles versions des entités modifiées (Encore [SZ89] [Zdo86], Avance [ABB$^+$83], Gemstone [PS87]), ce qui a pour effet de permettre, dans une certaine mesure, la réorganisation de la hiérarchie de classes et la migration d'instances, mais toujours selon des règles pré-établies. La seconde solution consiste essentiellement à convertir les instances de façon à ce qu'elles s'adaptent à la nouvelle définition de leur classe avant modification (ORION, IRIS [FBC$^+$87], O_2 [Bar91]). Ces deux solutions, même ponctuées par l'approche novatrice d'Erik Oldberg [Old94], ont, au regard de l'aspect conceptuel de la problématique de l'évolution de hiérarchies, une portée trop limitée. L'approche d'Oldberg consiste à considérer les classes non seulement comme des ensembles de propriétés mais aussi comme des collections d'instances, ce qui introduit une sémantique plus riche et donc moins contraignante que celles exclusivement fondées sur les propriétés des classes.

Une telle rigidité dans les modèles de connaissances n'est pas très adaptée, car leur leit-motiv est plutôt de permettre la révision des connaissances : la gestion de la migration d'instances n'est pas suffisante, la sémantique des modifications doit aussi prendre en considération la possibilité de remettre en cause, en tant que répercussions, la structure des classes et des hiérarchies, ce qui commence par l'acceptation de la violation provisoire des contraintes d'intégrité.

5.5.2 Modifications d'une base de connaissances

Dans le cadre de la représentation des connaissances par objets, le problème de la dynamique des représentations est techniquement le même que dans le domaine des bases de données : il s'agit pour le système, d'une part de garantir le respect de la cohérence structurelle des entités et des relations, et, d'autre part, de proposer la sémantique opérationnelle des modifications, au regard des répercussions qu'elles peuvent engendrer. L'objectif cependant est plus ambitieux que celui visé dans le cadre des bases de données car la dynamique des représentations est liée au processus d'acquisition des connaissances et ne doit pas, en conséquence, figer les interprétations des modifications. Autrement dit, les “contraintes d'intégrité” qui seront considérées sont celles guidées par l' interprétation intensionnelle des entités et relations, mais les intensions pourront être momentanément remises en cause, puis remaniées, afin de s'adapter à une modification extensionnellement validée.

Problématique

La construction d'une base de connaissances est un enchaînement de modifications de la base de connaissances. Les modèles à objets considèrent généralement que la construction d'une base est incrémentale, c'est-à-dire qu'elle ne s'effectue que par ajouts d'éléments de connaissances. Toutefois, nous entendons par *dynamique d'une base de connaissances* toute action d'ajout, de suppression ou de modification d'une entité, qui passe par l'ajout, la suppression ou la modification des descriptions.

Parmi ces modifications, nous pouvons citer, de façon non exhaustive, l'ajout ou la suppression d'une instance, l'ajout ou la suppression d'une classe, l'ajout, la suppression ou la modification d'un attribut de classe, l'ajout, la modification ou la suppression d'une valeur d'attribut dans une instance, etc.

Le traitement de ces modifications par le système de représentation engendre des répercussions qu'il s'agit alors de caractériser. Toute la problématique du processus de gestion de cette dynamique vient du fait qu'une modification ne peut pas être interprétée dans l'absolu dans la mesure où elle dépend du contexte qui la motive et de l'état de la base de connaissances. Si nous prenons l'exemple de la restriction du domaine d'un attribut de classe, engendre-t-elle éventuellement la restriction du domaine de cet attribut dans les sous-classes qui le redéfinissent, ou bien cette restriction est-elle refusée car elle viole la relation de spécialisation (figure 5.9) ? Ou encore, si certaines instances ne peuvent *plus* appartenir à cette classe à l'issue de la modification, s'agit-il de les re-classer, ou bien de les modifier pour conserver les liens d'instanciation ? Cette question peut être reformulée comme suit : s'agit-il d'une modification qui a pour but de modifier l'intension afin de la rapprocher de l'extension, ou bien d'une modification qui a pour but de modifier l'extension et qui, en conséquence, modifie les descriptions[20] ?

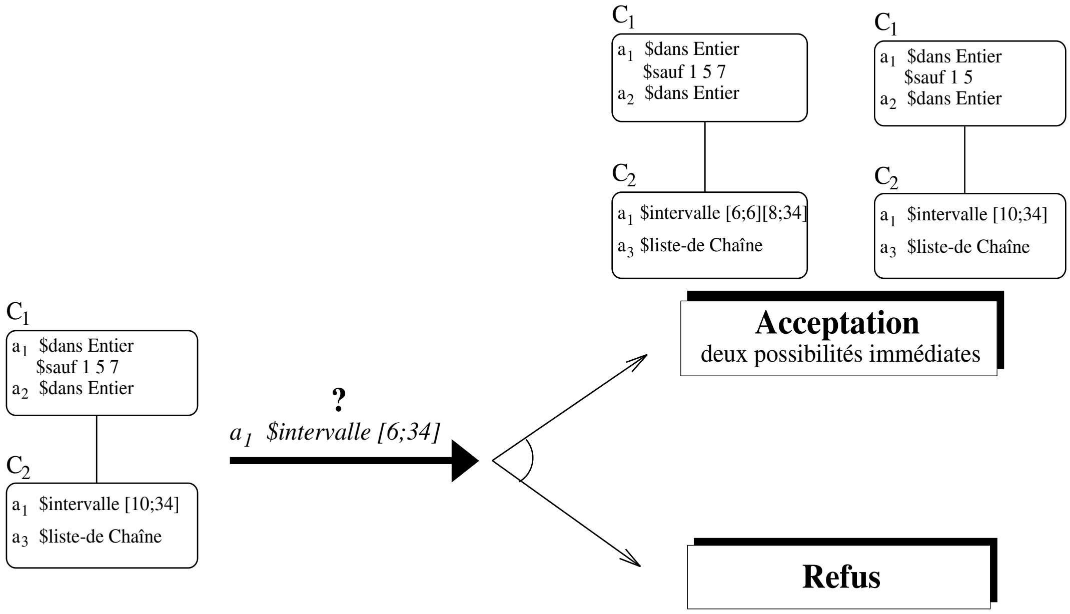


FIG. 5.9 - : La demande de modification de l'attribut a_1 peut être refusée car elle viole la spécialisation intensionnelle, ou peut être acceptée sous réserve de la modification d'au moins une description. Deux choix se présentent alors : la modification de C_1 ou de C_2 (extension du domaine de a_1 dans C_1 ou restriction du domaine de a_1 dans C_2).

Notre but n'est pas de dresser la liste de toutes les interprétations possibles des modifications

[20] Ce dernier cas relève du traitement de l'evolution du monde modélisé, ou tout au moins de l'évolution au niveau de la perception que l'utilisateur en a.

potentielles, mais d'étudier ici l'importance du rôle des interprétations intensionnelles lors de la phase d'acceptation et de correction d'une modification.

Architecture du gestionnaire de la dynamique

L'architecture du gestionnaire des modifications d'une base de connaissances est composé de quatre modules principaux :

- un module de gestion du dialogue (MGD) entre l'utilisateur et le système de représentation, dont le rôle est de gérer la communication entre l'utilisateur et les décisions du système, notamment en cas de modification problématique.
- Un module d'acceptation/réfutation de la modification, dont les critères correspondent aux contraintes intensionnelles portées sur la base de connaissances ; ce module ne doit pas se contenter d'une réponse "binaire", mais doit de plus fournir la raison précise d'un éventuel refus de sa part.
- Un module de gestion central des modifications et de leurs répercussions au sein de la base de connaissances (MGC), dont le rôle est d'activer, selon une sémantique opérationnelle définie par ce module même, des processus internes de réorganisation de la base (suppressions et créations d'entités, classification d'entités modifiées). Par exemple, lors de l'ajout d'une classe dans un arbre de spécialisation, c'est le MGC qui se charge de la modification effective des liens de spécialisation, et qui suggère la re-classification d'un certain nombre d'instances qui sont susceptibles de l'être. En cas de modification problématique, ce module est en outre chargé d'apporter des propositions réalistes de révision de la modification.
- Un module de gestion des versions de la base de connaissances. En effet, la gestion des modifications d'une base de connaissances Tropes consiste à ne pas remplacer entièrement une entité modifiée par sa nouvelle définition, mais à créer des *versions* de la base de connaissances, une modification engendrant la création d'une nouvelle version [Tay95]. Ces versions sont physiquement organisées en *couches* ordonnées chronologiquement, ce qui permet de ne pas recopier physiquement toute une base de connaissances pour une simple modification. Toutes les versions sont accessibles aux utilisateurs. Ce mode de gestion des modifications est intéressant, non seulement pour la gestion concourante de l'évolution d'une base de connaissances [LT94], mais aussi pour le caractère immédiat du retour en arrière à l'issue d'une modification, qui peut être motivé par le refus des conséquences de cette modification.

La figure 5.10 illustre l'interaction entre ces quatre modules. En particulier, le module de vérification intensionnelle des modifications est intégré à Metéo.

5.5.3 Analyse d'une modification

Le module de gestion central a pour rôle de "comprendre" la modification soumise par l'utilisateur, afin de l'interpréter correctement, notamment en ce qui concerne les répercussions que cette modification peut engendrer. Deux niveaux d'assistance caractérisent un tel module.

1. Il se contente d'accepter ou réfuter brutalement une modification, en fonction de la réponse du système de types, c'est-à-dire de la possibilité d'admissibilité intensionnelle. Dans ce cas, la coopération entre le système de types et le MGC est élémentaire : le MGC se contente de

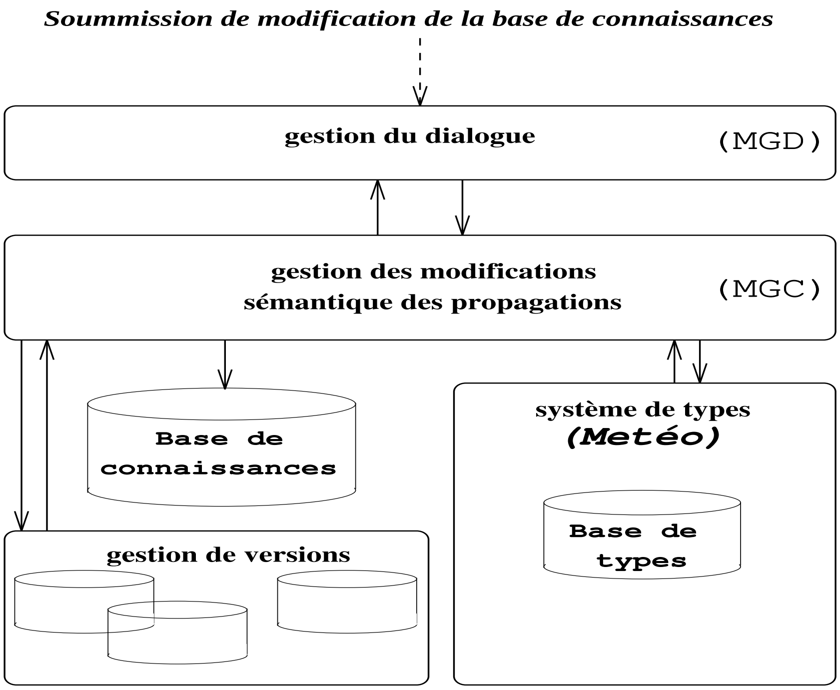


FIG. 5.10 - : Architecture logicielle du gestionnaire de la dynamique des représentations. Une modification de la base est soumise par un utilisateur, qui est transmise par le MGD au MGC. Ce dernier soumet au système de types la modification qui doit être vérifiée au regard de la cohérence des interprétations intensionnelles. La réponse du système de types est alors interprétée par le MGC dont le but, à ce stade de l'opération, est de proposer et réaliser récursivement les propagations de la modification de telle façon qu'elles satisfassent les attentes de l'utilisateur. Après validation ultime de la modification par l'utilisateur, la base de connaissances avant modification est sauvegardée par le module de gestion de versions qui met à jour les liens entre cette ancienne version et la nouvelle, issue de la modification.

déléguer au système de types un ensemble d'opérations de vérification élémentaires, comme des tests de spécialisation ou d'appartenance intensionnelles. Ces tests correspondent à la vérification de la validité de la modification *et* des propagations qu'elle engendre. La réponse fournie par le système de types est ici binaire.

- En cas de réponse positive, le MGC autorise la modification et la réalise explicitement, en propageant cette modification. Ceci provoque alors des modifications au niveau de la base de types qui doivent être amorcées par le MGC.
- En cas de réponse négative à l'acceptation intensionnelle, le MGC refuse la modification, sans appel.

2. Il tente d'analyser les violations intensionnelles en tentant de juger de leur gravité. Dans ce cas, la coopération entre le MGC et le système de types mène à une succession d'échanges dont l'objectif final est de proposer à l'utilisateur une variante de sa modification qui soit valide intensionnellement. Pour cela, les réponses fournies par le système de types doivent être plus élaborées car elles doivent renseigner le MGC du degré de réfutation d'une vérification ainsi que de sa cause exacte. Comme dans le cas précédent, à partir du moment où une modification est acceptée par le MGC (impliquant qu'elle l'est aussi par le système de types), qu'il s'agisse de la modification initiale ou d'une variante, elle est traitée par le MGC, de même le sont ses répercussions. Ce traitement engendre aussi des modifications dans la base de types.

Le traitement d'une modification de la base de connaissances tel que réalisé par le MGC conduit finalement à la création d'un arbre fini de modifications ou d'appel à des mécanismes d'inférence, dont le sommet est la modification initiale éventuellement remaniée. À chaque sommet d'un sous-arbre est associé, par ses sous-nœuds immédiats, les modifications qui sont issues de la propagation de la modification représentée par le sommet. Le MGC fournit en sus l'ordre de parcours de l'arbre des modifications, qui dépend des liens de dépendances entre les entités de représentation confrontés à la nature de chaque modification, dont l'objectif est de n'appliquer l'opération associée à un nœud qu'à partir du moment où cette dernière n'a aucune conséquence qui n'ait été déjà traitée. La figure 5.11 illustre un exemple d'arbre de modifications et son parcours, dans le cas d'une restriction apportée sur le domaine de valeurs d'un attribut.

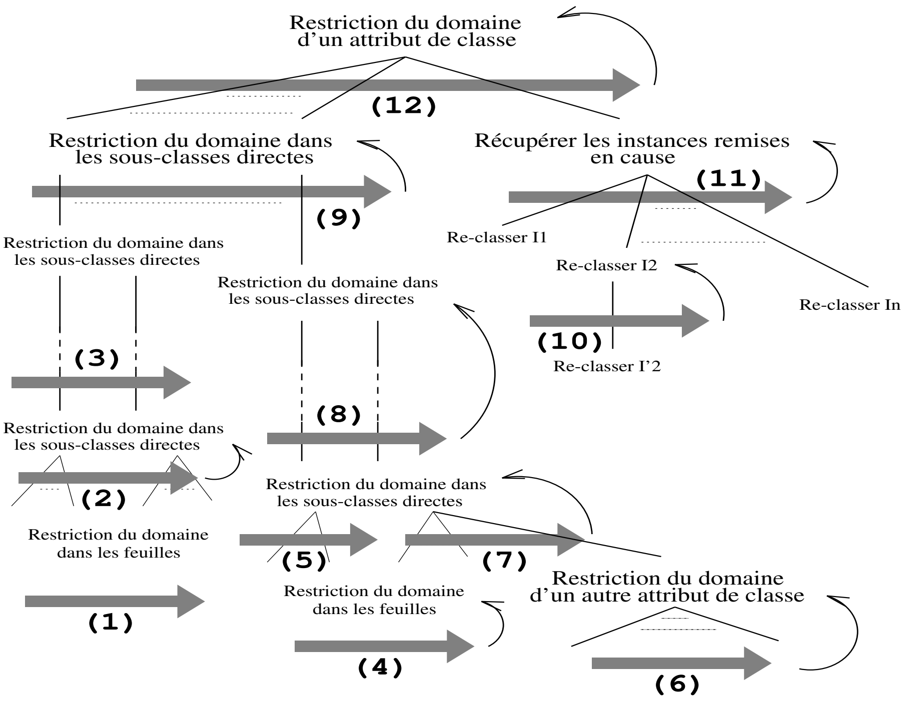


FIG. 5.11 - : Une restriction est portée sur le domaine d'un attribut d'une classe, il est décidé de propager cette restriction dans les sous-classes, et de reclasser les instances qui ne vérifient plus les conditions des classes modifiées. La modification d'un attribut dans une classe peut engendrer la modification d'un attribut référençant la classe modifiée dans une autre classe (éventuellement d'un autre concept). La re-classification des instances ne s'effectue qu'une fois que les descriptions catégorielles sont stables, selon un ordre établi préalablement mais qui peut être modifié au cours d'une de ces classifications (notamment lorsque la classification d'une de ces instances nécessite celle d'une autre prévue ultérieurement). Une fois accomplies séquentiellement toutes les modifications issues d'un sommet, celle concernant ce sommet peut l'être.

Le module de gestion centrale de TROPES est actuellement en cours d'élaboration : nous n'en avons ici que décrit le principe général [Cra95], qui correspond au second degré d'analyse présenté ci-dessus, afin de cerner le rôle que jouent les interprétations intensionnelles dans les processus de gestion de la dynamique des représentations. L'avantage du second degré d'analyse du MGC est qu'il offre une meilleure contribution à l'affinement de la structure d'une base de connaissances : il permet la remise en cause des définitions intensionnelles dans l'objectif de parfaire le rapport entre ce qui est décrit et ce que l'on veut décrire réellement.

En ce sens, le rôle des interprétations intensionnelles représentées dans METÉO diffère de celui joué lors des processus d'inférence de la base de connaissances. En effet, lors du traitement d'une modification de la description d'une entité, l'intension s'adapte à la décision du MGC qui est prise en accord avec l'utilisateur, c'est-à-dire avec celui qui définit explicitement l'extension des entités. Autrement dit, en cas de décision de la validation extensionnelle d'une modification, même si cette dernière ou ses conséquences, violent des interprétations intensionnelles, elle est acceptée et les descriptions sont remaniées en conséquence. Un tel comportement du système illustre la sémantique de la coopération analysée dans la section 5.1.1.

5.5.4 Validation d'une modification

À partir du moment où le MGC a décidé, avec l'accord de l'utilisateur, de la validité d'une modification et de ses conséquences, elles doivent être effectivement réalisées au niveau de la base, ce qui revient d'une part à modifier les termes correspondant aux entités de représentation, par le parcours de l'arbre d'interprétation de la modification, et d'autre part à modifier la base de types en conséquence.

Activations des modifications dans la base de types

À l'issue de chaque opération associée à un nœud de l'arbre des modifications, est activée une demande de modification élémentaire précise dans la base de types, qui peut aller du typage d'une nouvelle entité à la suppression du lien bidirectionnel entre une entité et son type, en passant par la modification d'un type d'entité existant. Plus précisément, à chaque modification d'une entité de représentation est associée une opération sur le type de cette entité.

La table 5.5 indique, pour les trois catégories de modifications élémentaires de la base de connaissances, les opérations associées dans la base de types et exécutées par METÉO. Ces opérations sur les types peuvent elles-mêmes être suivies d'une réorganisation de la base de types, comme nous le verrons ensuite.

Modification de la base de connaissances	**Modification de la base de types**
Modification d'une entité	Calcul du nouveau type de l'entité Insertion de ce nouveau type Mise à jour du lien `owner` de l'ancien type Mise à jour du lien `owner` du nouveau type Mise à jour du lien `type`
Suppression d'une entité	Mise à jour du lien `owner` du type de l'entité
Ajout d'une entité	Calcul du type de l'entité Classification de ce type Mise à jour du lien `owner` Mise à jour du lien `type`

TAB. 5.5 - : Opérations sur la base de types associées aux modifications de la base de connaissances.

En réalité, les opérations présentées dans la table 5.5 ont été préalablement effectuées virtuellement sur la base de types, lors du test de la validité intensionnelle d'une modification. Autrement dit, la modification a été simulée intensionnellement sur la base de types, en prenant en compte des éventuelles modifications de descriptions requises pour la validité extensionnelle de la modification et de ses propagations. Lors de cette phase de vérification, ici concrétisée par la validation décidée par le MGC, le processus est accompagné d'un gestionnaire d'erreurs élaboré dont l'objectif est de renseigner précisément le MGC de la nature exacte d'éventuelles erreurs intensionnelles. Cette phase de validation dans la base de types s'effectue donc assurément sans erreurs puisqu'elle a été préalablement testée intégralement. La validation au niveau de la base de types n'est en fait qu'une restitution de la simulation.

Propagations internes à la base de types

Toute modification sur un δ-type dans MÉTÉO peut engendrer d'autres remaniements de la base de types. Nous avons vu, lors de la validation de l'ajout d'un type, que la procédure d'insertion ne se contente pas de chercher la position du nouveau type, elle détecte en outre le cas où la structure de treillis n'est plus préservée, et le cas échéant, prend l'initiative de créer des types intermédiaires. Par ailleurs, la création d'un type pouvait engendrer la création d'autres types en chaîne, en particulier dans le cas des types construits.

Par la suite, nous considérons qu'une modification se définit intégralement par la séquence d'une suppression et d'un ajout. Même si une telle hypothèse ne rend pas optimaux les traitements, elle a l'avantage de clarifier l'exposé sans que le principe général du traitement n'en soit affecté.

Les traitements internes à la base de types ont déjà été étudiés dans le cas de créations de types et de liens de dépendances. La suppression (et a fortiori la modification) de tels liens ne conduit pas nécessairement à la suppression du type impliqué, dans la mesure où un type peut être partagé par plusieurs entités et autres types, ses *référents*, qu'il existe ou non des liens sémantiques entre ces référents. Ainsi, un type t doit être supprimé à partir du moment où il n'a plus de raison d'être, c'est-à-dire lorsque les trois conditions suivantes sont remplies :

1. `get-owner(`t`)` $= \emptyset$, c'est-à-dire lorsqu'aucune entité de représentation n'est typée par t,
2. `get-depend(`t`)` $= \emptyset$, c'est-à-dire lorsque t n'est pas issu de l'existence d'un δ-type construit,
3. `get-lattice(`t`)` $= \emptyset$, c'est-à-dire que t n'est pas utile pour la maintenance de la structure de treillis.

Ces trois conditions sont regroupées au sein de la procédure `Check-deletion(`t`)`. Si les trois conditions précédentes sont vérifiées (le type n'est plus "utile"), alors la suppression effective de ce type est activée. MÉTÉO réalise deux procédures différentes pour la suppression effective d'un δ-type. La première, élémentaire, est réservée à la suppression des δ-types issus de types simples. La seconde, récursive, est dédiée à la suppression de δ-types construits, qui nécessite la mise à jour du lien `depend` des types référés par le δ-type supprimé. Ces deux procédures sont intégrées à une procédure générale `Delete-type(`t`)` qui sélectionne la procédure de suppression correspondant à la nature du type t.

Procédure de déclenchement éventuel d'une suppression de type, `Check-deletion(`t`)`
Début
  Si (`get-owner(`t`)` $= \emptyset$) et (`get-depend(`t`)` $= \emptyset$) et (`get-lattice(`t`)` $= \emptyset$)
    Alors `Delete-type(`t`)`
Fin

Suppression du δ-type t d'une entité, `Delete-type(`t`)`
Soit $t = \langle T; E(T) \rangle$
Début
  Si $T \leq_C$ `Construits`
    Alors ($E(T) = [type\text{-}ref; D_1; D_2; \star]$)
      Pour tout $t_i \in type\text{-}ref$ faire
        `put-depend(`t_i`,get-depend(`t_i`)`$\backslash t$`)`
        `Check-deletion(`t_i`)`

```
        FinPour
    FinSi
    Remove-type-from-lattice(t)
Fin
```

(où le symbole $\star$ désigne une spécification indifférente). La procédure `Remove-type-from-lattice(t)` a pour effet de supprimer t du treillis des δ-types correspondant, par élimination des liens de δ-sous-typage, maintenus à l'intérieur même de la structure des δ-types, qui impliquent t, et par restructuration transitive du treillis. L'algorithme de cette procédure est donnée en annexe C.

L'arbre d'appel de ces deux procédures est fini, donc la terminaison assurée, car les δ-types construits infinis sont évités lors du typage de TROPES.

5.5.5 Exemple : suppression d'une classe

L'exemple suivant illustre quelques étapes de la demande de suppression de la classe C_2 du point de vue PV_1 du concept K_1, dans la base de connaissances de la figure 5.12. Dans la plupart des

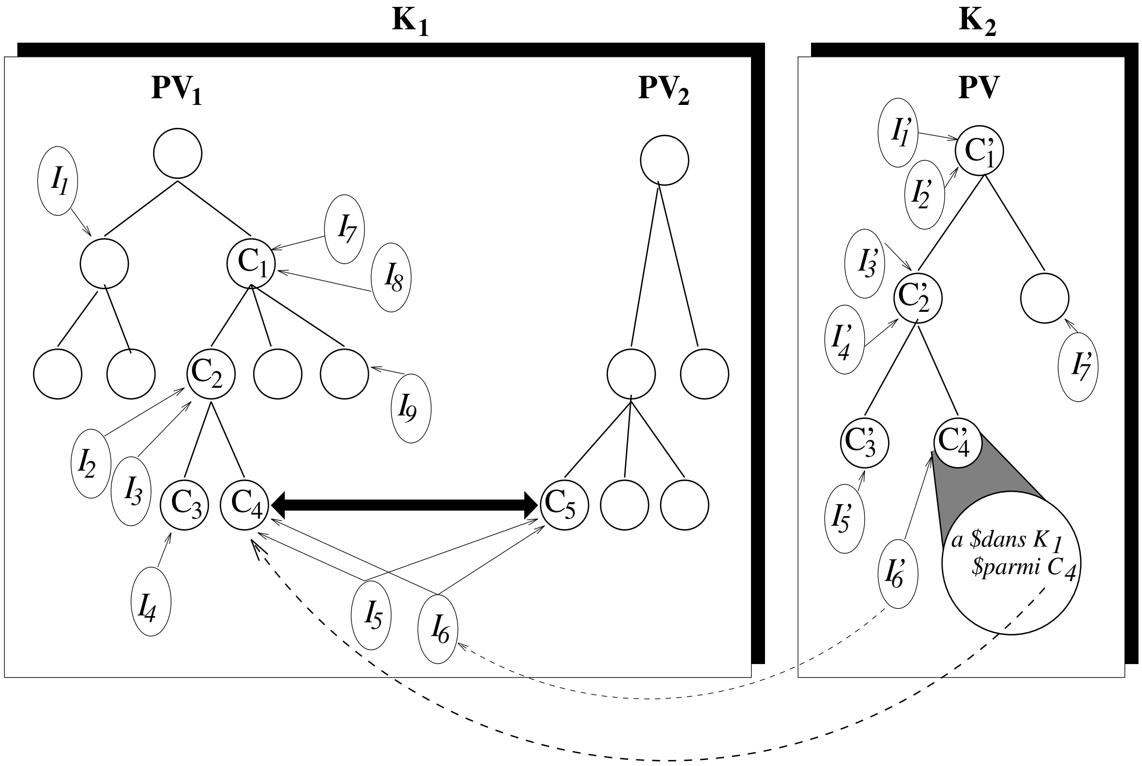


FIG. 5.12 - : La base de connaissances B considérée dans l'exemple est constituée de deux concepts, K_1 et K_2. La modification soumise par l'utilisateur est la suppression de la classe C_2.

systèmes à objets qui gèrent la dynamique des classes, cette demande aboutirait à la suppression de la sous-classe de C_2, de toutes les instances des classes supprimées, des attributs typés par C_2 ou sa sous-classe ; de façon générale, la sémantique communément appliquée dans une telle suppression est celle qui a pour effet de supprimer toutes les entités qui dépendent de celle supprimée. Même si des systèmes qui gèrent les versions permettent de ne pas perdre toutes les informations, ces systèmes ne proposent pas un accès simultané à ces différentes versions, c'est-à-dire ne proposent pas la fusion de plusieurs versions.

Dans Tropes, l'accent est mis sur la préservation de l'information pertinente. De ce fait, ce n'est pas parce qu'une catégorie est supprimée que tous les éléments de son extension doivent l'être[21], car ces éléments sont censés être la représentation d'une entité individuelle du monde modélisé. Ainsi, dans ce type de modification, Tropes tente de re-classer les instances, sauf si l'utilisateur demande explicitement la suppression d'une instance, signifiant ainsi au modèle que cet objet n'a pas (plus) de dénotation dans le monde modélisé. De la même façon, les attributs typés par une classe supprimée se verront plutôt typés autrement (par une sur-classe, par exemple) que supprimés. Toujours selon la même logique, le système tentera de garder les sous-classes d'une classe supprimée, sauf mention contraire de l'utilisateur.

Élaboration de l'arbre des modifications

La demande de suppression de la classe C_2 par l'utilisateur mène à une alternance d'interactions entre le MGC et l'utilisateur d'une part (dialogue pour déterminer les propagations de la modification), et entre de MGC et Metéo d'autre part (vérifications intensionnelles des propositions de modifications du MGC). À l'issue de ce traitement, les répercussions de la modification initiale sont établies, et représentées dans l'arbre des modifications de la figure 5.13. Le résultat de l'application des actions de cet arbre est visible sur la figure 5.14.

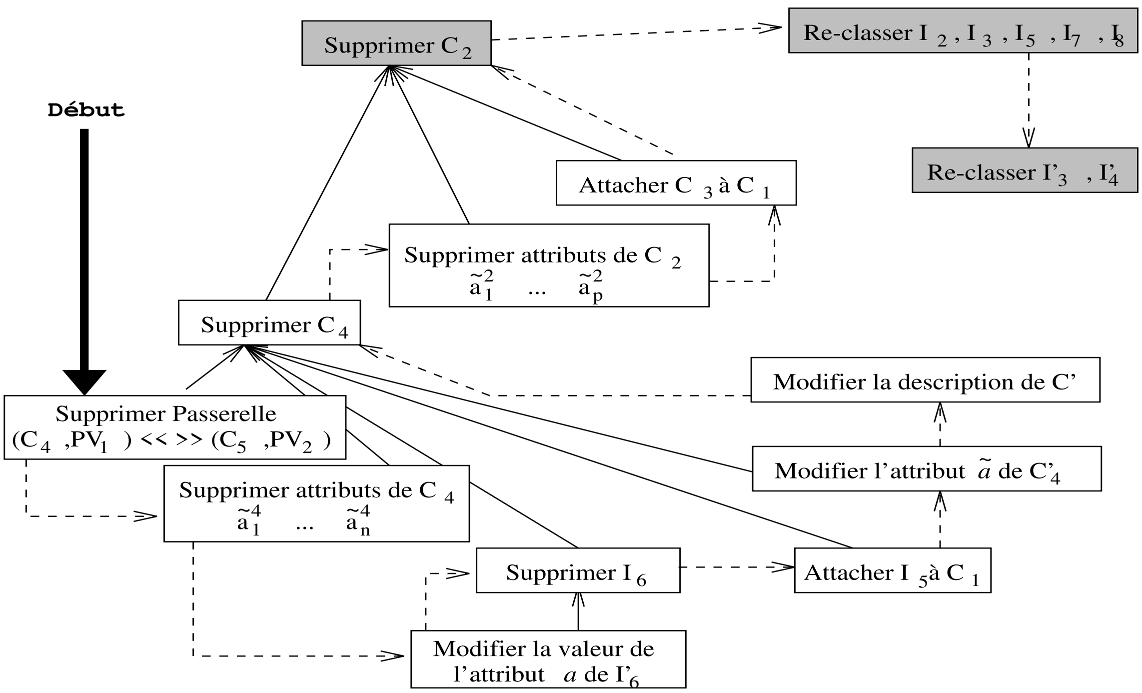


Fig. 5.13 - : Les modifications décidées conjointement entre le système de représentation et l'utilisateur sont représentées par un arbre et un sens de parcours de cet arbre. Suite à toute réorganisation structurelle de la base, certaines instances peuvent être re-classées, lorsque des contraintes sont relâchées.

Le rôle de Metéo dans l'élaboration de cet arbre des modifications est d'effectuer les vérifications nécessaires des propositions de propagations intermédiaires. Au delà de ces vérifications, Metéo fournit au MGC les erreurs intensionnelles qu'une modification engendre, ce dernier interprétant l'erreur et en déduisant d'éventuelles nouvelles modifications pour éviter cette erreur, ou décidant de la trop grande gravité de l'erreur et refusant la modification qui l'a engendrée,

[21] D'autant plus qu'une instance n'est pas créée à partir d'une classe, mais d'un concept ; de ce fait, son identité lui est attribuée par le concept.

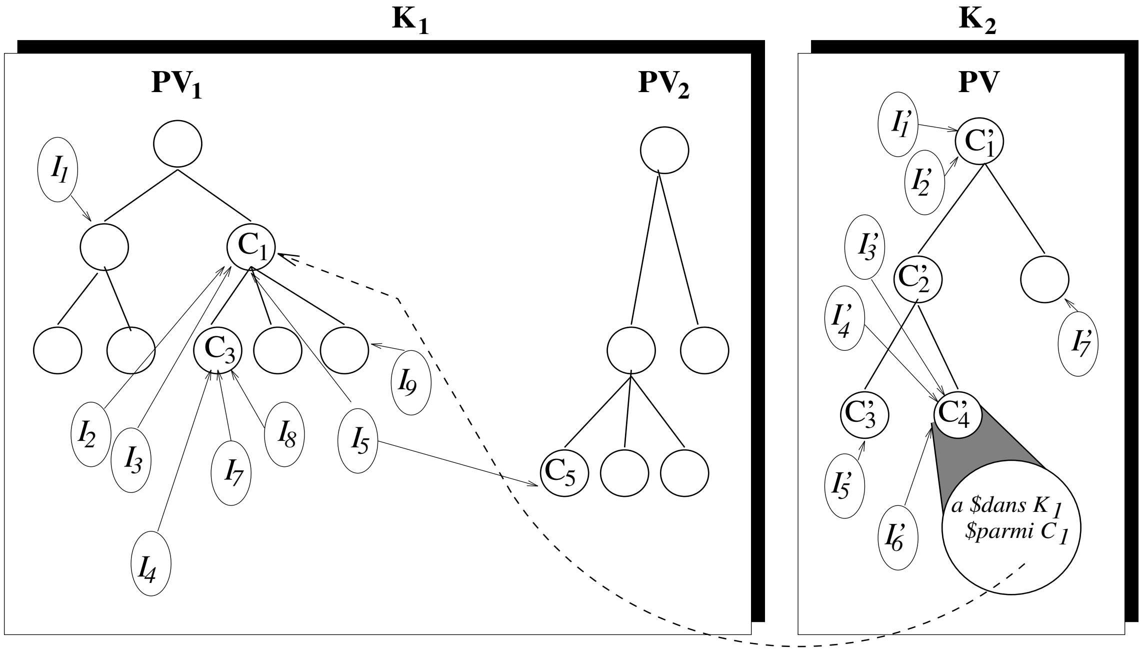


FIG. 5.14 - : État de B après la suppression de la classe C_2.

Par exemple, la modification de l'attribut dans C'_4 (extension du domaine de valeurs) pouvait engendrer plusieurs types d'erreurs que MÉTÉO recherche :

1. non-respect de l'exclusivité intensionnelle,
2. ou non-respect de la relation de spécialisation intensionnelle entre C'_4 et C'_2.

Dans ce cas, MÉTÉO en avertit le MGC qui peut ensuite décider, en consultant éventuellement l'utilisateur, de *refuser la modification* ou d'*adapter les descriptions mises en cause* comme suit :

1. propager l'extension du domaine de a au niveau de C'_2 (la spécialisation ne pouvant être remise en cause que du fait de la description de a dans C'_2 qui n'est plus une généralisation de a dans C'_4), afin de retrouver le lien de δ-sous-typage entre les types de C'_2 et de C'_4,
2. restreindre la description de C'_3 afin de retrouver la propriété d'exclusivité intensionnelle.

Un telle adaptation peut alors, bien entendu, mener à d'autres vérifications, jusqu'à stabilisation du processus de propagation par reconnaissance de la validité intensionnelle et extensionnelle de toutes les modifications engendrées.

Propagations dans Metéo

Dans la section précédente, la table 5.5 a montré que toute opération de modification effectuée sur la base de connaissances est effectuée une opération sur la base des δ-types qui se doit de toujours refléter les intensions des entités et l'interprétation intensionnelle des relations.

Ainsi, dans notre exemple, à l'arbre de modifications de la figure 5.13 complété de son ordre de parcours est associée la liste des opérations élémentaires correspondantes sur la base de types :

1. Pour tout attribut $\tilde{a}$ présent dans la description de C_4 Faire

put-owner($t(\tilde{a})$, get-owner($t(\tilde{a})$) $\backslash\{\tilde{a}\}$)

2. put-owner($t(I'_6)$,get-owner($t(I'_6)$)$\backslash\{I'_6\}$)
3. $t \longleftarrow$ get-type(I_5)
   $t' \longleftarrow$ Typing-b(I_5)
   Insert-type(t')
   put-owner(t,get-owner(t) $\backslash\{I_5\}$)
   put-owner(t',get-owner(t') $\cup\{I_5\}$)
   put-type(I_5,t')
4. Soit $\tilde{a}$ l'attribut à modifier dans C'_4
   $t \longleftarrow$ get-type($\tilde{a}$)
   $t' \longleftarrow$ Typing-b($\tilde{a}$)
   Insert-type(t')
   put-owner(t,get-owner(t) $\backslash\{\tilde{a}\}$)
   put-owner(t',get-owner(t') $\cup\{\tilde{a}\}$)
   put-type($\tilde{a}$,t')
5. Soit $\tilde{a}$ l'attribut modifié dans C'_4
   $t \longleftarrow$ get-type(C'_4)
   $t' \longleftarrow$ Typing-b(C'_4)
   Insert-type(t')
   put-owner(t,get-owner(t) $\backslash\{C'_4\}$)
   put-owner(t',get-owner(t') $\cup\{C'_4\}$)
   put-type(C'_4,t')
6. put-owner($t(C_4)$, get-owner($t(C_4)$) $\backslash\{C_4\}$)
7. Pour tout attribut $\tilde{a}$ présent dans la description de C_2 Faire
   put-owner($t(\tilde{a})$, get-owner($t(\tilde{a})$) $\backslash\{\tilde{a}\}$)
8. put-owner($t(C_2)$, get-owner($t(C_2)\backslash\{C_2\}$)
9. Pour toute instance I qui change de classe(s) d'attachement Faire
   $t' \longleftarrow$ Typing-b(I)
   Insert-type(t')
   put-owner(t,get-owner(t) $\backslash\{I\}$)
   put-owner(t',get-owner(t') $\cup\{I\}$)
   put-type(I,t')

Rappelons qu'à chaque modification d'un lien participant à la raison de l'existence d'un δ-type, le système de types MÉTÉO active la procédure `Check-deletion(t)` où t est le type modifié, afin éventuellement de supprimer effectivement t de la base de δ-types.

5.5.6 Conclusion

Le système de types MÉTÉO, en tant que module réalisant les inférences et vérifications intensionnelles, joue un rôle important dans le processus de gestion de la dynamique des représentations : non seulement il est utilisé pour procéder à la vérification de la correction intensionnelle des modifications soumises et transmises par le MGC (Module de Gestion Centrale), mais il fournit aussi au MGC la portée de chaque modification soumise au regard des intensions, ce qui permet à ce dernier de juger de la justesse d'une modification ou de la gravité d'une erreur intensionnelle.

L'étude développée dans ce chapitre illustre une fois de plus le rôle important que jouent les descriptions, autrement dit les intensions, lors des processus de gestion et construction d'une base de connaissances. Parallèlement, nous avons montré une fois encore que le rôle des validations et propositions extensionnelles est prépondérant dans TROPES, ce qui exige la mise en œuvre d'une gestion du dialogue entre utilisateurs/concepteurs et modèle de représentation, garant de l'évolution correcte et cohérente d'une base de connaissances.

5.6 Couplage avec un module de gestion de contraintes

Nous avons considéré, jusqu'à présent, que les intensions des entités de représentation correspondent exactement à la structure d'agrégation des attributs des objets, ces attributs étant spécifiés à l'aide de facettes statiques de typage. MICRO existe dans TROPES : il s'agit d'un module de pose et de résolution de contraintes attachées aux objets [Gen95]. Ces contraintes portent exclusivement sur les valeurs des attributs, en impliquant plusieurs attributs, ce qui explique le fait qu'elles ne sont pas attachées à un seul attribut, mais qu'elles sont définies au niveau de la déclaration de l'agrégation de ces attributs.

Les contraintes de MICRO correspondent à des conditions portées sur la valuation des attributs, en particulier lorsque ces conditions ne peuvent être exprimées à l'aide des facettes, et plus précisément lorsqu'il s'agit de conditions qui ne peuvent être vérifiées que par leur évaluation dans un contexte de pose spécifique. Lorsqu'une contrainte n'est pas vérifiée par une instance qui cherche à donner une valeur à un attribut, cela signifie que cette valeur d'attribut n'est pas valide au regard de la description de cet attribut, autrement dit, la valeur n'appartient pas à l'interprétation intensionnelle de l'attribut.

En conséquence, l'existence de contraintes ayant pour effet, entre autres, de restreindre le domaine des attributs, et de ce fait l'interprétation en intension des classes, doit être prise en compte au niveau du typage des entités qui sont censés correspondre exactement à toute interprétation intensionnelle. Les contraintes font en effet partie de la description des objets, et en ce sens, elles doivent être considérées comme faisant partie de l'intension de ces objets.

Nous allons montrer, dans cette section, dans quelle mesure les contraintes de MICRO peuvent être intégrées au typage des entités de représentation, et nous constaterons que cette intégration est limitée, ce qui remet en cause l'isomorphisme établi dans le chapitre précédent, entre l'interprétation intensionnelle d'un base de connaissances et le résultat du typage de cette base.

5.6.1 Contraintes descriptives

Nous ne considérons que l'aspect *descriptif* des contraintes, c'est-à-dire leur rôle au niveau de la restriction des domaines de valeurs. En effet, les contraintes peuvent aussi être considérées comme un mécanisme d'inférence, au même titre que le filtrage (section 2.3.2, page 74). Nous allons montrer ici le rôle tenu par les contraintes au niveau de l'intension des entités de représentation, et en conséquence au regard des relations entre ces entités.

Pose d'une contrainte Micro

Une contrainte peut être posée sur un attribut, entre plusieurs attributs d'une classe, entre plusieurs attributs d'une instance, ou encore entre plusieurs instances d'une classe. MICRO fournit une collection d'opérateurs pré-cablés pour l'expression de contraintes, qui sont fonction du type des objets contraints. Ces opérateurs sont en effet paramétrés par les objets contraints.

Par exemple, il existe un opérateur pour l'expression d'une contrainte entre trois attributs `Entier`, énonçant le fait que la somme des valeurs des deux premiers est égale à la valeur du troisième. Lorsqu'une telle contrainte est posée dans une classe, elle concerne toutes les instances qui se rattacheront à la classe ; lorsqu'elle est énoncée pour une instance, elle ne concerne que cette instance. En outre, une contrainte peut impliquer un ensemble d'instances d'une classe, elle est alors définie au niveau de la classe ; par exemple, il est possible de spécifier le fait que la somme des valeurs des instances, pour un attribut de la classe, est inférieure à un certain seuil.

À chaque opérateur est associée une stratégie de vérification de la cohérence de la contrainte, au regard de sa signification ensembliste[22]. Autrement dit, chaque opérateur d'expression d'une contrainte se voit associer une sémantique ensembliste, et une contrainte est valide à partir du moment où sa résolution ne risque pas de violer les conditions ensemblistes traduites au niveau des types des entités.

Signification d'une contrainte

Une contrainte est une condition, à évaluation dynamique, portée sur les valeurs des attributs des instances. À partir du moment où elle implique plusieurs attributs, elle établit une *dépendance* entre les domaines de valeurs de ces attributs, notamment lorsqu'il s'agit d'une contrainte posée de façon homogène sur toutes les instances d'une classe.

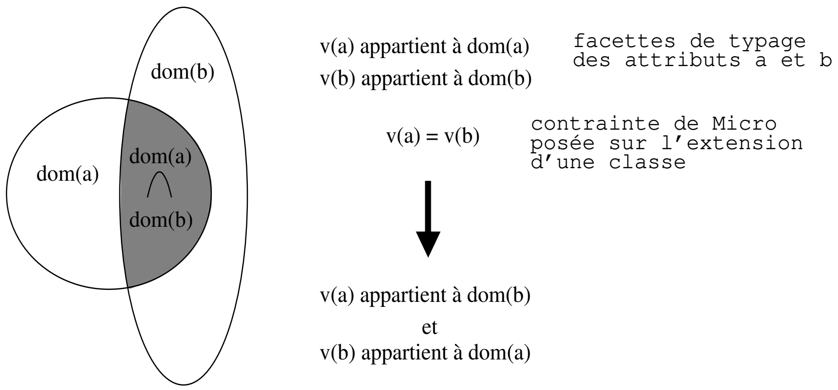


FIG. 5.15 - : Toutes les instances d'une classe sont telles que les valeurs qu'elles donnent aux attributs a et b sont égales. En syntaxe MICRO, cela s'exprime par une contrainte de classe : `mic-eq(a,b)`.

Par exemple, nous voyons sur la figure 5.15 que si les attributs a et b sont contraints à être égaux, cela a une implication au niveau des domaines de valeurs des deux attributs : cette contrainte ne sera jamais vérifiée pour des valeurs de a ou de b n'appartenant pas à l'intersection de leurs domaines de valeurs respectifs. En conséquence, peuvent être exclues de l'interprétation intensionnelle de la classe, toutes les valeurs *records* qui ont des valeurs différentes pour les étiquettes a et b, et en particulier toutes les valeurs *records* donnant une valeur pour a qui n'appartient pas au type de b et vice-versa. Par ailleurs, cette contrainte serait incohérente si l'intersection des domaines de a et de b était vide.

Nous voyons clairement sur cet exemple qu'une contrainte peut avoir des répercussions sur les domaines de valeurs des attributs. En ce sens, ces restrictions doivent être prises en compte au niveau de l'intension des classes, et donc au niveau des types des entités.

[22] MICRO dispose, bien entendu, d'un mécanisme de propagation et de résolution de contraintes, supportant la modification dynamique de ces contraintes, mais nous ne nous intéressons pas ici à cet aspect de MICRO.

5.6.2 Vérifications de cohérence

La technique de vérification de cohérence des contraintes au regard de leur signification ensembliste, est étendue pour réaliser des *inférences* de domaines de valeurs restreints par une contrainte.

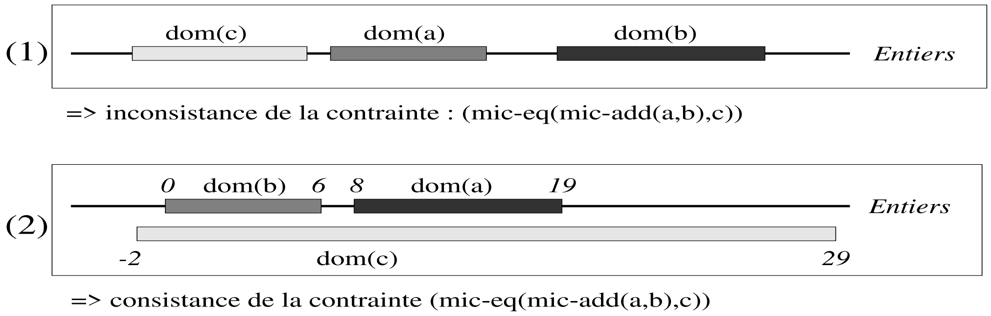


FIG. 5.16 - : Tests statiques de cohérence de contraintes.

MICRO vérifie la cohérence d'une contrainte statiquement, en appliquant la contrainte aux domaines de valeurs des entités contraintes, afin de vérifier qu'ils ne sont pas incompatibles avec la pose de cette contrainte. La figure 5.16 illustre deux cas de vérifications de cohérence d'une contrainte. Le mode d'application de la contrainte `mic-eq(mic-add(a,b),c)` au niveau des domaines des trois attributs impliqués est le suivant : on calcule le domaine dense H tel que $min(H) = min(dom(a)) + min(dom(b))$ et $max(H) = max(dom(a)) + max(dom(b))$, et on vérifie que $H \cap dom(c) \neq \emptyset$. Cette condition quant à l'application de la contrainte aux domaines est directement issue du fait que l'addition est une application monotone croissante.

Plus qu'une vérification de cohérence, on constate que l'application d'une contrainte aux domaines des entités impliquées permet, dans certains cas, de réduire les domaines de ces entités. Par exemple, dans le second cas de la figure 5.16, il est possible de réduire le domaine de c, en appliquant une règle à la contrainte d'addition qui a pour effet d'enlever des domaines donnés avant la pose de la contrainte, les valeurs qui ne pourront jamais être prises sans *obligatoirement* violer la contrainte. Le domaine de c devient ainsi $[8; 25]$. Dans le cas où $dom(a) = [1; 8]$, $dom(b) = [3; 47]$ et $dom(c) = [2; 18]$, les trois domaines sont modifiés, après pose de la contrainte, et l'on obtient $dom(a) = [1; 8]$, $dom(b) = [3; 17]$ et $dom(c) = [4; 18]$. Cette inférence de domaines, dans le cas de l'addition, est obtenue grâce aux propriétés de cette opération au regard des propriétés du type des valeurs auxquelles elle s'applique.

Il existe, pour *chaque opérateur de contrainte*, une règle pour la réduction des domaines, dont le but est d'éliminer toutes les valeurs qui ne pourraient être prises sans violer la contrainte. La règle associée à tout opérateur de contrainte exploite les propriétés de l'opération sous-jacente. Lorsque les domaines associés à des attributs sont modifiés, alors sont ré-activées les vérifications des contraintes impliquant l'un de ces attributs, jusqu'à stabilisation des domaines.

La réduction des domaines des attributs par METÉO, résultant de l'application des contraintes définies à ces attributs, représente une restriction de l'ensemble dénoté par l'intension de ces attributs, et en conséquence une restriction de l'interprétation intensionnelle des classes. En conséquence, ces restrictions doivent être prises en compte dans les types des attributs et dans ceux donc des classes et des instances, ce qui mène à une coopération nécessaire entre les modules MICRO et METÉO.

5.6.3 Coopération Micro/Metéo

Micro, par application des contraintes aux domaines des attributs impliqués, et selon des règles de restriction, détermine de nouveaux domaines pour ces attributs, qui reflètent éventuellement les conditions de restriction des contraintes. Metéo ayant pour rôle, de par les δ-types, de représenter les ensembles de valeurs dénotés par les intensions des entités de représentation, il doit intégrer les trestrictions de domaines calculées par metéo.

Principe

Le principe de la coopération entre Micro et Metéo dans la phase d'élaboration de la représentation, en termes de Eole, de l'interprétation intensionnelle des entités, est relativement simple.

Examinons ce principe dans le cas du domaine de valeurs dénoté par la description d'un attribut a. Metéo détermine, à l'aide des facettes de typage, le domaine de valeurs de a, et le représente par un δ-type t. Micro récupère alors ce δ-type, auquel il applique les règles de restriction de domaines qui correspondent à l'opérateur de la contrainte impliquant a. Il en déduit un nouveau domaine, qui mène à la création d'un nouveau δ-type t' pour a. Si $t' \sqcap_\tau t \neq \bot$, alors la contrainte est valide au regard de a, et Micro termine la vérification de la contrainte courante, qui peut, comme nous l'avons vu, entraîner l'activation d'autres applications de contraintes, ce qui peut bien évidemment mener à déterminer de nouvelles restrictions de domaines pour d'autres attributs, selon le même schéma que celle de a. Une fois que la propagation de Micro s'est stabilisée, et ce sans incohérence, alors Metéo intègre les types des attributs : pour chaque attribut A dont le domaine est modifié par Micro, Metéo type A avec le δ-type résultant de l'intersection entre le type initial de A (avant application des contraintes) et le type pour A calculé par Micro.

La modification des types des attributs entraîne bien entendu celle des types des classes possédant ces attributs.

Dans le cas d'une contrainte posée sur les attributs d'une instance, cela mènera naturellement à recalculer le type de cette instance. Dans le cas d'une contrainte portant sur l'ensemble des instances de l'extension d'une classe, l'intension est elle aussi affinée, par restriction des types des attributs impliqués dans cette contrainte.

Ainsi, Metéo intègre l'information véhiculée par les contraintes de Metéo, relative à l'interprétation intensionnelle des entités, et ce grâce aux calculs de restriction des domaines effectués par metéo. Pourtant, nous analysons dans la section suivante, les limites de cette intégration.

Limites de la coopération

L'intégration, dans Metéo, des restrictions de domaines calculées par Micro n'est pas complète, au sens formel du terme. En effet, Metéo ne considère que l'aspect *statique* des intensions des entités, alors qu'une contrainte représente une description, dont la signification ensembliste ne peut parfois qu'être déterminée par évaluation dynamique de cette description, c'est-à-dire par la considération d'un contexte.

Si l'on reprend l'exemple de la contrainte additionnelle entre les valeurs des attributs a, b, et c, posée sur toutes les instances d'une classe C, qui a mené au calcul, par Micro, des domaines suivants : $dom(a) = [1;8]$, $dom(b) = [3;17]$ et $dom(c) = [4;18]$. Soit une instance i qui cherche à se rattacher à la classe C, telle que $\|i\|^I = \langle a = 2; b = 16; c = 5\rangle$. Au regard de METÉO, `compute-value(`i`)` $\in^{\delta}_{Record}$ `get-type(`C`)`. Pourtant, la contrainte n'est pas respectée, ce qui a pour conséquence le refus, par TROPES, du rattachement de i à C, car la *description* (donc l'interprétation de l'intension) de C n'est pas respectée.

Cet exemple illustre clairement le fait que METÉO, qui réalise un typage statique des entités, n'est plus isomorphe à l'interprétation intensionnelle d'un base de connaissances, lorsqu'il est possible de décrire dans la base des contraintes dynamiques sur les objets. En particulier, METÉO peut décider de la validité de certaines assertions que MICRO réfutera. Pourtant, à partir du moment où METÉO réfute une assertion, au regard des types, MICRO la réfutera nécessairement.

5.6.4 Conclusion

TROPES comporte MICRO, un module de gestion de contraintes, qui permet l'expression contrôlée de contraintes dynamiques sur les objets. Ces contraintes ont une composante descriptive, donc intensionnelle, qui ne peut être intégralement prise en compte dans METÉO. En conséquence, en présence de MICRO, METÉO n'offre plus une algèbre isomorphe à celle qui correspond à l'interprétation intensionnelle d'une base de connaissances.

De ce fait, toute validation par METÉO devra être validée par MICRO pour être valide intensionnellement, et ensuite être éventuellement validée extensionnellement. Par contre, une réfutation par METÉO est définitive. Malgré tout, le couplage de ces modules permet une exploitation statique maximale des contraintes autres que celles énoncées au sein des facettes, ce qui est tout de même un résultat non négligeable.

Nous retiendrons que l'interprétation intensionnelle d'une base de connaissances TROPES est réalisée conjointement par MICRO et METÉO.

5.7 Conclusion

Ce chapitre a illustré, dans le cas de TROPES, qui peut être généralisé à tous les modèles ne supposant pas l'équivalence entre extensions et intensions des entités de représentation, que les deux interprétations des entités, et des relations entre ces entités, doivent être considérées conjointement et non pas l'une indépendamment de l'autre. En effet, si certaines inférences ou vérifications peuvent être effectuées intensionnellement, elles doivent toujours être validées du point de vue de l'existence d'une signification dans le monde modélisé.

Nous avons esquissé une architecture d'un module de coopération entre mécanismes intensionnels et extensionnels, et nous retiendrons le fait qu'un modèle de connaissances doit intégrer, au niveau du comportement de ses mécanismes, une recherche de l'équivalence entre extensions et intensions, qui ne peut être effectuée que par un dialogue avec l'utilisateur, qui est le seul médiateur entre le monde modélisé et la représentation qui en est faite.

La distinction entre processus intensionnels et extensionnels a été rendue plus nette grâce à la distinction des domaines d'interprétation des objets, matérialisée par l'existence de METÉO.

Nous constatons, à l'issue de l'étude des mécanismes définis dans un modèle de connaissances à objets, que le rôle mené par les interprétations extensionnelles est essentiel, notamment en ce qui concerne la recherche de la justesse des intensions, ce qui devrait être un des buts de ces modèles. Cela nous amène naturellement à nous interroger quant à la pertinence du développement de structures de données en termes du langage de représentation, alors qu'elles n'ont pas, dans l'absolu, de signification dans le monde modélisé.

Nous avons vu que toute inférence intensionnelle, donc effectuée sur les descriptions des entités de représentation, doit être validée extensionnellement pour être intégrée à la base de connaissances. Imaginons maintenant que l'on étend le langage de représentation afin de permettre le développement de structures de données, et que l'on crée ainsi la classe `Matrice`. Compte-tenu du comportement du modèle de connaissances, chaque création d'une instance de la classe `Matrice` devra être validée par l'utilisateur afin d'être intégrée à la base, ou encore, tout calcul menant à la modification d'une telle instance devra être validé par l'utilisateur, avec comme question sous-jacente "cette instance a-t-elle une signification dans le monde modélisé, correspond-elle réellement à une matrice ?".

Cet exemple, certes naïf, illustre toutefois la perversité de représenter, à un même niveau, des connaissances qui ne relèvent pas du même degré d'abstraction, tout simplement parce qu'elles se voient associer les mêmes mécanismes d'exploitation, avec les mêmes objectifs. Si l'intérêt de la représentation de structures de données ne réside que dans le fait que les opérations sur ces structures peuvent correspondre à la modélisation de phénomènes réels, mêmes élémentaires, nous proposons de les développer au sein de METÉO, c'est-à-dire dans l'absolu. C'est alors le fait de rattacher, à un objet ayant une signification dans le monde réel, le résultat d'une opération sur une *donnée*, qui confère à ce résultat une signification dans ce même monde. Parce que `6` n'est d'abord qu'une courbe, qui prend une première signification quand on l'associe au type `Entier`, qui symbolise alors une marque du temps quand on l'associe à l'attribut `âge` d'une classe, ou qui représente un fort tremblement de terre s'il est associé à un degré de l'échelle de Richter, on ne peut parler de la signification "unique" de `6`. Ces multiples significations pour une même "description" sont en opposition avec l'un des objectifs de la représentation des connaissances, qui est d'être une assistance à l'élaboration de l'équivalence entre la description d'un objet et sa signification dans le monde réel, et ce compte-tenu d'une pré-abstraction de ce monde.

Chapitre 6

Exploitation des types abstraits

Les chapitres 4 et 5 ont montré comment MÉTÉO peut être utilisé par TROPES à des fins de vérifications de correction et cohérence de l'interprétation intensionnelle d'une base de connaissances. En particulier, il a été montré que les δ-types, leurs propriétés et leur organisation, constituent un support adéquat pour la mise en œuvre de tels processus.

Nous proposons, dans ce chapitre, de montrer l'intérêt des C-types pour une modélisation en TROPES, en ce qui concerne le développement de nouvelles structures de données, éventuellement directement issues de la description des entités de représentation. En effet, les C-types permettent la spécification et l'implémentation du comportement de leurs valeurs.

6.1 Problématique

Le chapitre 2 a fait état, dans la section 2.4, de la nécessité pour un système de représentation d'intégrer un module dédié à la spécification et à l'implémentation de structures de données qui n'ont pas de correspondance dans le monde modélisé. En effet, une base de connaissances peut contenir des modélisations du monde qui, elles-mêmes, bénéficient d'une modélisation mathématique connue, que l'on veut pouvoir intégrer à la base de connaissances. Par exemple, un mouvement de rotation, qui peut faire partie de la caractérisation d'une classe d'objets, peut à son tour être modélisé à partir de divers formalismes mathématiques, comme la définition d'une matrice de rotation, ce qui implique le développement de la structure de données `Matrice`. Nous considérons que la représentation de telles structures, à vocation calculatoire, *ne doit pas* être faite en termes du langage de représentation des connaissances, et ce pour deux raisons majeures, non indépendantes, et toutes deux liées à l'interprétation faite de telles structures.

- Une matrice, comme toute autre *donnée* structurée, et ce au même titre qu'une valeur, n'a pas véritablement de signification dans le monde modélisé : c'est son rattachement à un objet représentant une partie de ce monde modélisé, qui lui confère une signification particulière (section 2.4.1). En ce sens, il n'est pas pertinent de représenter une matrice par un objet de la base de connaissances. En effet, elle serait alors interprétée intensionnellement *et* extensionnellement, dans le même univers où le serait une `protéine`; il est pourtant évident qu'il s'agit de deux modélisations réalisées à des niveaux d'abstraction différents.
- Nous parlons ici de structures de données dont la principale raison d'être provient des opérations qui leur sont appliquées, et de la sémantique de ces opérations. Par exemple, une matrice

est une structure adaptée à la modélisation de nombreux phénomènes fondés sur le nombre de dimensions (orthogonales ou non) impliquées, et les opérations sur des matrices servent à la modélisation de la simulation d'une superposition de ces phénomènes, et c'est parce que ces simulations sont ainsi possibles que le concept de matrice est intéressant. Nous assistons alors à deux niveaux de définition sémantique :

1. d'une part, il s'agit de définir le mode de production du résultat (direct ou indirect) d'une opération, et ce, de par une sémantique opérationnelle classique, qui indique quelle manipulation est faite des données en entrée de l'opération[1],
2. d'autre part, il s'agit d'interpréter le résultat d'une telle opération lorsqu'elle est activée sur des données qui ont une interprétation dénotationnelle dans un univers particulier[2].

Le modèle de connaissances a certes pour rôle de permettre la seconde de ces interprétations, mais pas la première, qui appartient à la définition d'une structure de données, et doit donc être valide dans tous les mondes possibles.

Un modèle de connaissances n'est donc destiné qu'à la modélisation directe d'un domaine d'application, et en aucune façon il ne doit être un support pour l'implémentation de structures de données, sous peine de devoir modifier la sémantique dont il est muni, qui est réservée à l'interprétation des objets dans le monde modélisé. En particulier, et c'est une des différences à signaler d'avec les langages de programmation par objets, un langage de représentation des connaissances doit éviter l'écriture de méthodes de classes qui n'exploitent que la structure de description d'une classe, sans pour autant correspondre à la modélisation du comportement des objets du monde modélisé. En effet, la description (l'état) d'une classe de représentation étant la caractérisation, vis-à-vis du monde modélisé, des objets qu'elle regroupe, et non pas la génératrice d'une structure instanciable et manipulable, une méthode de classe devrait aussi correspondre à la modélisation du comportement des dénotations de ses objets.

Il est analysé, dans ce chapitre, dans quelle mesure METÉO peut être un support pour la spécification et l'implémentation de structures de données exportables dans TROPES. La section suivante concerne ainsi l'exploitation qui peut être faite par TROPES de la propriété d'extensibilité de METÉO. Nous développerons dans la section 6.3 un exemple complet de cette propriété, pris dans le domaine de la biologie moléculaire. Nous verrons ensuite, dans la section 6.4, que le C-typage d'une classe permet à un modèle de connaissances de se décharger de la spécification et de l'implémentation de toute opération qui, sans être interprétable dans le monde modélisé, ne peut se définir qu'à partir de la structure de description d'une classe. Cela nous permettra, en conclusion, de mettre en évidence un mode de coopération des langages de programmation par objets (pour lesquels les classes sont l'implémentation de types abstraits de données) et des langages de représentation par les *frames*, par confrontation de leurs sémantiques respectives.

6.2 Exportation de structures et opérations

METÉO comporte une base ordonnée de C-types qui, au regard de TROPES, sont des types abstraits de données, chacun définissant de ce fait une structure élaborée de données et un ensemble d'opérations de manipulation et d'exploitation de ces structures. Ces opérations, lorsqu'elles sont activées sur des valeurs d'un C-type[3], sont exportées dans TROPES, où elles peuvent être utilisées

[1] Comment procède-t-on pour calculer la somme de deux matrices ?

[2] Soient deux matrices carrées qui représentent chacune un mouvement de rotation dans le plan, que signifie le fait d'additionner ces deux matrices ?

[3] Ces opérations sont à distinguer de celles qui représentent le comportement des δ-types.

dans un environnement de contrôle établi et défini par le modèle de connaissances lui-même. L'extensibilité de METÉO traduit une ouverture de ce système quant à l'intégration, par le biais des C-types, de nouvelles structures de données qui peuvent de ce fait être utilisées par des applications développées dans TROPES.

6.2.1 Opérations sur des valeurs de C-type

Un C-type définit deux niveaux d'opérations, celles manipulant les δ-types issus de ce C-type, et celles définissant l'exploitation faite des valeurs de ce C-type. Nous nous intéressons ici, et tout au long de ce chapitre, à cette seconde catégorie d'opérations.

METÉO possède la propriété de *polymorphisme d'inclusion*. En ce sens, les opérations définies sur les valeurs d'un C-type, le sont aussi sur les valeurs de ses C-sous-types, avec une redéfinition éventuelle. Pour cette raison, et pour permettre la sélection d'une opération activée sur une ou plusieurs valeurs, lorsqu'elle est définie dans plusieurs C-types, chaque opération est paramétrée par le C-type qui la définit. La signature de chaque opération est donnée dans la spécification abstraite du C-type, elle peut impliquer plusieurs autres C-types, à la manière des algèbres de sortes. Par exemple, le C-type `réel` définit l'opération d'arrondi d'une valeur réelle vers une valeur entière, sa signature est donc : `mt-trunc : Réel` $\mapsto$ `Entier`.

Toutes les opérations définies sur des valeurs ont une signature respectant un format uniforme, de façon à ce que leur sélection et activation puissent être gérées globalement par METÉO, et non pas localement dans chaque C-type. En particulier, cette gestion globale comprend la vérification statique du bon typage des appels externes à ces opérations. Le format de la signature de toute opération activée sur des valeurs de C-type est le suivant :

$$\texttt{mt-nom-opération}(T_D, (T_i)^*) \mapsto T_R$$

Où T_D est le C-type qui définit l'opération spécifiée, $(T_i)^*$ est la liste ordonnée (éventuellement vide) des types des paramètres en entrée, et T_R est le type d'un éventuel résultat. Dans la suite de cet exposé, lorsqu'il n'y aura aucune ambiguité possible, nous ne spécifierons pas le premier de ces paramètres, c'est-à-dire celui qui indique le C-type propriétaire de l'opération.

Par ailleurs, la sémantique d'une opération d'un C-type est elle aussi donnée par la spécification (axiomatique ou équationnelle) du C-type. Sa réalisation vérifie alors cette spécification. Soit $\mathcal{O}_v$ l'ensemble des opérations définies par les C-types sur l'ensemble $\mathcal{V}_\tau$ des valeurs de ces C-types. Le code d'une opération de $\mathcal{O}_v$ est écrit en termes du langage hôte, dont la bibliothèque de fonctions est justement complétée par les opérations de $\mathcal{O}_v$.

C-sous-typage et opérations

Au même titre que les opérations de manipulation des δ-types, les opérations définies sur les valeurs d'un C-type, le sont aussi pour les valeurs de ses C-sous-types. Cette propriété est un gage de modularité et de réutilisabilité, elle est en outre très intéressante au regard de l'ajout de nouveaux C-types, car cela permet à l'utilisateur de récupérer l'implémentation d'une opération dont la spécification correspond à celle d'une autre opération définie dans le C-sur-type donné.

Par exemple, si le C-type `EntierPair` est ajouté par l'utilisateur, ce dernier n'aura pas à redéfinir l'opération d'addition. En effet, cette opération est définie dans le C-type `Entier`, où elle est spécifiée à partir des opérations `successeur` et `prédécesseur`. Ainsi, l'utilisateur n'aura qu'à redéfinir ces deux dernières opérations, et l'addition correspondra naturellement.

Il s'agit, là encore, d'une exploitation de la propriété de polymorphisme d'inclusion que possède METÉO.

Opérations sur les C-types primitifs

Les C-types primitifs sont ceux dont la définition est reprise de celle considérée par le langage hôte, complétée et adaptée pour être intégrée à METÉO[4]. De ce fait, METÉO intègre aussi les opérations existant dans le langage hôte sur les valeurs de ces types, dont il change la signature afin qu'elle corresponde au format requis.

Les opérations sur des valeurs primitives, traduites au format de METÉO, remplacent littéralement leurs équivalentes définies dans le langage hôte, à savoir que celles du langage hôte ne pourront plus être utilisées dans METÉO à partir du moment où elles s'y voient associer une opération dans METÉO.

Opérations sur les constructeurs

METÉO définit, pour chacun des C-types assimilé à un constructeur, un ensemble d'opérations qui exploitent justement le mode de construction des valeurs. Par exemple, le C-type correspondant au constructeur `Liste` définit la fonction `mt-get-element(Liste(`X`),Entier)`$\mapsto X$, qui, à partir d'une liste l d'éléments d'un C-type X, et à partir d'un entier positif n, renvoie le n-ième élément de l.

Par ailleurs, lorsque le constructeur est appliqué à un (des) C-type(s) donné(s), d'autres opérations y sont définies, qui intègrent certaines propriétés et opérations du (des) C-types référencé(s). Par exemple, le C-type correspondant au constructeur `Liste(Booléen)` définit l'opération `mt-and-list(Liste(Booléen))` $\mapsto$ `Booléen`, qui applique l'opération `mt-and(Booléen,Booléen)` $\mapsto$ `Booléen` récursivement à tous les éléments de la liste en paramètre.

Toute opération de $\mathcal{O}_v$ définie pour un constructeur en termes du langage hôte, ne doit utiliser que des opérations de $\mathcal{O}_v$, et pas d'opérations définies directement par le langage hôte sur ses propres valeurs et données structurées.

Opérations sur les C-types ajoutés

Lorsque l'utilisateur définit un nouveau C-type, il doit aussi donner l'implémentation des opérations de manipulation de ses valeurs, opérations qu'il a préalablement spécifiées.

Le code de ces opérations, comme toutes celles de $\mathcal{O}_v$, est écrit en termes du langage hôte, et à partir des autres opérations de $\mathcal{O}_v$[5].

Une fois les opérations d'un nouveau C-type exprimées au format de METÉO, elles seront considérées par METÉO au même titre que n'importe quelle autre opération de $\mathcal{O}_v$. Soit o_T l'ensemble

[4] Il s'agit essentiellement des C-types `Entier`, `Réel`, `Booléen`, `Chaîne` et `Caractère`.

[5] Si le nouveau C-type reprend un type de données T du langage hôte qu'il se contente d'adapter au format requis par METÉO, il peut utiliser des opérations primitives du langage hôte, lorsqu'elles correspondent à des opérations de manipulation des valeurs de T.

des opérations de manipulation des valeurs d'un nouveau C-type T. Une fois T intégré à METÉO en tant que C-type, $\mathcal{O}_v$ est enrichi :

$$\mathcal{O}_v \longleftarrow (\mathcal{O}_v \cup o_T)$$

Cet enrichissement de $\mathcal{O}_v$ fait partie de la phase de personnalisation de METÉO lors du développement d'une base de connaissances particulière, puisqu'il accompagne naturellement l'augmentation de la base des C-types.

6.2.2 Exportation des opérations de Metéo

Schéma de l'exportation

Nous avons vu dans les chapitres 1 et 2 qu'un modèle de connaissances possède des mécanismes permettant d'intégrer les résultats de processus calculatoires activés sur des entités de représentation. Le modèle de connaissances garde toujours le contrôle de l'exécution de ces processus et de l'intégration de leurs résultats et/ou effets de bord. Ces mécanismes sont principalement :

- le détachement procédural : écrit intégralement en termes du langage hôte (ex. page 73, l'exemple de la figure 2.11 montre que le calcul du nombre d'étapes d'un voyage correspond à une fonction écrite en termes du langage hôte),
- le filtrage : écrit en termes de TROPES, fait appel à des fonctions sur des données, définies dans le langage hôte (ex. page 75, sur l'exemple de la figure 2.13, le filtre utilise le prédicat d'egalité entre deux valeurs),
- la pose et la résolution de contraintes : écrit en termes de MICRO, fait appel à des fonctions sur des données, définies dans le langage hôte.

De façon générale, il s'agit des mécanismes qui sont à l'origine de l'inférence de valeurs d'attributs pour des instances particulières. Dans la plupart des langages de *frames*, les fonctions dédiées à l'inférence de valeurs d'attributs sont écrites en termes du langage hôte. Plus précisément, les attachements procéduraux sont des opérations du langage hôte, et le filtrage peut utiliser des fonctions primitives du langage hôte pour la manipulation de données justement primitives. En conséquence, la collection d'opérations disponibles pour ces processus est limitée. On mesure alors ici toute la puissance de l'extensibilité de METÉO, et l'importance du traitement homogène des C-types. En effet, toutes les opérations de $\mathcal{O}_v$ étant considérées uniformément par METÉO, elles sont toutes exportées dans TROPES selon le même schéma, qu'il s'agisse d'opérations activées sur des valeurs de type primitifs, construits ou définis par l'utilisateur.

Avec la présence de METÉO, les mécanismes d'inférences utilisent directement les opérations définies, au sein des C-types, sur les valeurs de ces C-types (figure 6.1). Notons d'ailleurs que le code de ces opérations est toujours écrit en termes du langage hôte, la différence étant la signature qui doit répondre au critère d'homogénéité imposé par METÉO en vue d'un meilleur contrôle de l'application de ces opérations.

L'activation d'une opération définie dans METÉO nécessite l'interprétation des entités de représentation qui sont en paramètres de la fonction, vers l'univers de valeurs de METÉO, et ce grâce à la fonction d'interprétation des valeurs `compute-value`.

La section 6.3 présente un exemple complet d'utilisation de l'extensibilité de METÉO et de l'exportation des opérations sur les valeurs de C-type.

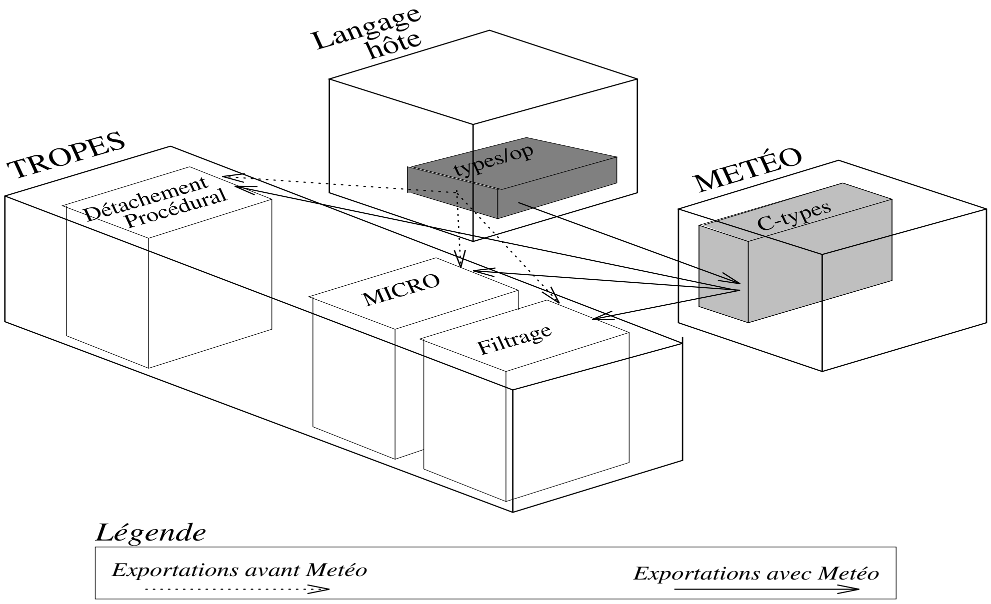


FIG. 6.1 - : Exportation des opérations de METÉO dans TROPES.

Vérifications de types

Un autre avantage de cette homogénéité dans l'activation d'opérations externes à TROPES réside dans la possibilité de réaliser des vérifications statiques de types, dans le but d'éviter l'application incohérente d'opérations à des paramètres non adaptés du point de vue des types de la signature des opérations. En effet, METÉO dispose d'une part des signatures de toutes les opérations de $\mathcal{O}_v$, et d'autre part il détient les types des entités de représentation. Le contrôle de ces vérifications est effectué par les modules de TROPES chargés de contrôler justement l'activation des mécanismes d'inférences. Par exemple, lors d'un détachement procédural de la forme `aire $un Entier $sib-exec mt-mult(Entier,*.longueur,*.largeur)`, si `longueur` et `largeur` sont deux attributs de types `Entier`, alors il est vérifié d'une part que le type des deux paramètres de `mt-mult` dans `Entier` sont bien des `Entiers`, et que le type du résultat est aussi un `Entier`, afin que cela corresponde au typage des trois attributs impliqués.

Il s'agit donc ici d'une réelle contribution de METÉO, et du fait qu'il interprète les descriptions des entités de représentation, car le système de types du langage hôte, en supposant qu'il existe, ne permettrait pas de telles vérifications, dans la mesure où il ne peut justement pas interpréter le typage des entités de TROPES.

En outre, le polymorphisme d'inclusion de METÉO est aussi sollicité, lors de ces vérifications de types. Soit `mt-fonction(`T, t`)` une opération définie par le C-type T, dont le paramètre appartient au δ-type t, et soit T' un C-sous-type de T, alors `mt-fonction` est activable pour tout paramètre dont le δ-type t' est issu de T', à condition cependant que $t' \leq_\gamma t$. Autrement dit, si cette opération est activée comme suit :

$$\texttt{mt-fonction(}T', v\texttt{)} \text{ avec } \texttt{get-type(}v\texttt{)} = t'$$

alors il est vérifié, d'une part que $T' \leq_C T$ et d'autre part que $t' \leq_\gamma t$. METÉO généralise cette vérification à plusieurs paramètres en entrée, et à un paramètre en sortie, en appliquant la contravariance aux tests de vérifications de types.

6.2.3 Conclusion

Les C-types définissent des opérations sur les valeurs qui leur appartiennent, ces opérations pouvant impliquer des valeurs d'autres C-types. Ces opérations, pour être exportées dans TROPES doivent être définies avec une signature spécifiée dans une syntaxe homogène, qui garantit la validité des processus de vérifications et d'application.

Les opérations sur les valeurs des C-types sont exportées, dans TROPES, au niveau du détachement procédural, du filtrage et de la pose de contraintes. Toutefois, le contrôle de l'intégration du résultat de l'activation d'une telle opération, lorsqu'elle est associée à des objets de représentation, est effectué par TROPES. En particulier, ces opérations ne peuvent pas, d'elles mêmes, modifier un ou plusieurs objets, car elles ne concernent qu'une manipulation externe d'une partie de la description de ces objets. En effet, nous avons vu que toute opération effectuée sur les descriptions des entités de connaissances ne sont pas forcément valides, du fait de la non-équivalence des interprétations intensionnelle et extensionnelle d'une base de connaissances, et parce que l'apport d'une nouvelle connaissance doit être validée extensionnellement (il s'agit de s'assurer que cette connaissance peut être interprétée dans le monde modélisé, au regard de son contexte d'élaboration). Ce n'est donc pas parce qu'un résultat est valide compte-tenu des descriptions des entités, qu'il est cohérent au regard du monde modélisé, et c'est pourquoi le contrôle de son intégration doit être laissé au modèle de connaissances, éventuellement par soumission du résultat à l'utilisateur.

6.3 Exemple en biologie moléculaire

Afin d'illustrer l'utilité de METÉO dans le développement d'applications pour lesquelles la manipulation de structures de données complexes s'intègre à la modélisation, nous avons choisi d'analyser un exemple en biologie moléculaire, et plus précisément en analyse de séquences génomiques, domaine d'application pour lequel certains modèles mathématiques ont déjà été développés, et qu'il est important de pouvoir faire figurer dans la modélisation de ce domaine. Cet exemple illustre plus particulièrement la distinction entre des structures de données dépourvues de signification, et des descriptions d'objets interprétables dans le domaine modélisé.

Le but de cet exemple est donc d'illustrer simplement le fait que la distinction entre objets de représentation et structures de données est naturelle d'un point de vue conceptuel, mais aussi que METÉO est un cadre adapté à l'explicitation de cette différence. Ainsi, au niveau de TROPES sont modélisés les objets ayant une signification directe en biologie, tandis que dans METÉO, est développée la structure de données `séquence` avec quelques opérations sur ces séquences pouvant être utilisées, en tant qu'opérations sur des valeurs, dans la description des objets de représentation. METÉO évite de cette façon que le langage de représentation de TROPES ne s'adapte à la définition complète et efficace de la structure de données `séquence` qui est pourtant une structure de base dans le processus de modélisation mathématique de maints phénomènes biologiques.

6.3.1 Le domaine d'application : analyse de séquences génomiques

But de la modélisation

Pour le développement de cet exemple, nous nous sommes inspirés de l'important travail de modélisation effectué par Guy Perrière, qui concerne la modélisation, en SHIRKA, de l'expression des gènes chez la bactérie *Escherichia coli* [Per92a].

Nous cherchons plus précisément à représenter le lien qui existe entre les notions de gène et de protéine, à savoir qu'un gène code systématiquement une et une seule protéine.

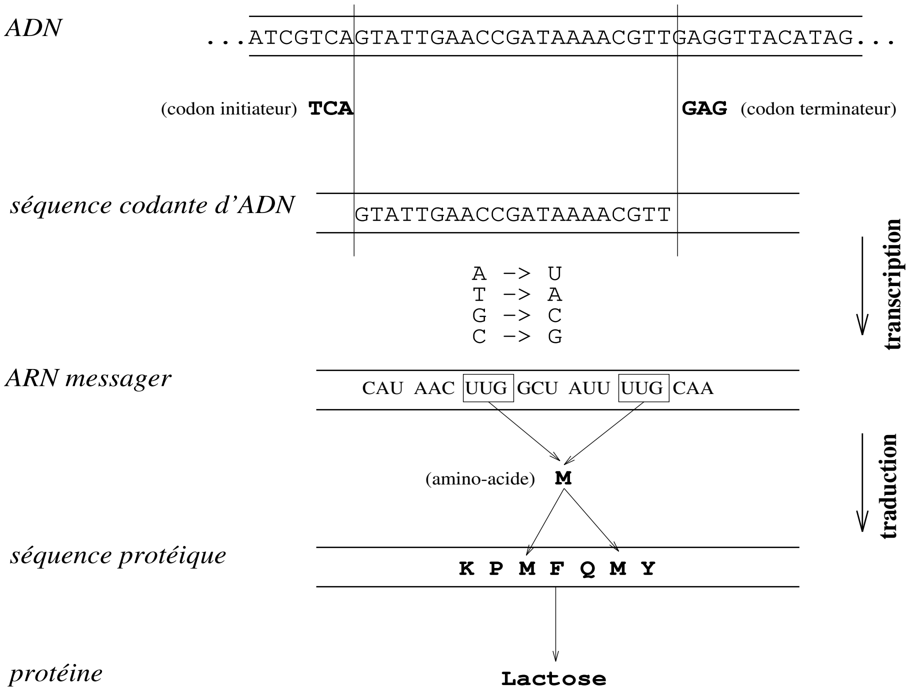


FIG. 6.2 - : Modélisation du processus d'obtention d'une protéine à partir de la séquence d'ADN du gène qui la code.

À chaque gène correspond ce qui est appelé une *séquence génomique* unique, elle-même constituée par un enchaînement de quatre motifs élémentaires (les quatre bases nucléotidiques que l'on trouve au niveau de l'ADN) communément codés par les caractères 'A', 'T', 'G' et 'C'. Cette séquence génomique contient un ensemble d'informations propres au gène auquel elle correspond, et en particulier elle détient l'information qui permet d'inférer sa protéine associée. Il existe un mode d'extraction de cette protéine à partir de la séquence génomique s_g du gène g. Le principe simplifié, illustré par la figure 6.2, est le suivant :

1. Recherche de la séquence codante s_c de s_g, qui est la plus longue sous-séquence de s_g comprise entre un *codon initiateur* et un *codon terminateur*. Les deux codons sont chacun une séquence de trois caractères pris parmi 'A', 'C', 'T' et 'G', et ne sont pas spécifiques au gène, ils correspondent à des méthodes générales d'extraction de séquences codantes.

2. Transcription de la séquence s_c obtenue dans la phase précédente, vers une séquence d'ARN messager. Cette transcription consiste à réaliser la complémentarité des molécules, et se traduit, au niveau des séquences de caractères, en changeant, dans la séquence s_c, le caractère 'A' par le caractère 'U', 'T' par 'A', 'G' par 'C' et 'C' par 'G'. On obtient ainsi la séquence d'ARN messager, codée par une séquence de caractères pris parmi 'A', 'C', 'G' et 'U', que l'on note s_a.

3. Traduction de la séquence d'ARN messager s_a vers une séquence protéique, constituée d'un enchaînement ordonné d'amino-acides. Il existe en tout vingt amino-acides, chacun étant codé par un caractère. Cette phase de traduction s'opère par découpage de s_a en sous-séquences de trois caractères chacune. À chacune de ces sous-séquences correspond, via le code génétique un amino-acide qui est donc produit par simple association. On obtient à l'issue de la traduction

une séquence de caractères pris parmi un alphabet de vingt caractères[6], nous notons cette séquence s_p, il s'agit d'une séquence protéique.

4. Sélection de la protéine correspondant à s_p. En effet, chaque protéine est caractérisée de façon unique par une séquence protéique.

Sans vouloir représenter le processus complet qui a pour effet d'associer une protéine à une séquence génomique, nous cherchons à faire figurer les différentes étapes de ce processus au sein même de la description d'un gène et/ou d'une protéine. Ainsi, chaque étape du calcul, qui va être effectuée sur des structures de données définies dans MÉTÉO, verra son résultat interprété dans TROPES, et ce grâce au rattachement du résultat à un attribut porteur d'une signification dans le monde biologique.

Principe de la modélisation

Pour transcrire la modélisation du processus de production d'une protéine à partir de la séquence d'ADN du gène qui la code, nous allons tout d'abord créer, dans MÉTÉO, les structures de données et les opérations nécessaires.

Nous définirons ensuite la modélisation, sous formes de classes TROPES, des notions biologiques telles que `Gène-protéique` et `Protéine`. Nous spécifierons, dans ces classes, des attributs porteurs de l'information relative aux différentes étapes du processus, en nous assurant que chacun des attributs mis en évidence a une signification au regard de l'objet biologique qui le contiendra.

Nous attacherons alors, à chacun des attributs, une méthode adaptée pour l'inférence de sa valeur.

6.3.2 Définition de nouveaux C-types

Afin de représenter les codages des séquences biologiques, nous définissons tout d'abord le C-type `Seq`, puis trois autres types de séquences, correspondant respectivement aux séquences protéique, d'ADN et d'ARN messager.

Le C-type `Seq`

Le C-type `Seq` est en réalité un constructeur unaire, au même titre que les constructeurs `Liste` et `Ensemble`. Nous cherchons donc à le définir comme un C-sous-type de `Multi-valués`, ce qui nous permet, entre autres, de récupérer l'information relative à la cardinalité des valeurs construites.

Conceptuellement, une séquence est une liste dont tous les éléments se voient associer une position unique dans cette liste. De ce fait, toutes les opérations existant sur les listes peuvent être associées aux séquences, et cette notion de position n'influe pas sur le mode de représentation de sous-ensembles de séquences, à savoir des δ-types issus du C-type `Seq`. Par conséquent, nous définissons le C-type `Seq` comme un **C-sous-type direct de `Liste`**.

- Les prédicats d'appartenance d'une valeur à une séquence, et d'égalité entre deux valeurs séquences, sont, en pratique, les mêmes que ceux définis pour les listes. En théorie, cependant,

[6] Il s'agit usuellement des caractères 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y'.

le prédicat d'appartenance devrait prendre en considération la position associée à chaque élément dans une séquence, et en conséquence, le prédicat d'égalité devrait aussi porter sur l'egalité des positions pour chaque élément testé. Nous considérons par la suite que ce sont les opérations de manipulation définies pour les séquences qui indiquent la différence d'avec de simples listes.

– Toutes les opérations portant sur le mode de représentation des δ-types issus de `Seq` sont toutes reprises de celles de `Liste`, sans aucune modification, dans la mesure où ce mode de représentation n'a pas changé. Ceci est possible grâce à la propriété de polymorphisme d'inclusion que possède METÉO.

– Le C-type `Seq` bénéficie de toutes les opérations définies sur les valeurs listes, comme par exemple :

  – `mt-concat(Liste,⌈Liste(X)⌉,⌈Liste(X)⌉)` $\mapsto$ `⌈Liste(X)⌉`, qui réalise la concaténation de deux listes,

  – `mt-get-card(Liste, ⌈Liste(X)⌉)` $\mapsto$ $\langle$`Entier`$;[0;+inf]\rangle$, qui rend la cardinalité de la liste en paramètre,

  Ces fonctions, définies dans `Liste`, pourront être appliquées à des séquences, puisque `Seq` $\leq_C$ `Liste`.

– `Seq` définit en outre une batterie d'opérations exploitant cette notion de position d'un élément, comme par exemple :

  – `mt-extract-el(Seq,` $\langle Entier;[0;+inf]\rangle$`, ⌈Seq(X)⌉)` $\mapsto$ `⌈X⌉`, qui rend l'élément d'une séquence dont la position est celle donnée par le paramètre entier,

  – `m-extract-seq-el(Seq,` $\langle Entier;[0;+inf]\rangle$`,`$\langle Entier;[0;+inf]\rangle$`, ⌈Seq(X)⌉)` $\mapsto$ `⌈ Seq(X) ⌉`,; si n_1 est le premier paramètre entier, si n_2 est le second paramètre entier, et si s est la séquence en paramètre, alors cette fonction a pour résultat la sous-séquence de s qui débute à la position n_1 et qui a pour longueur n_2.

  Les fonctions propres à `Seq` sont toutes définies les unes en fonction des autres ; elles peuvent aussi faire référence à des fonctions de ses C-sur-types, et en particulier à des fonctions de `Liste`.

La création du C-type `Seq`, qui est un constructeur unaire, entraîne automatiquement la création des C-types `Seq(Caractère)`, `Seq(Entier)`, `Seq(Réel)`, etc. (section 3.6.3). Rappelons que cette définition automatique consiste à adapter les prédicats d'appartenance et d'égalité d'une valeur au C-type.

Par ailleurs, comme nous l'avons vu dans la section 4.4.6, il s'agit d'étendre la syntaxe du langage de représentation de TROPES afin de permettre, lors du développement de base de connaissances, la prise en compte de ce nouveau constructeur. Ce constructeur sera ainsi référencé par la facette de typage \$seq-de. Étant donné que le mode de représentation des séquences ne diffère pas de celui des listes, il n'y a pas lieu d'adapter le processus de typage d'entités de représentation : un attribut typé comme une séquence le sera selon le même schéma que s'il l'était par une liste, il suffit de rajouter à la table de la page 141, les trois lignes suivantes :

\$seq-de	Concept	$\Longrightarrow$`Sequence(Record)`
\$seq-de	Liste de Classes	$\Longrightarrow$`Sequence(Record)`
\$seq-de	Identificateur de type de base T	$\Longrightarrow$ `Seq(`$ADT(T)$`)`

Nous disposons donc maintenant du C-type `Seq` et `Seq(Caractère)` est automatiquement défini, qui "hérite" toutes les opérations disponibles dans `Seq`. Nous pouvons maintenant affiner encore le C-type `Seq(Caractère)` pour définir les C-types qui correspondent à la modélisation des séquences biologiques.

Le C-type `Séquence de 'A', 'C',...,'Y'`

Nous définissons le C-type `Seq(carac20)` chargé de représenter le codage des séquences protéiques, comme un C-sous-type de `Seq(caractère)`, telle que les séquences construites ne le soient qu'à partir des caractères correspondants à un acide-aminé.

La création de `Seq(carac20)` nécessite celle du C-type `carac20`, comme l'ensemble des caractères pris parmi 'A' 'C' ... 'W' 'Y'. Ce dernier est créé comme C-sous-type de `Caractère`, dont il ne redéfinit que le prédicat d'appartenance, comme suit :

$$\in_{carac20)} (v) \Longleftrightarrow v \in^{\delta}_{carac20} \langle Caractere; [A; A] + [C; C] + \cdots + [Y; Y] \rangle$$

À partir de `carac20`, nous créons donc `Seq(carac20)`, dont le prédicat d'appartenance est issu de celui de `carac20`: tous les éléments d'une séquence de `Seq(carac20)` doivent appartenir à `carac20`.

Les C-types `Séquence de 'A', 'C', 'T', 'G'` et `Séquence de 'A', 'C', 'U', 'G'`

Nous définissons de la même façon que `Seq(carac20)` les deux C-types qui correspondent aux codages respectifs des séquences d'ADN et d'ARN messager : seul le prédicat d'appartenance d'une valeur à chaque C-type est redéfini, à partir des prédicats d'appartenance des C-types `ACGT` et `ACGU`, créés comme C-sous-types `Caractère`. Nous pouvons remarquer qu'il aurait été possible de spécifier `ACGT` comme étant un C-sous-type de `Seq(carac20)`, mais il n'y a aucun intérêt à cela dans cette application.

Ces deux nouveaux C-types pour la représentation de séquences sont baptisés `Seq(ACGT)` et `Seq(ACGU)`, ils sont définis comme des C-sous-types de `Seq(Caractère)`.

Sont ensuite définies des opérations sur chacun de ces deux C-types, qui correspondent à de simples réécritures syntaxiques, mais qui seront, comme nous le verrons, utilisées dans TROPES pour représenter les différentes étapes du processus de détermination de la protéine codée par un gène, à partir de la séquence génomique caractérisant ce gène. Les fonctions définies sont données ci-dessous.

Nous notons, pour plus de clarté :

- $\delta t_{3ACGT} = \langle Seq(ACGT); \langle Caractere; [A; A] + [C; C] + [G; G] + [T; T] \rangle; all; nothing; [3; 3] \rangle$,
- $\delta t_{3ACGU} = \langle Seq(ACGU); \langle Caractere; [A; A] + [C; C] + [G; G] + [U; U]; all; nothing; [3; 3] \rangle$,
- $\delta t_{20car} = \langle Caractere; [A; A] + [C; C] + \cdots + [Y; Y] \rangle$

nous pouvons alors définir :

- `mt-extract-maxseq(Seq(ACGT),⌈Seq(ACGT)⌉,` $\delta t_{3ACGT}; \delta t_{3ACGT} \rangle$`)` $\mapsto$ `⌈Seq(ACGT)⌉` ; si s_1 est la séquence en premier paramètre, s_2 est la première séquence de cardinalité trois (le second paramètre), et s_3 est la seconde séquence de cardinalité trois (le second paramètre de ce type), alors cette fonction renvoie la plus longue sous-séquence de s_1 comprise entre s_2 et s_3.

- `mt-traduct(Seq(ACGT),`$\lceil$`Seq(ACGT)`$\rceil$`)` $\mapsto$ $\lceil$`Seq(ACGU)`$\rceil$; soit s la séquence en paramètre, cette fonction renvoie la séquence qui fait correspondre, dans s, à 'A' un 'U', à 'T' un 'A', à 'G' un 'C' et à 'C' un 'G'.

- `mt-associate-seq3(Seq(ACGU),`δt_{3ACGU}`)` $\mapsto$ δt_{20car}, qui à une séquence de trois caractères parmi 'A', 'C', 'G' et 'U', fait correspondre un caractère parmi 'A', 'C', ..., 'Y'. Cette fonction est réalisée par consultation d'une table d'associations.

- `mt-transcript(Seq(ACGU),` $\lceil$`Seq(ACGU)`$\rceil$`)` $\mapsto$ $\lceil$`Seq(Carac20)`$\rceil$; cette fonction est l'application de `mt-associate` à tous les triplets, pris dans l'ordre, d'une séquence de caractères pris parmi 'A', 'C', 'G', et 'U', pour obtenir une séquence correspondant au codage d'une protéine.

Les fonctions indiquées ci-dessus vont nous permettre de retracer les différentes étapes du processus que nous considérons, en les utilisant comme méthodes de calcul de valeurs d'attributs de classes d'objets biologiques : nous constatons ici que, bien que ces calculs soient effectués sur des structures de données externes à TROPES, leur interprétation dans le monde modélisé demeure possible. C'est une première illsutration du fait que la structure de `Séquence` n'est pas à définir en termes du langage de représentation de TROPES, puisqu'elle n'est utilisée qu'à des fins de calculs dont les *résultats* sont interprétables dans le monde modélisé, par association avec des attributs significatifs. Nous précisons dans la suite comment ces fonctions, définies dans METÉO, peuvent effectivement être associées aux attributs des objets significatifs de TROPES.

6.3.3 Exploitation de ces nouveaux C-types dans la modélisation

Nous considérons maintenant le concept des `Objets-biologiques-simples` qui ne comporte qu'un seul point de vue.

Nous y définissons les classes `Gène-protéique` et `Protéine` qui contiennent, *entre autres*, les informations relatives à la relation qui les lie, et qui va être représentée, à l'aide d'attributs, par la description de ces attributs qui correspondent chacun à une étape du processus de détermination de la protéine que code un gène, à partir de la séquence d'ADN portant ce gène ; ce processus va être représenté à l'aide de méthodes attachées aux attributs, par utilisation du mécanisme de détachement procédural.

Nous supposons que la classe `Gène` est déjà définie, dont hérite la classe `Gène-protéique` que nous allons définir. En ce sens, nous n'allons pas représenter toutes les caractéristiques d'un gène au sein de `Gène-protéique`.

Nous décrirons ensuite une sous-partie des propriétés caractéristiques d'une protéine, afin de compléter la représentation de la relation qui lie `Protéine` à `Gène-protéique`.

La classe `Gène-protéique`

Un gène protéique est avant tout un gène, qui a la propriété de coder une protéine. Nous décrivons ci-dessous, à l'aide des fonctions précédemment définies sur les séquences de caractères, les différentes étapes du processus de détermination de la protéine associée à un gène protéique, ces différentes étapes étant associées à des attributs. Ces étapes sont celles que représentent les attributs `seqADN`, `SeqCod`, `SeqProt` et `Protéine`.

```
Classe Gène-protéique sorte-de Gène
    Attributs :
        promoteur $seq-de ACGT
                  $card [2;6]
        Type-gene $un Chaîne
                  $domaine ("CDS" "ORF" "URF")
                  $defaut "CDS"
        seqADN $seq-de ACGT
        SeqCod $seq-de ACGT
               $sib-exec mt-extract-maxseq(Séquence(ACGT),
                                           *.seqADN,
                                           ('T' 'A' 'C'),
                                           ('T' 'T' 'G'))
        SeqProt $seq-de Carac20
                $sib-exec mt-transcript(Séquence(Carac20),
                                   mt-traduct(Seq(ACGU),*.seqCod))
        Protéine $un Protéine
                 $sib-filtre { P Protéine, P.nom :
                               P.SeqAA = *.SeqProt }
        Fonction $un Chaîne
                 $domaine ("catalytique" "transport" "stockage" ..."inhibiteur")
        RBS $un RBS
```

Où `RBS`[7] est une classe du même concept qui décrit les caractéristiques des sites de fixation des ribosomes.

On vérifie en particulier que tous les attributs de la classe `gène-protéique` ont une signification au regard de la signification de cette classe dans le monde modélisé. De ce fait, toutes les fonctions associées aux attributs voient leurs résultats effectivement interprétés dans le monde modélisé, alors que la sémantique opérationnelle de ces fonctions n'ont aucun sens dans ce monde.

La classe `Protéine`

La classe protéine se décrit, en partie, comme suit :

```
Classe Protéine sorte-de Objet-Biologique
    Attributs :
        SeqAA $seq-de Carac20
        nom $un Chaîne
```

Bien entendu, la classe `Protéine` contient de nombreuses autres caractéristiques, qui ne sont pas pertinentes pour notre exemple.

[7] *Ribosome Binding Site*

Remarque

Il pourrait être concevable de considérer les fonctions de manipulation de séquences attachées aux attributs comme des *méthodes* (au sens des langages de programmation par objets) de la classe `gène-protéique`, dans la mesure où elles concernent des objets de cette classe.

Pourtant, ces fonctions de calculs ne modélisent un *comportement* ni des *gènes*, ni des *protéines*. La sémantique opérationnelle que nous pourrions leur associer ferait fi du monde modélisé ; elles sont l'implémentation du codage des phénomènes de traduction et de transcription, qui eux mêmes sont des étapes du processus de synthèse des protéines. Elles n'ont en ce sens pas à faire partie de la description des `gènes` en particulier, puisqu'impliquent d'autres considérations biologiques : la représentation que nous en avons faite ici, par représentation des étapes du processus, a l'avantage de cerner le rôle pris par chaque objet biologique lors de ce phénomène biologique, de façon déclarative donc interprétable dénotationnellement.

6.3.4 Conclusion

Nous avons montré, dans cet exemple, que MeTéo permet le développement de structures de données et d'opérations sur ces structures, qui servent de support à la *programmation* des étapes d'un calcul correspondant à la modélisation mathématique, donc abstraite et calculatoire, d'un processus biologique qui n'est d'ailleurs pas encore, de nos jours, complètement défini.

Nous retiendrons de cet exemple que le processus ainsi représenté n'est pas maître de l'évolution des objets biologiques modélisés au sein de la base Tropes. Autrement dit, l'activation des inférences n'a pas comme effet de bord la création intempestive d'objets biologiques qui pourraient correspondre à une séquence calculée. Nous considérons que ceci est essentiel, dans la mesure où il s'agit d'un processus opérant sur des structures dénuées, a priori, de signification dans l'univers modélisé. C'est, en particulier, le fait de rattacher les méthodes de calcul à des objets biologiques, qui confère à ces méthodes une signification, et non pas le contraire.

Notons, enfin, que les méthodes qui ont été définies dans les nouveaux C-types peuvent aussi être utilisées par un système de résolution de problèmes qui exploiterait une base de connaissances modélisant des objets biologiques, comme par exemple un système de *tâches* [Wil94]. Un système de tâches consiste à représenter un enchaînement de tâches élémentaires (opérations spécifiques, dont on contrôle les entrées et les sorties, ce qui permet, en aval, le contrôle de l'enchaînement de ces opérations, par la représentation explicite des flux de données), qui correspond à une stratégie de résolution d'un problème général donné. Les tâches sont appliquées à des objets de représentation, et sont définies à partir d'opérations sur ces objets ou sur leurs descriptions.

6.4 Classe de représentation et type abstrait de données

Une classe de représentation décrit les objets du monde modélisé qu'elle cherche à regrouper. Cette description est représentée au sein d'une structure informatique qui est exploitable, en tant que structure, ou en tant que représentation des objets de la classe. Cette section est réservée à illustrer les deux niveaux d'exploitation de la structure d'agrégat d'une classe.

Tout comme il y a deux niveaux dans l'interprétation de la description d'une classe, il y a deux niveaux d'exploitation de cette description : il y a un premier niveau d'exploitation qui consiste à *représenter le comportement des objets du monde modélisé ainsi décrits par la classe*, et il y a un second niveau de manipulation de cette description qui consiste à *définir le comportement des données structurées d'après la structure d'agrégation de cette classe.* Par exemple, on considère la classe `Vin`, une opération du premier niveau pourrait être la représentation du `vieillissement d'un vin`; une opération du second niveau consisterait à imprimer les caractéristiques de ce vin. Si la première de ces opérations est effectivement la modélisation du comportement de tout vin, la seconde n'est que l'exploitation de sa structure.

Ainsi, de la même façon que nous distinguons les deux interprétations de la description d'une classe, en intension et en extension, nous proposons, par le biais de MetÉo, un support pour l'expression de cette différence au niveau de la spécification du comportement d'une classe, en autorisant l'interprétation d'une classe de représentation comme un type abstrait de données à part entière, c'est-à-dire définissant la structure de ces données et les opérations de manipulation de ces données. Cette interprétation est effectuée en associant à une classe un C-type de MetÉo.

6.4.1 C-typage d'une classe

Le C-typage d'une classe Tropes représente l'interprétation de cette classe comme un type abstrait de données[8].

Création minimale du C-type

Le C-type d'une classe n'est autre qu'un C-sous-type de `Record`, dont le prédicat d'appartenance est directement défini à partir du δ-type, issu de `Record`, qui correspond à cette classe. Soit $T(C)$ le C-type correspondant à la classe de représentation C, soit $t(C)$ le δ-type issu du typage classique de C. On définit alors :

$$\in_{T(C)}(v) \Longleftrightarrow v \in^{\delta}_{Record} t$$

Quant aux autres prédicats et fonctions du C-type `Record`, notamment tous ceux qui correspondent au mode de représentation des δ-types, ils sont récupérés par $T(C)$ selon le même principe que l'héritage.

Notons que si le C-type de la classe C existe dans MetÉo, son δ-type est conservé au sein de MetÉo, et c'est sur ce δ-type que MetÉo procèdera aux diverses vérifications de l'interprétation intensionnelle de la base de connaissances. Ceci permet de ne pas compléter inutilement tous les algorithmes de parcours de treillis des δ-types, du fait qu'il existerait alors deux expressions Eole dénotant le même ensemble de valeurs, ce qui est incompatible avec l'hypothèse d'expressions normalisées de δ-types. Pour résumer, malgré la création de $T(C)$, on aura toujours `get-type(`C`)`$=$ t.

Théoriquement, le type d'une classe, en tant qu'entité de représentation, doit être un δ-type. Le C-typage d'une classe est la représentation de l'ensemble des valeurs records qui appartiennent au δ-type record correspondant à cette classe. Or pour un C-type T, le δ-type $\lceil T \rceil$ dénote le même ensemble de valeurs que celui dénoté par T, c'est-à-dire :

$$\|T\|^{\xi} = \|\lceil T \rceil\|^{\xi}$$

or d'après la définition du prédicat d'appartenance d'une valeur à $T(C)$, on peut déduire :

$$\|T(C)\|^{\xi} = \|t(C)\|^{\xi} \quad \text{donc } \|t(C)\|^{\xi} = \|\lceil T(C) \rceil\|^{\xi}$$

[8] De la même façon, nous pouvons considérer le C-typage d'un concept

En conséquence, le C-typage d'une classe reste l'interprétation intensionnelle de cette classe, la différence d'avec un typage classique (un δ-typage), c'est que l'on peut associer un comportement à la description de la classe, indépendamment de la signification de cette classe dans le monde modélisé.

Ce C-type n'est pas créé dans l'optique de la représentation de δ-types, et pour cette raison, MÉTÉO ne l'exporte pas dans TROPES comme un C-type pouvant typer des attributs (il ne pourra être associé à la facette $un). Cela a pour conséquence le fait qu'aucune création de δ-type issu de $T(C)$ ne pourra être faite dans MÉTÉO, exceptés les δ-types $\lceil T(C) \rceil$ et $\lfloor T(C) \rfloor$ qui le sont automatiquement à la création de $T(C)$.

L'ajout de $T(C)$ dans MÉTÉO est possible du fait de la propriété d'extensibilité, et de surcroît, cet ajout peut être dynamique. En revanche, les opérations à définir sur les valeurs de $T(C)$ nécessiteront une compilation afin d'être activables.

Définition des opérations sur les valeurs du C-type

Une fois ce C-type créé, le type record sous-jacent à la description de la classe de représentation est disponible, et accessible à des fins de manipulation, qui sont à spécifier et à définir de la même façon que peut l'être toute opération sur un C-type, en termes du langage hôte, à partir d'autres opérations des C-types de MÉTÉO, et selon une signature formatée.

Dans ce cadre, on peut par exemple imaginer la création du C-type `M-Note`, issu du C-typage de la classe de représentation `Note-Musique` contenant la description d'une note de musique : sa fréquence, sa durée, sa tonalité, la donnée de ses notes associées pour la définition d'un accord majeur... Au C-type `Note` on peut alors exploiter les données de cette description pour définir l'opération d'impression de cette note selon un schéma de partition établi, et en fonction de la clé de cette partition. Cette opération représente bel et bien une exploitation de la structure de la classe `Note-Musique`, contrairement à une opération modélisant l'atténuation d'une note.

Exportation des opérations du C-type

Les opérations associées à $T(C)$ sont exportées dans TROPES, et pourront être activées selon le même principe que toute autre opération sur des valeurs de C-type.

Par ailleurs, ces opérations pourront être exportées au niveau de systèmes exploitant, dans un objectif précis, les connaissances représentées dans TROPES, tel qu'un système de résolution de tâches.

6.4.2 Classe de représentation et classe de programmation

Le C-typage d'une classe est la spécification d'un comportement associé à la *description* de cette classe, et réalisé par un C-type. On se rappelle que les classes considérées par les langages de programmation par objets sont elles-mêmes des implémentations de types abstraits. Cette constatation, représentée naïvement sur la figure 6.3, permet d'établir un lien entre les classes pour la représentation, et les classes pour la programmation.

On constate en effet qu'un C-type, puisqu'il est l'implémentation d'un type abstrait, peut être assimilé à une classe de programmation par objets. D'ailleurs, l'implémentation de MÉTÉO,

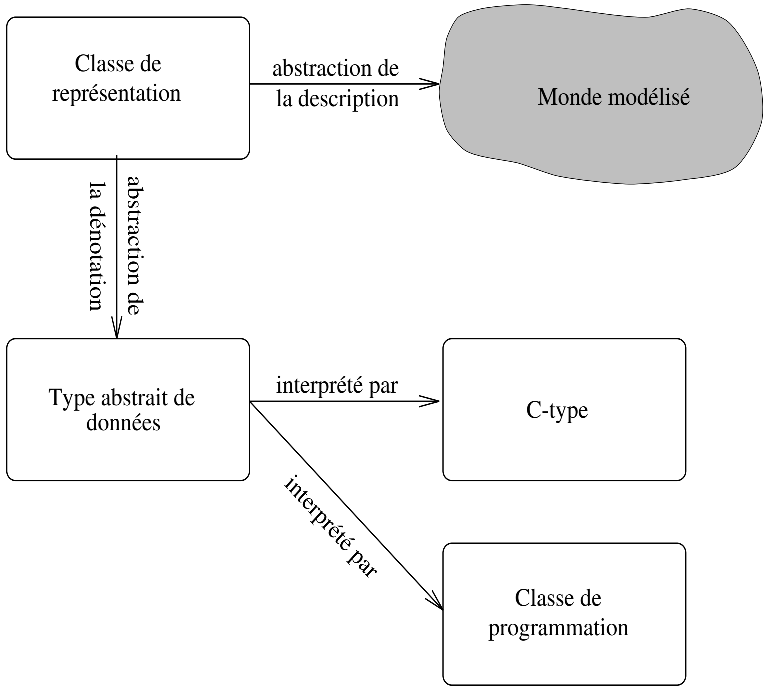


FIG. 6.3 - : Les différents niveaux d'abstraction des classes pour la représentation des connaissances et des classes pour la programmation

commentée dans l'annexe D, illustre cette correspondance, puisque les C-types y sont programmés par des classes d'un langage à objets.

Lorsqu'un C-type est issu du C-typage d'une classe de représentation, il est donc une implémentation, donc une interprétation, de l'interprétation intensionnelle de cette classe, à savoir une interprétation dans un univers de valeurs particulier d'une abstraction de cette classe ayant pour effet d'"oublier" le fait que cette classe dénote un groupe dans le monde modélisé. Un C-type pouvant être apparenté à une classe en programmation, il est tentant d'affirmer qu'*une classe d'objets en programmation peut être vue comme l'implémentation de la description d'une classe de représentation, indépendamment de ce qui lie cette description à un groupe d'individus du monde modélisé.*

Cette correspondance ne nous permet certes pas d'affirmer que les classes de programmation n'offrent pas un bon cadre pour la représentation des connaissances, mais, et compte-tenu de la sémantique des langages de programmation par objets pour lesquels l'existence d'un objet est tributaire de l'appartenance figée de cet objet à une classe donnée, nous sommes enclins à préférer une *coopération* de la programmation et de la représentation des connaissances par objets. Une telle coopération reflète les différents niveaux d'interprétation d'un monde à modéliser.

6.5 Conclusion

Nous avons montré, dans ce chapitre, que la propriété d'extensibilité de METÉO permet de définir des structures de données utilisables dans des bases de connaissances TROPES, sans pour autant modifier la sémantique du langage de représentation, puisque ces structures de données sont définies au sein de METÉO, et donc interprétées par METÉO, indépendamment du monde modélisé par la base de connaissances. Tout nouveau C-type peut ainsi être manipulé par une application, de la même façon que le sont des types plus primitifs, tels que `Entier` ou `Booléen`.

En outre, nous avons montré qu'une classe de représentation (et plus généralement un concept), peut voir son intension représentée par un C-type, et peut donc se voir associer des opérations de manipulation dénuées de signification dans le monde modélisé. Cela illustre deux niveaux d'exploitation d'une classe, à savoir la modélisation du comportement des objets regroupés par la classe, à dénotation dans le monde modélisé, et la spécification du comportement associé à la structure *record* extraite de cette classe.

Cette distinction nous a permis, au regard de la sémantique des classes en programmation par objets, d'établir une correspondance entre la notion de classe en programmation, et celle de classe en représentation. Nous avons émis l'hypothèse que ces deux notions peuvent coopérer dans le but d'illustrer les différents niveaux d'interprétation d'un monde à modéliser.

L'étude des différences entre les domaines de la représentation des connaissances et de la programmation, est le thème de nombreux groupes de travail (Workshop de l'IJCAI en 1993, workshop de l'ECAI en 1994) et autres séminaires [CDE+95], voire le thème central de numéros spéciaux de revues (TSI). Les divergences au sein des deux communautés sont nombreuses, et nous pensons avoir, de par ce travail, apporté un nouvel élément de réponse, si ce n'est un nouveau point de départ pour les discussions à venir, en orientant la discussion vers une tentative de coopération de tels systèmes plutôt qu'une réelle intégration dans l'un des composantes de l'autre.

Conclusion

Nous avons abordé, dans ce travail, la délicate distinction entre les interprétations intensionnelle et extensionnelle d'une base de connaissances. La première concerne l'attribution, à chaque entité de représentation, d'une signification dans le monde modélisé. Elle trouve donc son importance dans la validation de la pertinence d'éléments représentés, au regard du domaine considéré : tel objet dénote-t-il réellement une protéine ? La seconde de ces interprétations s'attache à interpréter les descriptions des entités, exprimées en termes du langage de représentation. Ces deux interprétations ont été introduites par les langages terminologiques [BFL83] [Neb90] [DLNS94], qui ont rapidement fait l'hypothèse de leur équivalence afin de rendre significatifs, au regard du monde modélisé, les processus de ces systèmes activés sur les descriptions des entités.

Généralement, dans les systèmes de représentation, les co-domaines de ces deux fonctions d'interprétation sont confondus, ce qui se traduit par le plongement d'un espace de valeurs dans le monde modélisé, bien qu'une valeur puisse avoir de multiples interprétations dans ce monde modélisé. Après une analyse informelle des interprétations intensionnelle et extensionnelle au sein du modèle de connaissances à objets TROPES (chapitre 2), nous avons élaboré un système de types, appelé METÉO, dans le but d'expliciter la différence entre les deux co-domaines des fonctions d'interprétations. La réelle matérialisation de cette distinction avait pour finalité la clarification de la coopération, au sein du modèle de représentation des connaissances, entre interprétations intensionnelle et extensionnelle. La mise en évidence de cette coopération est, par ailleurs, liée à la distinction qu'il y a lieu de faire entre la représentation de données structurées dénuées de signification dans le monde modélisé, et la représentation d'individus interprétables dans ce monde : les premières se voient attribuer une signification à partir du moment où elles sont associées aux modélisations d'objets réels, d'où la présence de deux niveaux d'interprétation qu'il s'agit de distinguer.

Le système de types METÉO, que nous avons présenté dans le chapitre 3, est la matérialisation d'un espace de valeurs ouvert à la spécification de structures de données même complexes. Son développement, et son couplage à TROPES, ont permis l'atteinte de nos objectifs, du fait de son organisation hiérarchique à deux niveaux.

Metéo pour le développement de structures de données

Les *C-types* de METÉO sont l'implémentation de types abstraits de données, tels que `Entier`, `Liste(X)` ou encore `Record`, qui sont ordonnés par la relation de C-sous-typage correspondant à l'inclusion ensembliste. Les C-types sont exportés dans TROPES et les opérations définies sur leurs valeurs sont activables à partir de TROPES.

La hiérarchie des C-types est extensible, grâce au polymorphisme de METÉO et, de ce fait, le concepteur d'une base de connaissances a la possibilité de définir, dans METÉO, des structures de données qui peuvent être requises par son application (chapitre 6). Une conséquence immédiate de cette fonctionnalité de METÉO est que le langage de représentation des connaissances n'a pas à adapter sa syntaxe et sa sémantique pour supporter la définition de structures de données à vocation calculatoire, qui n'ont, de surcroît, pas de correspondance directe dans le monde modélisé. D'ailleurs, le fait que ces structures puissent être développées dans METÉO tout en restant disponibles dans TROPES au même titre que des types de base, illustre bien le fait que l'interprétation extensionnelle de ces structures n'est pas pertinente.

METÉO contribue ainsi à cette mise en évidence et permet de rétablir la finalité du langage de représentation qui est d'accueillir la description de modélisation d'un domaine d'application, sans pénaliser le concepteur puisqu'il a la possibilité de définir des structures de données complexes et de les utiliser au sein de son application. Nous rejoignons ici les objectifs du module SPIDER intégré au système WEAVE, mis au point par Mark Graves, qui sont d'offrir un support à la définition de structures de données requises pour l'expression de modélisations en biologie moléculaire [Gra93].

Metéo pour la représentation normalisée des intensions

METÉO réalise le typage des entités de représentation. Le type d'une entité est une expression normalisée, appelée δ-type, dénotant l'ensemble des *valeurs* qui vérifient l'intension de cette entité. En particulier, le type d'une classe est un type *record*. Les δ-types issus d'un même C-type sont ordonnés par le δ-sous-typage, et METÉO permet la comparaison de δ-types issus de C-types différents grâce au γ-sous-typage. Le sous-typage traduit l'inclusion ensembliste sur un ensemble structuré de valeurs. Ainsi, le sous-typage a été élaboré pour correspondre à la spécialisation entre descriptions de classes, tandis que l'appartenance d'une valeur à un type correspond au rattachement d'une instance à une classe ou à un concept.

Nous avons montré (chapitre 4), en l'absence de contraintes dynamiques sur les objets, l'isomorphisme entre l'algèbre issue de l'interprétation intensionnelle d'un base de connaissances et celle issue de METÉO. Par conséquent, toute opération de manipulation des descriptions d'entités trouve une opération équivalente dans METÉO, qu'il s'agisse de la spécialisation, de l'instanciation, de la relation entre passerelles ou d'opérations visant à simuler l'héritage multiple.

De ce fait, toutes les opérations de TROPES, qui auparavant manipulaient les descriptions des classes et objets de représentation, sont maintenant confiées à METÉO, et leurs résultats sont ensuite explicitement interprétés par TROPES selon un schéma général. Pour cela, il a fallu établir clairement dans quelles conditions et à quels moments se faisaient les sollicitations à METÉO, c'est-à-dire quand et comment les intensions sont considérées dans les mécanismes d'exploitation et d'inférence du modèle de connaissances. Le chapitre 5 s'y est attaché, et a permis la mise en évidence d'une réelle coopération entre les interprétations en intension et en extension. Nous retiendrons en particulier le fait que les opérations activées sur les intensions ne sont pas garantes de la rapidité et de la complétude des mécanismes d'exploitation et d'inférence, et doivent souvent faire appel à des opérations de manipulation des extensions.

En outre, nous avons vu, dans le chapitre 6, qu'il est possible d'associer un C-type à une classe de représentation, et en ce sens, les opérations de ce C-type traduisent la manipulation de la structure de description de la classe ; en aucun cas elles ne peuvent être assimilées à la modélisation du comportement des objets réels dénotés par cette même classe.

Coopération langage de programmation / langage de représentation

Le chapitre 6 a montré à quel niveau peut se situer une coopération entre les compétences respectives des langages de programmation par objets et des systèmes de représentation des connaissances. En effet, nous avons proposé un schéma de coopération dans lequel le langage de programmation par objet est dédié à l'implémentation des spécifications abstraites des C-types, dans la mesure où les classes des langages de programmation sont définies comme étant la réalisation d'un type abstrait de données.

En particulier, nous avons montré que dans un tel schéma, et compte-tenu de leurs objectifs respectifs, un système de représentation des connaissances et un langage de programmation peuvent évoluer à des niveaux d'interprétations distincts mais coopératifs : pour prendre un exemple, disons que le système de représentation a pour rôle d'associer le type abstrait `Matrice` à des représentations d'objets réels, tandis que le langage de programmation s'attache à donner une implémentation efficace de ce type abstrait.

Mise en œuvre opérationnelle de Metéo

Metéo a été implémenté pour être faiblement couplé au modèle de connaissances à objets Tropes. Un prototype de ces deux systèmes a été réalisé en Le-Lisp version 16, dialecte de Lisp et comprenant une couche objets, développé par la société Ilog (Gentilly, France). Ce prototype a permis de valider la contribution de Metéo au niveau des différentes composantes de représentation et d'exploitation de Tropes. L'annexe D présente un bref descriptif de cette implémentation.

Une version stable de Tropes et de Metéo a été réalisée par Jérôme Euzenat, au dessus du langage Talk de Ilog, version disponible à l'URL `http://everest.imag.fr`.

Perspectives

Les perspectives de ce travail se situent à deux niveaux, le premier concerne l'enrichissement et la consolidation de Metéo, le second est relatif à une exploitation de la coopération intension/extension qui pourrait s'étendre à d'autres relations entre objets et à d'autres mécanismes d'exploitation.

Perspectives concernant Metéo

L'organisation du système de types Metéo présente l'avantage d'être ouverte vers des améliorations et enrichissements divers. Nous avons fait état de ces perspectives dans la section 3.7, et nous retiendrons deux d'entre elles qu'il serait intéressant d'étudier dans un avenir proche.

– Développer une sémantique adaptée à la résolution des cycles dans les expressions. Les études théoriques sur ce thème sont nombreuses, et les composantes propres à Metéo (sous-typage, opérations de INF et SUP) offrent un cadre adapté à l'opérationnalisation de ces théories. Nous pensons plus particulièrement aux travaux de Bernhard Nebel [Neb91], de Roberto Amadio et Luca Cardelli [AC91], ainsi qu'à ceux d'Hassan Aït-Kaci [AK86], qui sont tous dédiés à la résolution de ce problème en présence d'ordres sur les expressions.

– Élaborer un langage de spécifications de types abstraits de données, inspiré de OBJ2 ou Larch, qui intègre en sus la spécification du mode de représentation et de manipulation des δ-types.

En outre, il nous paraît particulièrement essentiel de bénéficier des nombreux travaux existants quant au couplage types/contraintes, pour améliorer la prise en compte, au sein des structures de METÉO, des contraintes exprimées sous MICRO (le module de gestion de contraintes couplé à TROPES [Gen95]) et portant sur les entités de représentation. Faire évoluer le langage d'expressions de δ-types EOLE pour y permettre l'expression statique de contraintes inter-attributs devrait pouvoir s'effectuer en s'inspirant des langages LIFE [AKP91] ou TM [BdBZ93] qui autorisent l'expression de types construits et étiquetés, avec la possibilité de contraintes entre étiquettes.

Perspectives relatives à l'interface Tropes/Metéo

Le développement de METÉO pour TROPES a permis de mettre en évidence les liens conceptuels entre les descriptions des entités et leur dénotation. En ce sens, il est possible d'étendre les fonctionnalités du couplage de METÉO avec un modèle de connaissances. D'ailleurs, rappelons ici que METÉO est déjà un support pour un algorithme de construction de concepts à partir d'ensembles d'instances [VE95], de par son organisation hiérarchique qui a la propriété d'être un système classificatoire [Euz94].

Dans un premier temps, la résolution des expressions cycliques de types dans METÉO autorisera les définitions d'objets cycliques dans TROPES. Cette possibilité, encore trop peu répandue dans les systèmes de représentation actuels, est pourtant une composante essentielle d'un langage dit “expressif”.

Dans un second temps, nous envisageons de paramétrer la relation de sous-typage[9] selon les différentes utilisations qui en sont faites. Plusieurs δ-sous-typages entre types *records* sont ainsi réalisables, comme celui qui ne considèrerait qu'un sous-ensemble des étiquettes d'un tel type (pour établir une correspondance avec la notion d'attributs contingents), ou encore le sous-typage paramétré par une mesure de similarité, de telle façon que les opérations sur ces mesures soient définies au sein même de METÉO quand elles concernent les descriptions d'entités de représentation.

Il est aussi envisagé de prendre en compte, au niveau des types des entités, la notion d'objets composites, qui nécessite la mise en œuvre d'une procédure particulière de typage des attributs composites, qui s'inpirerait du typage des attributs complexes.

Enfin, nous pensons, à plus long terme, mettre à contribution des algorithmes de comparaison de treillis de types d'attributs, dans l'optique d'un processus de comparaison structurelle, voire de fusion, de bases de connaissances modélisant un même monde, mais développées par différents concepteurs [LNE89]. En effet, à l'heure où l'on assiste à l'avènement du partage et de l'échange d'informations, une telle fonctionnalité au sein d'un système de représentation s'avère particulièrement intéressante.

[9] Et de la même façon, la relation d'appartenance d'une valeur à un δ-type.

Annexe A

Normalisation des δ-types

Principes de la normalisation

La fonction de normalisation, notée $NormalForm_T(\delta t)$, calcule la forme normalisée du δ-type δt, issu du C-type T. Avant cette phase de normalisation, les champs d'expression du δ-type concerné sont renseignés, et les informations contenues dans ces champs sont cohérentes et sans redondance, lorsque chaque champ est observé indépendamment des autres. Le but de la normalisation est, en tout premier lieu, d'obtenir cette cohérence et cette absence de redondance lorsque tous les champs sont considérés ensemble. Ainsi, le théorème suivant, concernant la normalisation, peut être respecté :

Théorème 5 *Correction, complétude et unicité des formes normales*

$$\forall t \text{ un } \delta\text{-type}, \exists \text{ un et un seul } \delta t^N = NormalForm(\delta t) \text{ tel que } \|\delta t\|^{\xi} = \|\delta t^N\|^{\xi}$$

Nous présentons dans les sections qui suivent les étapes de la normalisation, tout d'abord dans le cas des δ-types issus de C-types monovalués (simples), et ensuite dans le cas de δ-types construits. La phase de normalisation est, quel que soit le mode de représentation considéré, toujours constituée de deux phases qui peuvent être vues comme successives :

1. élimination des redondances et incorrections suite à l'observation simultanée des champs de représentation,
2. détermination de la forme normale, c'est-à-dire réécriture des contenus des champs.

À l'issue de ces deux phases, nous prouverons, dans le cas de chaque C-type, que le théorème 5 est respecté, tout au moins en théorie.

Normalisation des types de base

Les C-types simples qui existent actuellement dans METÉO sont exclusivement les `Ordonnés` et les `Énumérés`. La normalisation dans ces deux cas est relativement triviale, du fait de la simplicité de la syntaxe d'expression des δ-types.

Normalisation des `Ordonnés`

Rappelons que dans le cas des ordonnés, les ensembles de valeurs dénotés par les δ-types sont exprimés par des listes d'intervalles. L'affinement du C-type `Ordonnés` vers les types `Continus` et `Discrets` ne dépend que d'une différence entre les propriétés de ces intervalles :

- les δ-types issus de `Continus` (de ses C-sous-types) sont exprimés à l'aide de listes d'intervalles *à bornes ouvertes*,
- les δ-types issus de `Discrets` (de ses C-sous-types) sont exprimés à l'aide de listes d'intervalles *à bornes fermées*,

En tout état de cause, et ce quelles que soient les propriétés des intervalles, les δ-types issus de C-sous-types de `Ordonnés` sont exprimés à l'aide d'un seul champ de représentation, ce qui signifie que la seconde phase de normalisation est inutile. En effet, `Ordonnés` concerne les C-types totalement ordonnés ; il n'existe donc qu'une seule façon d'exprimer des sous-ensembles de ces types à l'aide de listes d'intervalles cohérentes et non redondantes, à condition d'ordonner les intervalles d'une liste.

Or les fonctions GR_T ($Ground-Range$) CR_T ($Compute-Range$), ainsi que $inter_T$ (intersection de deux intervalles), $union_T$ (union de deux intervalles), $appval_T$ (appartenance d'une valeur à un intervalle) et $remove_T$ (suppression d'un élément dans un intervalle), sont définies pour le C-type `Discrets` d'une part, et pour le C-type `Continus` d'autre part, et ce à partir des propriétés des intervalles. Elles assurent la correction et la cohérence des listes d'intervalles produites lors du typage des attributs, et plus généralement lors de toute initialisation d'un δ-type issu de `Ordonnés`. En particulier, ces fonctions intègrent les propriétés des bornes (chez les discrets, les opérations $succ_T$ et $pred_T$ sont utilisées), ainsi que les propriétés à l'infini (ou plus exactement au niveau des bornes inférieure et supérieure). Le détail de ces fonctions n'est pas ici très utile, puisqu'elles relèvent d'une arithmétique simple des intervalles.

Ainsi, la formation même de ces listes d'intervalles assure leur correction, leur cohérence, et garantit l'absence de redondance. Par exemple, il n'est pas possible d'obtenir la liste d'intervalles d'entiers ($[4;10]$ $[9;12]$) qui sera réécrit en ($[4;12]$). De ce fait, la normalisation ne concerne plus que la phase d'obtention de cohérence, correction et absence de redondance dans l'expression des δ-types, lorsque l'on considère simultanément tous les champs d'expression de ces δ-types. Étant donné qu'il n'existe, dans le cas des ordonnés, qu'un champ d'expression (**domaine**), cette phase est, elle aussi, superflue.

En conclusion, la normalisation des δ-types issus de `Ordonnés` et de ses C-sous-types n'a pas lieu d'être, puisque le seul mode d'expression de ces δ-types passe par des listes d'intervalles, pour lesquelles la forme normale, correcte et complète est assurée par construction, lors de la création de δ-types.

Normalisation des `Énumérés`

Les δ-types issus de `Énumérés` sont quant à eux représentés à l'aide de deux champs : **domaine** et **comp-dom**. Ces deux champs sont renseignés sous formes d'*énumérations explicites de valeurs*, ou par le symbole TOUT qui indique une référence à toutes les valeurs vérifiant le prédicat d'égalité du C-type.

La construction de ces deux champs, lors de la création d'un δ-type, assure la cohérence et la non-redondance individuelles des contenus de chacun d'eux ; autrement dit, le champ **domaine** (de même le champ **comp-dom**) ne contient pas de doublet, et toutes les valeurs qui y sont appartiennent au C-type.

Par contre, la construction d'un δ-type ne garantit pas sa non-redondance et sa cohérence lorsque les deux champs sont considérés simultanément. Par exemple, certaines valeurs contenues dans **domaine** peuvent apparaître dans **comp-dom**, ou encore certaines valeurs de **comp-dom** n'apparaissent pas dans **domaine** et sont donc inutiles. La seconde phase de la normalisation a pour but d'obtenir la non-redondance et la cohérence des δ-types lors de la prise en compte conjointe des champs d'expression. Elle est réalisée par un système de réécriture.

Rappelons la syntaxe d'un δ-type issu de `Énumérés` :

$$t = \langle Enumeres; [d_1; d_2] \rangle$$

où d_1, tout comme d_2, est soit $\emptyset$, soit le symbole TOUT, soit une énumération finie de valeurs $v_1, \cdots, v_n = v_1 + V$ où $+$ dénote la concaténation, avec :

$$\begin{array}{rcl} \|t\|^\xi & = & \|d_1\|^\xi \backslash \|d_2\|^\xi \\ \|\emptyset\|^\xi & = & \emptyset \\ \|\textsc{tout}\|^\xi & = & \|Enumeres\|^\xi \\ \|v_1, \cdots, v_n\|^\xi & = & \{v_1, \cdots, v_n\} \end{array}$$

Les quatre formes normales que nous cherchons à obtenir sont les suivantes :

1. $\langle Enumeres; [\textsc{tout}; \emptyset] \rangle$ qui dénote $\|enumeres\|^\xi$,
2. $\perp$ qui dénote $\emptyset$,
3. $\langle Enumeres; [\textsc{tout}; v'_1, \cdots, v'_p] \rangle$ qui dénote $\|enumeres\|^\xi \backslash \{v'_1, \cdots, v'_p\}$
4. $\langle Enumeres; [v_1, \cdots, v_n; \emptyset] \rangle$ qui dénote $\{v_1, \cdots, v_n\}$.

Tout sous-ensemble de l'ensemble infini des énumérés peut s'exprimer sous au plus une de ces quatre formes. La normalisation des δ-types issus de `Énumérés` a donc pour but de dériver, de n'importe quel δ-type vérifiant les préconditions (cohérence et non-redondances individuelles des champs), un δ-type s'exprimant sous l'une de ces quatre formes normales. Ainsi, les règles suivantes obéissent au principe suivant : toutes les valeurs apparaissant dans **comp-dom** (i.e. dans d_2) sont éliminées de **domaine** (i.e. d_1) lorsque ce dernier n'est pas infini (donc exprimé par TOUT).

$$\frac{t = \langle Enumeres; [v_1 + V; v_1 + V'] \rangle}{t = \langle Enumeres; [V; V'] \rangle} \quad (A.1)$$

$$\frac{t = \langle Enumeres; [v_1, \cdots, v_n; v'_1, \cdots, v'_p] \rangle, n, p \geq 1, \text{ t.q. } \forall i, v_i \neq v'_i}{t = \langle Enumeres; [v_1, \cdots, v_n; \emptyset] \rangle} \quad (A.2)$$

$$\frac{t = \langle Enumeres; [d_1; \textsc{tout}] \rangle}{t = \perp} \quad (A.3)$$

$$\frac{t = \langle Enumeres; [\emptyset; d_2] \rangle}{t = \perp} \quad (A.4)$$

Où $\perp$ indique un δ-type inconsistant (dénotant l'ensemble vide), et aboutit à une erreur de type. Nous avons supposé ici que l'ordre des énumérations n'a aucune importance, par souci de simplification des règles.

Confluence et terminaison

Il n'existe pas d'ordre d'application de ces règles. Une seule règle applicable à chaque réduction (sauf les règles A.3 et A.4 qui peuvent s'appliquer en même temps mais leur résultat est identique) assure la confluence de ce système, et du même coup l'unicité de la forme normale.

L'application de ces règles termine, quelque soit le δ-type initial vérifiant les pré-conditions : les règles A.3 et A.4 produisent un terme qui n'apparaît pas en prémisse d'une règle, de même que la règle A.2. Quant à la règle A.1, elle réduit strictement la longueur des énumérations, ce qui élimine le risque de cycle infini dans son application de cette règle. Lors d'une longueur vide, soit le type obtenu apparaît en prémisse de la règle A.4, soit le type obtenu correspond à une forme normalisée (le domaine complémentaire est vide).

Correction et complétude

La correction et la complétude de la normalisation des δ-types issus de `Énumérés` sont relativement triviales. La complétude (tout δ-type admet une forme normale) est assurée, d'une part grâce à la confluence du système, et d'autre part grâce au fait que les règles de réduction traitent tous les cas de combinaison entre **domaine** et **comp-dom** (les cas non traités par les règles sont forcément déjà sous forme normale) :

	comp-dom = TOUT	**comp-dom** = $\emptyset$	**comp-dom** = $\{v'_1, \cdots, v'_p\}$
domaine = TOUT	règle A.3	forme normale 1	forme normale 3
domaine = $\emptyset$	règles A.4 ou A.3	règle A.4	règle A.4
domaine = $\{v_1, \cdots, v_n\}$	règle A.3	forme normale 4	règles A.1 ou A.2

La correction est vérifiée en montrant que chaque règle préserve l'invariance de l'ensemble de valeurs dénoté par le δ-type. Le tableau A.1 fait office de preuve, chaque règle étant analysée selon la préservation de l'ensemble de valeurs dénoté : on vérifie qu'avant (seconde colonne) et après (dernière colonne) application de chaque règle (une par ligne), l'ensemble de valeurs dénoté par le δ-type t est le même ; pour cela, nous appliquons la fonction d'interprétation ξ définie pour `Énumérés` au début de cette section.

Règle A.1	$t = \langle Enumeres; [v_1 + V; v_1 + V'] \rangle$ $\|\|t\|\|^\xi = (\{v_1\} \cup V) \backslash (\{v_1\} \cup V') = V \backslash V'$	$t = \langle Enumeres; [V; V'] \rangle$ $\|\|t\|\|^\xi = V \backslash V'$
Règle A.2	$t = \langle Enumeres; [v_1, \cdots, v_n; v'_1, \cdots, v'_p] \rangle$ $\|\|t\|\|^\xi = \{v_1, \cdots, v_n\} \backslash \{v'_1, \cdots, v'_n\} = \{v_1, \cdots, v_n\}$	$t = \langle Enumeres; [v_1, \cdots, v_n; \emptyset] \rangle$ $\|\|t\|\|^\xi = \{v_1, \cdots, v_n\} \backslash \emptyset = \{v_1, \cdots, v_n\}$
Règle A.3	$t = \langle Enumeres; [d_1; \text{TOUT}] \rangle$ $\|\|t\|\|^\xi = \|\|d_1\|\|^\xi \backslash \|\|Enumeres\|\|^\xi = \emptyset$	$t = \perp$ $\|\|t\|\|^\xi = \emptyset$
Règle A.4	$t = \langle Enumeres; [\emptyset; d_2] \rangle$ $\|\|t\|\|^\xi = \emptyset \backslash \|\|d_2\|\|^\xi = \emptyset$	$t = \perp$ $\|\|t\|\|^\xi = \emptyset$

TAB. A.1 - : Correction de la normalisation des énumérés

Conclusion sur les `Énumérés`

La normalisation des δ-types issus de `Énumérés` respecte bien le théorème 5 (correction, complétude et unicité de la forme normale).

Normalisation des `Construits`

La normalisation des δ-types construits est plus complexe que celle des types simples dans la mesure où un des champs d'expression des δ-types contient l'ensemble des δ-types sur lesquels est construit celui à normaliser ; ces δ-types doivent aussi être normalisés, certes, mais la difficulté réelle provient de l'élimination des redondances lorsque l'on considère ce champ conjointement aux champs d'énumération des valeurs admises et interdites.

Normalisation des `Construits` n-aires

Nous considérons ici, de façon générale, la normalisation des δ-types issus du C-type `Construits`, que nous appellerons, certes par abus de langage, *construits n-aires* afin de signifier qu'il s'agit de types issus de l'application de $n-1$ constructeurs entre n types. Nous verrons que cette normalisation peut être intégralement reprise par les δ-types issus de `Record`, mais elle devra être complétée pour les δ-types issus des multivalués afin de prendre en considération le champ contenant la cardinalité.

Un δ-type issu de `Construits` est exprimé sous forme de trois champs, qui sont **type-ref**, **domaine** et **comp-dom**. Le premier de ces champs s'exprime comme l'application de $k-1$ constructeurs sur k δ-types. Les deux autres champs sont des *énumérations explicites de valeurs construites à partir de ces mêmes constructeurs.*

De même que précédemment, la construction de ces trois champs, lors de la création d'un δ-type, assure la cohérence et la non-redondance individuelles des contenus de chacun d'eux ; autrement dit, les champs **domaine** et **comp-dom** ne contiennent pas de doublet, et toutes les valeurs qui y sont ont bien la structure attendue par l'application des constructeurs définis dans **type-ref**. Prenons l'exemple du produit cartésien de deux entier. Le δ-type

$$\langle Construits; [(\langle Entier; ([0;12])\rangle \otimes \langle Entier; ([2;14])\rangle); (4 \otimes 1)(12 \otimes 12); \emptyset]\rangle$$

vérifie les préconditions (même si la valeur 1 n'appartient pas au second des δ-types de **type-ref**). Par contre, le δ-type

$$\langle Construits; [(\langle Entier; ([0;12])\rangle \otimes \langle Entier; ([2;14])\rangle); (12)(12 \otimes 9); \emptyset]\rangle$$

ne vérifie pas les préconditions (et d'ailleurs ne peut être produit lors de la création d'un δ-type, et ce grâce à la fonction $CV_{Construits}$).

Quant au champ **type-ref**, il ne refère que des δ-types normalisés. Suivant l'exemple ci-dessus, le δ-type

$$\langle Construits; [(\langle Entier; ([0;12][9;10])\rangle \otimes \langle Entier; ([2;14])\rangle); (4 \otimes 1)(12 \otimes 12); \emptyset]\rangle$$

ne peut pas être créé, du fait de la redondance d'un des δ-types référencés.

Par contre, la construction d'un δ-type issu de `Construits`, comme pour les autres C-types, ne garantit pas sa cohérence et sa non-redondance lorsque les trois champs sont considérés simultanément. C'est donc une des priorités de la phase de normalisation.

Rappelons la syntaxe d'un δ-type issu de `Construits` :

$$t = \langle Construits; [T_c; d_1; d_2]\rangle$$

où $T_c = t_1\chi_1 \cdots \chi_{k-1}t_k$, et d_1 (tout comme d_2) est soit $\emptyset$, soit le symbole TOUT, soit une énumération finie de valeurs construites $v_1, \cdots, v_n = v_1 + V$ où $+$ dénote la concaténation. Toute valeur de ces énumérations est de la forme $v = e_1\chi_1 \cdots \chi_{k-1}e_k$. Nous rappelons l'interprétation dénotationelle d'un δ-type issu de `Construits` à partir de l'information contenue dans ces champs :

$$
\begin{array}{rcl}
\|t\|^\xi & = & (\|T_c\|^\xi \cap \|d_1\|^\xi) \backslash \|d_2\|^\xi \\
\|\text{TOUT}\|^\xi & = & \|Construits\|^\xi \\
\|\emptyset\|^\xi & = & \emptyset \\
\|v_1, \cdots, v_n\|^\xi & = & \{v_1, \cdots, v_n\}
\end{array}
$$

Les quatre formes normales que nous cherchons à obtenir sont les suivantes :

1. $\langle Construits; [T_c; \text{TOUT}; \emptyset]\rangle$ qui dénote $\|T_c\|^\xi$,
2. $\langle Construits; [T_c; v_1, \cdots, v_n; \emptyset]\rangle$ qui dénote $\{v_1, \cdots, v_n\}$,
3. $\langle Construits; [T_c; \text{TOUT}; v'_1, \cdots, v'_p]\rangle$ qui dénote $\|T_c\|^\xi \backslash \{v'_1, \cdots, v'_p\}$,
4. $\bot$ qui dénote $\emptyset$.

En outre, nous chercherons d'une part à *minimiser* T_c (en supprimant, lorsque cela est possible, des valeurs qui lui appartiennent et qui n'appartiennent pas au domaine ou qui apparaissent dans la liste des valeurs interdites), et d'autre part, à énumérer le domaine lorsqu'il est fini[1]. Ainsi, les formes normales de type 1 et 3 sont réservées à la représentation d'ensembles infinis de valeurs, alors que la forme normale de type 2 a pour but de représenter des ensembles finis. Entre autres, lorsque l'ensemble dénoté par T_c est connu pour être fini, alors il sera automatiquement énuméré lors de la normalisation. Les règles suivantes obéissent à ces priorités. Elles peuvent être partitionnées en quatre groupes ayant des objectifs distincts :

- Les deux premières ont pour but d'éliminer des champs d'énumération, les valeurs qui sont composées d'au moins un élément n'appartenant pas au type de construction correspondant. Dans le premier exemple ci-dessus, nous cherchons en fait à supprimer de d_1 la valeur $(4 \otimes 1)$ car 1 n'appartient pas à $\langle Entier; ([2; 14])\rangle$.
- Les deux suivantes sont identiques à celles énoncées dans le cas des `Énumérés`, elles s'attachent à confronter les champs d'énumération lorsque les deux contiennent des valeurs effectives : elles cherchent à vider **comp-dom**.
- Les cinquième, sixième et septième règles identifient des cas évidents de δ-types inconsistants.
- Enfin, les trois dernières règles s'attachent à la réalisation des objectifs mentionnés plus haut (minimisation de T_c – règles A.12 et A.14 – et énumération systématique d'un domaine fini – règle A.13).

[1] Notons d'ores et déjà qu'une telle énumération, même si elle est possible en théorie dans la mesure où le domaine est fini, n'est pas toujours réaliste en pratique, notamment lorsque la cardinalité de ce domaine est très élevée. Malgré cela, nous présentons dans la suite la normalisation *en théorie*, et nous aborderons à la fin les problèmes rencontrés en pratique, ainsi que leurs conséquences.

Par la suite, dans les prémisses de toutes les règles, nous supposons que $T_c = t_1\chi_1 \cdots \chi_{k-1}t_k$. En outre, nous supposons les opérateurs de construction non-commutatifs, c'est-à-dire que l'ordre d'application est important. Enfin, de la même façon que pour les δ-types issus de **Énumérés**, il est supposé que l'ordre des énumérations n'a aucune importance, et ce par souci de simplification dans l'écriture et la lecture des règles[2]

$$\frac{t = \langle Construits; [T_c; v + V; d_2]\rangle, v = e_1\chi_1 \cdots \chi_{k-1}e_k, \exists j : e_j \notin t_j}{t = \langle Construits; [T_c; V; d_2]\rangle} \tag{A.5}$$

$$\frac{t = \langle Construits; [T_c; d_1; v' + V']\rangle, v' = e'_1\chi_1 \cdots \chi_{k-1}e'_k, \exists j : e'_j \notin t_j}{t = \langle Construits; [T_c; d_1; V']\rangle} \tag{A.6}$$

$$\frac{t = \langle Construits; [T_c; v + V; v + V']\rangle}{t = \langle Construits; [T_c; V; V']\rangle} \tag{A.7}$$

$$\frac{t = \langle Construits; [T_c; v_1, \cdots, v_n; v'_1, \cdots v'_p]\rangle, p, n \geq 1, \text{ t.q. } \forall i, j, v_i \neq v'_j}{t = \langle Construits; [T_c; v_1, \cdots, v_n; \emptyset]\rangle} \tag{A.8}$$

$$\frac{t = \langle Construits; [T_c; \emptyset; d_2]\rangle}{t = \perp} \tag{A.9}$$

$$\frac{t = \langle Construits; [T_c; d_1; \text{TOUT}]\rangle}{t = \perp} \tag{A.10}$$

$$\frac{t = \langle Construits; [T_c; d_1; d_2]\rangle, \exists j \in [1; k] : t_j = \perp}{t = \perp} \tag{A.11}$$

$$\frac{\begin{array}{c} t = \langle Construits; [T_c; \text{TOUT}; v'_1, \cdots, v'_p]\rangle, p \geq 1, \exists j \in [1; k] : \\ \exists m = (Card(t_1) * \cdots Card(t_{j-1}) * Card(t_{j+1}) * \cdots * Card(t_k)), m \leq p \\ \text{et } \forall i \in [1; m], v'_i = e'_{1i}\chi_1 \cdots \chi_{k-1}e'_{ki} \text{ avec } e'_{ji} = e \in t_j \\ \text{Soit } t'_j = t_j \backslash_{T_j} NormalForm_{T_j}(\textbf{domaine} \longleftarrow \{e\}) \end{array}}{t = \langle Construits; [T_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k; \text{TOUT}; v'_{m+1}, \cdots, v'_p]\rangle} \tag{A.12}$$

$$\frac{\begin{array}{c} t = \langle Construits; [T_c; \text{TOUT}; d_2]\rangle, \exists m \neq +inf \text{ t.q.} \\ m = Card(t_1) * \cdots * Card(t_k) \\ \text{Soient } v_i = e_{1i}\chi_1 \cdots \chi_{k-1}e_{ki}, i \in [1..m], \text{t.q. } \forall j \in [1; k], e_{ji} \in t_j \end{array}}{t = \langle Construits; [T_c; v_1, \cdots, v_m; d_2]\rangle} \tag{A.13}$$

$$\frac{\begin{array}{c} t = \langle Construits; [T_c; v_1, \cdots, v_n; \emptyset]\rangle, n \geq 1, \exists j \in [1; k] : \\ \text{où } \forall i, v_i = e_{1i}\chi_1 \cdots \chi_{j-1}e_{ji}\chi_j \cdots \chi_{k-1}e_{ki} \\ t'_j = NormalForm_{T_j}(\textbf{domaine} \longleftarrow \{e_{j1}, \cdots, e_{jn}\}), \\ \text{et } t'_j <_\tau t_j \end{array}}{t = \langle Construits; [T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k; v_1, \cdots, v_n; \emptyset]\rangle} \tag{A.14}$$

La règle A.13 vise à énumérer T_c lorsque sa cardinalité est finie. Les règles A.12 et A.14, quant à elles, ont pour but de minimiser T_c. Lorsque le domaine des valeurs autorisées est énuméré, T_c est reconstruit à partir de cette énumération, et s'il s'avère que cette reconstruction mène à un sous-type de T_c, alors T_c est minimisé vers ce sous-type (règle A.14). Dualement, si un élément d'un des δ-types référencés est systématiquement refusé dans **comp-dom** (*i.e.* toutes les valeurs

[2] Les trois dernières règles de ce système ne semblent pas être de pures réécritures syntaxiques, dans la mesure où elles font appel à des constructions et interprétations de domaines de valeurs, et ce dans les prémisses. Indépendamment de la réelle nature de cet ensemble de règles, nous chercherons à montrer sa confluence et sa terminaison, car nous pensons qu'il peut se ramener à un système de réécriture (certes très compliqué et illisible !), en présumant que le δ-sous-typage et l'appartenance d'une valeur à un δ-type peuvent aussi se définir syntaxiquement. Nous parlerons par la suite de *système de transition*.

construites sur cet élément sont interdites), alors cet élément est enlevé du δ-type concerné dans T_c (règle A.12). Illustrons cette dernière règle par un exemple. Soit

$$\delta t = \langle Construit; [T_c; \text{TOUT}; (1{\times}4{\times}6)\ (1{\times}4{\times}7)\ (1{\times}4{\times}8)\ (2{\times}4{\times}6)\ (2{\times}4{\times}7)\ (2{\times}4{\times}8)\ (1{\times}9{\times}6)\ (2{\times}12{\times}8)]\rangle$$

avec

$$T_c = \langle Entier; ([1;2])\rangle \times \langle Entier; ([2;+inf])\rangle \times \langle Entier; ([6;8])\rangle$$

Alors, avec les notations de la règle, si $j = 2$,

$$t_j = t_2 = \langle Entier; ([2,+inf])\rangle \text{ et } m = 2 * 3 = 6 \text{ et } p = 8$$

or parmi **comp-dom**, il existe bien $m = 6$ valeurs qui possèdent l'élément 4 à la position $j = 2$; cela signifie qu'aucune valeur ne peut être construite avec 4 en seconde position. On élimine donc 4 de t_2, et toutes ces valeurs interdites de **comp-dom**. On obtient

$$T'_c = \langle Entier; ([1;2])\rangle \times \langle Entier; ([2;3]\,[5;+inf])\rangle \times \langle Entier; ([6;8])\rangle$$

et donc

$$\delta t^N = \langle construits; [T'_c; \text{TOUT}; (1 \times 9 \times 6)\ (2 \times 12 \times 8)]\rangle$$

Confluence et terminaison

Afin de prouver tant la confluence que la terminaison de ce système, il convient de distinguer deux principaux ensembles de règles :

- les sept premières qui, prises ensemble, amènent automatiquement à l'une des quatre formes terminales souhaitées, selon le même schéma que dans le cas des `Énumérés`, mais qui ne garantissent pas l'unicité de cette forme terminale,
- les trois dernières, qui ont pour rôle d'assurer l'unicité de la forme obtenue.

Ainsi, ce système de transition doit être étudié en deux temps. Tout d'abord, nous montrons que les sept premières règles produisent une forme terminale, qui n'est pas forcément unique : les sept premières règles terminent. Nous verrons alors que, quelle que soit la forme terminale obtenue, si elle n'est pas une forme normale, alors elle vérifie automatiquement les prémisses d'une règle parmi A.12, A.13 et A.14 qui traduisent l'équivalence dénotationnelle entre différentes formes terminales. Ce sont ces trois dernières règles qui garantissent la confluence du système vers des formes normales.

La figure A.1 illustre l'automate d'états finis directement associé au système de transition ci-dessus. Montrons tout d'abord que les états correspondants aux règles A.5, A.6 et A.7 ne sont pas des puits, c'est-à-dire qu'il n'y a pas de risque de boucle et que le système de transition en sort automatiquement.

- Partant d'un **domaine** énuméré donc fini, et d'un **comp-dom** quelconque, la règle A.5 se contente d'enlever un élément de **comp-dom** : une boucle n'est donc pas possible, puisqu'un état d'arrêt est automatiquement atteint (**comp-dom** vidé : la prémisse n'est plus vérifiée), ce qui mène alors à une forme terminale de type 2.
- Partant d'un **comp-dom** énuméré donc fini, et d'un **domaine** quelconque, la règle A.6 n'a pour effet que de diminuer strictement la cardinalité de **domaine** ; il n'y a donc pas de risque de boucle puisqu'une application itérée de cette règle mènerait à un **domaine** vide, donc ne vérifiant plus sa prémisse mais celle de la règle A.9.

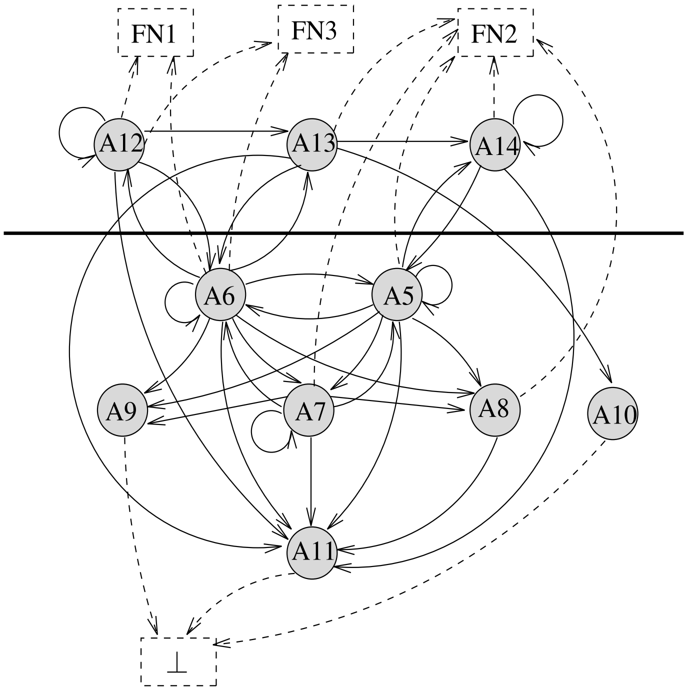


FIG. A.1 - : Automate associé au système de transition correspondant à la normalisation des `Construits`. Tous les états sont des points d'entrée de l'automate. Si nous choisissions d'ordonner l'application de ces règles, cet automate serait réduit ; en particulier, et pour des raisons évidentes de complexité, nous avons défini un ordre d'application de ces règles au niveau de l'algorithme de normalisation.

- Partant d'un **domaine** et d'un **comp-dom** énumérés donc finis, la règle A.7 se contente de diminuer strictement la cardinalité des deux énumérations. Il n'y a donc pas de risque de boucle, puisque l'une de ces deux énumérations peut se vider, les prémisses ne sont alors plus vérifiées : dans un tel cas, soit la règle A.9 s'applique (**domaine** vidé), soit une forme terminale de type 2 est atteinte.

- Globalement, ces trois règles ne présentent pas de risque de boucle puisqu'elles ont toutes pour effet de restreindre conjointement les mêmes ensembles finis de valeurs. En outre, nous vérifions que lorsque les prémisses ne sont plus vérifiées, il existe une transition vers une autre partie de l'automate[3].

  - Lorsque la prémisse de A.5 n'est *plus* vérifiée (ce qui signifie que l'énumération de **domaine** ne possède plus de valeur inconsistante avec **type-ref**), **domaine** étant encore une énumération (éventuellement vide), et selon **comp-dom**, soit une forme terminale de type 2 est atteinte, soit au moins une des règles parmi A.8, A.9, A.10, A.11 est applicable (nous supposons que les prémisses des règles A.6 et A.7 ne sont plus vérifiées).
  - Lorsque la prémisse de A.6 n'est *plus* vérifiée, (ce qui signifie que l'énumération de **comp-dom** ne possède plus de valeur inconsistante avec **type-ref**), donc **comp-dom** étant encore une énumération (ou $\emptyset$), et selon la nature de **domaine**, soit une forme normale de type 3 (ou de type 2) est atteinte, soit au moins une des règles A.8, A.9, A.11 est applicable (nous supposons que les prémisses des règles A.5 et A.7 ne sont plus vérifiées).
  - Lorsque la prémisse de A.7 n'est *plus* vérifiée (ce qui signifie qu'il n'existe plus de valeur redondante – indépendamment de **type-ref** – dans **domaine** et **comp-dom**), **domaine**

[3] Nous ne considérons pas ici la possibilité d'application des règles A.12 à A.14 car elles ne s'appliquent qu'aux formes terminales : nous indiquons comment les formes terminales sont atteintes à l'issue de l'application des règles A.5 à A.7.

et **comp-dom** étant des énumérations (éventuellement vides), soit une forme terminale de type 2 est atteinte, soit au moins l'une des règles parmi A.8, A.9, A.11 est applicable (nous supposons que les prémisses des règles A.5 et A.6 ne sont plus vérifiées).

Ceci nous assure donc que le triplet ⟨A.5,A.6,A.7⟩ n'est pas un puits de l'automate : quand on y rentre on en sort.

Les règles A.8, A.9, A.10 et A.11, prises ensemble, ne présentent aucun risque de boucle : elles mènent toutes obligatoirement à une forme terminale.

Nous avons donc montré que, indépendamment des règles A.12, A.13 et A.14 qui permettent l'obtention de formes *normales*, le système de transition composé seulement des sept premières règles termine vers une forme terminale. Montrons maintenant la confluence locale de ce système réduit aux sept premières règles : pour un δ-type, un seul type de forme terminale peut être atteint. Ceci est vérifié immédiatement, en s'assurant qu'aucune règle, lorsqu'il y a choix avec un autre, n'empêche l'application ultérieure de l'autre (à moins que les effets des deux règles soient équivalents). En effet, les quatre premières règles ont pour effet d'éliminer des valeurs des énumérations :

- lorsqu'une même valeur peut être éliminée pour plusieurs raisons (cette valeur est inconsistante, ou n'appartient pas à **domaine**, ou n'appartient pas à **type-ref**, ou appartient à **comp-dom**), le choix de la règle n'a pas d'importance puisque cette valeur sera éliminée de toute façon ;
- lorsque deux valeurs peuvent être éliminées au même moment, l'application d'une des deux règles ne peut empêcher l'application de l'autre. Notons ici le cas particulier de A.7 qui, lorsque v est inconsistante, est équivalente à l'application successive (et quel que soit l'ordre) de A.5 et A.6.

Les trois règles suivantes mènent directement au type inconsistant, donc le choix entre les trois ne change pas le résultat. Enfin, il est important de noter que le choix d'une des quatre premières règles, lorsqu'une alternative est l'application d'une des trois suivantes, ne peut pas modifier la vérification de la prémisse d'une de ces trois (**type-ref** n'est jamais modifié par une des quatre premières, pas plus que **comp-dom** = TOUT, ni même **domaine** = $\emptyset$). Ainsi, nous avons montré que le non-déterminisme sur le système de transition réduit aux sept premières règles n'influe pas sur le résultat produit : nous avons montré une confluence locale.

Nous nous attachons maintenant à montrer la terminaison et la confluence du système de transition global, c'est-à-dire celui prenant en compte les trois dernières règles : A.12, A.13 et A.14. Montrons en premier lieu qu'individuellement, ces trois règles terminent.

- La règle A.12 ne peut boucler indéfiniment sur elle-même car, partant d'une énumération finie de **comp-dom**, elle a pour effet de diminuer la cardinalité de cette énumération, au pire jusqu'à une cardinalité nulle, le δ-type ne vérifiant plus alors la prémisse.
- La règle A.13 ne peut boucler sur elle-même, puisque la condition sur le champ **domaine** change.
- Il y aurait, en ce qui concerne la règle A.14, deux possibilités d'itération infinie, pourtant rendues impossibles du fait de la nature des règles.
  - La règle peut s'appliquer un nombre infini de fois sur le même type référencé (t_j) : ceci est impossible, car pour que cette règle s'applique plusieurs fois sur un même type

référencé, il faut que l'énumération de **domaine** change entre deux applications. Or cette énumération est finie au départ, et ne fait que décroître en cardinalité, éventuellement jusqu'à une énumération vide : la prémisse n'est alors plus vérifiée.

– La règle boucle sur le nombre de types référencés: ceci est impossible, car ils sont en nombre fini, et ne sont que réduits, au pire jusqu'au type inconsistant. Dans ce cas, la prémisse n'est plus vérifiée car il n'existe pas de δ-type δ-sous-type de $\perp$.

Donc la règle A.14 n'est pas un puits de l'automate.

– Globalement, ces trois règles prises ensemble ne présentent pas de risque de boucle. Elles mènent soit à une forme terminale, soit à une règle de réduction des énumérations.

Ainsi, le système de transition réduit aux règles A.12, A.13 et A.14 termine : quand on y entre, on en sort. Montrons maintenant la confluence locale de ce système réduit : quels que soient les choix de règles effectués, la forme terminale obtenue pour un δ-type finit toujours par être du même type de forme terminale. Parmi ces règles, et compte-tenu de leurs prémisses, les trois règles ne peuvent pas être possibles au même moment : nous ne sommes en présence que d'un seul choix binaire, que nous étudions ci-dessous afin de montrer que ce choix n'a pas de conséquence sur le résultat : il s'agit du choix à effectuer entre l'application de A.12 et celle de A.13.

Soit $t = \langle Construits; [T_c; \text{TOUT}; v'_1, \cdots, v'_p]\rangle, p \geq 1$, le δ-type concerné par le choix, avec $Card(T_c) = n$. Étudions le devenir de t selon les deux choix initiaux possibles.

Choix de la règle A.13 alors t est réécrit comme suit :

$$\langle Construits; [T_c; v_1, \cdots, v_n; v'_1, \cdots, v'_p]\rangle$$

avec les trois conditions ci-dessous. $\exists p, q : 0 \leq m \leq q \leq p$ t.q.:

1. $\forall i \in [1; m], v'_i \in \{v_1, \cdots, v_n\}$ et $\exists j \in [1; k]$ t.q. $e'_{ji} = e \in t_j$
2. $\forall r \in [m+1; q], v'_r \in \{v_1, \cdots, v_n\}$
3. $\forall s \in [q+1; p], v'_s \notin \{v_1, \cdots, v_n\}$

À ce stade, les règles A.6 et A.5 peuvent s'appliquer, mais elles ne changeraient rien à la nature des trois conditions suivantes puisqu'elles se contentent d'enlever des énumérations les valeurs inconsistantes. Ici, seule la règle A.7 peut s'appliquer m fois pour enlever les v'_i de **domaine** et de **comp-dom** : on obtient alors

$$\langle Construits; [T_c; v_{m+1}, \cdots, v_n; v'_{m+1}, \cdots, v'_p]\rangle$$

À ce stade, si $\{v_{m+1}, \cdots, v_n\}$ est vide, la règle A.9 s'applique et on obtient $\perp$. Sinon, la règle A.7 est encore la seule à pouvoir s'appliquer $q - m$ fois (condition 2), puis la règle A.8 (condition 3) pour vider le champ **comp-dom**. L'ordre n'a pas d'importance comme nous l'avions montré précédemment. Nous obtenons alors

$$\langle Construits; [T_c; v_{q+1}, \cdots, v_n; \emptyset]\rangle$$

La règle A.9 peut s'appliquer ici (ou avant l'application de A.8) si $\{v_{q+1}, \cdots, v_n\}$ est vide : on aboutit dans ce cas à $\perp$. Sinon (**domaine** n'est pas vide), seule la règle A.14 peut s'appliquer, d'après la forme syntaxique de t, et de plus il est certain qu'à ce stade, **domaine** ne contient pas les m valeurs qui avaient toutes, à la position j, l'élément $e \in t_j$: la prémisse de A.14 est vérifiée, la règle est appliquée et nous obtenons :

$$\langle Construits; [T'_c; v_{q+1}, \cdots, v_n; \emptyset]\rangle$$

où $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ et $t'_j = t_j \backslash e$. À ce stade, soit $t'_j = \perp$ et la règle A.11 s'applique menant à $t = \perp$, soit nous avons obtenu une *forme normale de type 2*.

Choix de la règle A.12 alors t est réécrit comme suit :

$$\langle Construits; [T'_c; \text{TOUT}; v'_{m+1}, \cdots, v'_p]\rangle$$

avec $Card(T'_c) = n - m$, et $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ et $t'_j = t_j \backslash e$. À ce stade, on peut avoir (comme à la fin de la réécriture précédente) $t'_j = \perp$ et la règle A.11 s'applique menant à $t = \perp$ (cette règle pourra toujours s'appliquer après si elle n'est pas choisie tout de suite ; quoi qu'il en soit, nous avons ici le même résultat que pour le choix précédent avec les mêmes hypothèses). Si $t'_j \neq \perp$, alors seule la règle A.13 peut s'appliquer (les applications à tout moment des règles A.5 et A.6 n'auraient aucune conséquence)[4]. Nous obtenons alors

$$\langle Construits; [T_c; v_1, \cdots, v_{n-m}; v'_{m+1}, \cdots, v'_p]\rangle$$

c'est-à-dire (pour les besoins de lisibilité de cette preuve) :

$$\langle Construits; [T_c; v_{m+1}, \cdots, v_n; v'_{m+1}, \cdots, v'_p]\rangle$$

avec comme conditions (que nous reprenons évidemment des hypothèses du choix premier de A.13). $\exists q : m \leq q \leq p$ t.q. :

1. $\forall r \in [m+1; q]$ t.q. $v'_r \in \{v_{m+1}, \cdots, v_n\}$
2. $\forall s \in [q+1; p]$ t.q. $v'_s \notin \{v_{m+1}, \cdots, v_n\}$
3. mais la première condition du cas précédent n'est plus vérifiée puisque ces valeurs ont été éliminées lors de l'application de A.12.

La règle A.14 ne peut pas s'appliquer ici car l'énumération de **domaine** correspond exactement à T_c : ce dernier ne peut donc pas être minimisé.

À ce stade, la règle A.9 peut s'appliquer si $\{v_{m+1}, \cdots, v_n\}$ est vide, on aboutit alors à $t = \perp$ (comme dans le cas précédent). Sinon, seule la règle A.7 est applicable $q - m$ fois (condition 1), puis la règle A.8 est applicable une fois (condition 2). On obtient alors

$$\langle Construits; [T'_c; v_{q+1}, \cdots, v_n; \emptyset]\rangle$$

À ce stade, la règle A.9 peut s'appliquer (ou aurait pu s'appliquer à l'étape précédente) si $\{v_{q+1}, \cdots, v_n\} = \emptyset$, pour donner $t = \perp$ comme dans le cas précédent. Sinon, plus aucune règle ne peut s'appliquer et nous sommes en présence d'une *forme normale de type 2* équivalente à celle obtenue précedemment lorsque le premier choix s'est porté sur A.13.

Dans les deux cas il est à noter que la règle A.11 menant à $\perp$ est *toujours* applicable à tout moment (même à la fin), dès que sa prémisse est vérifiée : donc le choix entre A.11 et une autre règle ne changera jamais le résultat final, à savoir $\perp$.

Nous avons traité ci-dessus le seul cas de non-déterminisme une fois qu'une forme terminale est obtenue, donc le système de transition composé des règles A.5 à A.14 vérifie les propriétés de terminaison et de confluence.

Il reste à montrer que les formes terminales, obtenues après épuisement des possibilités d'application des trois dernières règles du système de transition, sont bien des formes normales, c'est-à-dire

[4] Notons toutefois que l'application multiple de A.12 est bien entendu envisageable, mais ne change rien à la nature des conditions postérieures.

que $\forall i, j \in [1;4], i \neq j, \|FN_i\|^\xi \neq \|FN_j\|^\xi$, si FN_i dénote la ième forme terminale parmi les quatre obtenues. Pour cela, étudions les quatre formes normales :

1. $FN_1 = \langle Construits; [T_c; \text{TOUT}; \emptyset] \rangle$ qui dénote $\|T_c\|^\xi$. Cet ensemble dénoté est forcément infini, car sinon la règle A.13 se serait appliquée, menant à FN_2, ou bien à $FN_4 = \perp$ si l'un des types de **type-ref** avaient été inconsistant (par application de A.11).

2. $FN_2 = \langle Construits; [T_c; v_1, \cdots, v_n; \emptyset] \rangle$ qui dénote $\{v_1, \cdots, v_n\}$. Cet ensemble de valeurs dénoté est forcément fini, du fait de l'énumération de **domaine**, est non vide car sinon la règle A.9 se serait appliquée si **domaine** avait été vide, et A.11 se serait appliquée si **type-ref** avait eu un δ-type inconsistant. En outre, la règle A.14 minimisant T_c à partir de l'énumération du domaine fini, il n'existe pas deux δ-types sous FN_2 différents (donc dont les **type-ref** sont différents) dénotant le même domaine. Ceci se prouve aisément par l'absurde et par considération des conditions de la prémisse de A.14, et de la conclusion correspondante.

3. $FN_3 = \langle Construits; [T_c; \text{TOUT}; v'_1, \cdots, v'_p] \rangle$ qui dénote $\|T_c\|^\xi \backslash \{v'_1, \cdots, v'_p \rangle\}$. Cet ensemble de valeurs dénoté est forcément infini, car sinon la règle A.13 se serait appliquée. Il n'est en outre pas vide car cela signifierait obligatoirement, soit que $\|T_c\|^\xi \subseteq \{v'_1, \cdots, v'_p\}$ donc la règle A.13 se serait appliquée, soit que $\|T_c\|^\xi = \emptyset$ mais alors la règle A.9 aurait pu s'appliquer. Enfin, la règle A.12 minimisant **comp-dom** et **type-ref**, il est aisé de montrer par l'absurde que deux δ-types sous FN_3 différents (donc dont soit les **type-ref**, soit les **comp-dom**, sont différents) ne dénotent pas le même ensemble de valeurs à moins qu'une autre règle puisse s'appliquer sur l'un des deux pour obtenir une expression syntaxiquement équivalente à l'autre.

4. $FN_4 = \perp$ qui dénote $\emptyset$. Nous avons vu que les trois précédentes formes ne dénotent jamais l'ensemble vide : FN_4 est donc bien une forme normale.

Pour récapituler, disons qu'il n'existe qu'une forme d'expression des ensembles de valeurs finis non vides (FN_2), une seule forme d'expression d'un ensemble vide (FN_4), mais deux formes d'expressions d'ensembles de valeurs infinis (FN_1 et FN_3). Pourtant, au vu de leurs dénotations, ces deux formes n'exprimeront jamais le même ensemble de valeurs. Ceci se montre aisément par l'absurde en considérant l'effet de la règle A.12 qui minimise les deux champs. En conclusion, les quatre formes terminales produites par le système de transition constitué des règles A.5 à A.14 sont bien des formes normales.

Notons ici que la conception des trois dernières règles de ce système a été guidée par l'énumération systématique de l'ensemble de valeurs dénoté lorsque ce dernier est connu pour être fini, ce qui mène alors le champ **type-ref** à jouer un rôle secondaire. On se souviendra de ce principe de base de la normalisation dans le cas des multivalués (construits unaires).

Nous avons montré dans les quelques pages précédentes la terminaison, la confluence du système de transition correspondant aux `Construits`, ainsi que l'unicité des formes terminales obtenues.

Complétude et correction

Maintenant que nous avons montré la confluence et la terminaison du système de transition, ainsi que l'unicité des formes terminales, nous nous attachons à montrer la complétude de ce système ainsi que sa correction vis-à-vis des ensembles dénotés par les δ-types.

Tous les états de l'automate associé au système de transition sont des points d'entrée, et le système termine et conflue. Donc il suffit que le δ-type initial respecte au moins la prémisse d'une des dix règles de ce système pour que la complétude soit assurée, à moins que le δ-type initial soit *déjà* sous forme normale. Soit $T_c = t_1\chi_1 \cdots \chi_{k-1}t_k, k \geq 1$; soit $t = \langle Construits; [T_c; d_1; d_2]\rangle$ le δ-type à normaliser, qui vérifie les pré-conditions[5].

- Si T_c est inconsistant ($\exists j \in [1;k]$ t.q. $t_j = \perp$), alors la règle A.11 est un point d'entrée.
- Si T_c a une cardinalité finie ($\exists m : \forall j \in [1;k], Card(t_j) \leq m$), alors la règle A.13 est un point d'entrée.
- Si T_c a une cardinalité infinie, alors examinons toutes les combinaisons possibles des deux autres champs :
  1. $t = \langle Construits; [T_c; \text{TOUT}; \emptyset]\rangle$, alors t est déjà sous forme normale ;
  2. $t = \langle Construits; [T_c; \text{TOUT}; v'_1, \cdots, v'_p]\rangle$, alors t est déjà sous forme normale, à moins qu'il vérifie les prémisses de A.5 et/ou A.12 ;
  3. $t = \langle Construits; [T_c; v_1, \cdots, v_n; v'_1, \cdots, v'_p]\rangle$, alors t vérifie au moins l'une des quatre règles parmi A.5, A.6, A.7 et A.7 ;
  4. $t = \langle Construits; [T_c; v_1, \cdots, v_n; \emptyset]\rangle$, alors t est déjà sous forme normale à moins que A.5 soit applicable et/ou qu'il vérifie les prémisses de A.14 ;
  5. $t = \langle Construits; [T_c; \emptyset; d_2]\rangle$, alors la règle A.9 constitue un point d'entrée (la sortie est rapide !) ;
  6. $t = \langle Construits; [T_c; d_1; \text{TOUT}]\rangle$, alors la règle A.10 s'applique et constitue donc un point d'entrée possible.

Nous avons traité ci-dessus tous les cas de δ-types possibles en entrée, et nous avons vu qu'à chacun d'eux est associé un point d'entrée dans l'automate à moins d'être déjà sous forme normale. La complétude est ainsi montrée, puisqu'en outre le système termine.

La correction est vérifiée si nous montrons que chaque règle du système préserve l'invariance de l'ensemble dénoté par le δ-type vérifiant la prémisse. Le tableau A.2 fait office de preuve, sa lecture est similaire à celle commentée du tableau A.1. Dans tout le tableau, $T_c = t_1\chi_1 \cdots \chi_{k-1}t_k$, et $\|T_c\|^\xi = \|t_1\|^\xi\|\chi_1\|^\xi \cdots \|\chi_{k-1}\|^\xi\|t_k\|^\xi$, cette notation étant abrégée en $\|t_1\chi_1 \cdots \chi_{k-1}t_k\|^\xi$; de même, une valeur v_i (ou v'_i bien sûr) est exprimée par $v_i = e_{1i}\chi_1 \cdots \chi_{k-1}e_{ki}$ avec $\|v_i\|^\xi = \|e_{1i}\|^\xi\|\chi_1\|^\xi \cdots \|\chi_{k-1}\|^\xi\|e_{ki}\|^\xi$, cependant nous considérons, dans ce tableau, et ce par souci de lisibilité, que $\|e_{ji}\|^\xi = e_{ji}$, $\|\chi_j\|^\xi = \chi_j$ et enfin $\|v_i\|^\xi = v_i$. Les abbréviations CS et NF signifient respectivement *Construits* et $NormalForm$.

Normalisation des `Record`

Un δ-type issu de `Record` n'est rien d'autre qu'un produit cartésien n-aire, les éléments des produits étant des associations (étiquette/δ-type). Ces associations (en fait des fonctions) peuvent être vues comme induisant un ordre dans les produits, ce qui nous amène à considérer les δ-types issus de `Record` comme $k-1$ produits ordonnés de k autres δ-types. Dans la mesure où la normalisation des δ-types issus de `Construits` suppose l'existence d'un ordre entre les applications de constructeur, nous sommes à même de considérer, dans le cas de δ-types *records*, que le seul constructeur est le produit cartésien appliqué $k-1$ fois (c'est-à-dire, $\forall i \in [1; k-1], \chi_i = \times$).

[5] Rappel : pas de doublet dans les énumérations, $\forall j \in [1;k], t_j$ est normalisé, et les énumérations ne contiennent pas de valeur qui soit construite avec z éléments, où $z \neq k$.

Grâce à cette correspondance entre *record* et $k-1$ produits ordonnés, et étant donné que la représentation syntaxique des δ-types *records* est la même que celle de δ-types issus de `Construits`, il est correct d'affirmer que la normalisation des δ-types *records* est équivalente à celle des δ-types issus de `Construits`.

Normalisation des `Multivalués`

Les multivalués sont avant tout des types construits à partir d'un seul type, c'est-à-dire, dans notre jargon, des construits unaires. De ce fait, la normalisation des δ-types issus de `Multivalués` est très similaire à celle des δ-types issus de `Construits`, en tout cas dans la nature des règles et leurs objectifs. Toutefois, les multivalués, comme `Liste` ou `Ensemble`, décrivent leurs δ-types avec un champ de plus, qui indique les cardinalités autorisées des valeurs construites acceptables. De plus, le champ **type-ref** ne contient plus qu'un seul δ-type. En fait, d'un point de vue de la normalisation, il s'agit simplement de voir la cardinalité comme une indication supplémentaire relative à la propriété de finitude des ensembles de valeurs dénotés.

La représentation d'un δ-type issu de `Multivalués` est donc composée de quatre champs, qui sont **type-ref**, **domaine**, **comp-dom** et **card**. Le premier de ces champs fait référence à un seul δ-type, qui se voit appliqué un constructeur (c'est le constructeur de listes (resp. d'ensembles) lorsque le C-type est `Liste` (resp. `Ensemble`)). Les deux champs suivants sont des *énumérations explicites de valeurs*, ces valeurs étant construites à partir du constructeur. Enfin, le dernier champ d'expression correspond à un intervalle de valeurs entières positives.

Nous supposons des préconditions sur les δ-types auxquels la normalisation peut s'appliquer : cohérence et absence de redondance, lorsque chaque champ de représentation est considéré indépendamment des autres. De même que dans le cas des autres C-types étudiés jusqu'ici, la construction de ces quatre champs (intervenant lors de la création d'un δ-type) garantit la cohérence et la non-redondance individuelles des contenus de chacun d'eux ; autrement dit, les énumérations ne contiennent pas de doublet, leurs valeurs ont bien la structure imposée par le constructeur considéré, et l'intervalle de cardinalité, à moins d'être vide, est bien constitué de deux bornes valuées par des entiers positifs. Enfin, le δ-type référencé par **type-ref** est sous forme normalisée. Par exemple, dans le cas des ensembles, le δ-type

$$\langle Ensemble; [\langle Entier; [1;10]\rangle; (1,2,3,1)(2,3,4); \emptyset; [2;8]]\rangle$$

ne vérifie pas les préconditions car $(1,2,3,1)$ n'a pas la structure d'ensemble. Quant au δ-type

$$\langle Ensemble; [\langle Entier; [1;10]\rangle; (1,2,12); \emptyset; [4;4]]\rangle$$

il vérifie les préconditions, même si, d'une part la valeur $(1,2,12)$ contient des éléments n'appartenant pas au δ-type référencé, et d'autre part cette valeur ne répond pas à la condition imposée par la cardinalité : nous ne considérons, dans les préconditions, que la cohérence et la non-redondance des champs, les uns pris indépendamment des autres.

Ainsi, cette cohérence et absence de redondance doit être assurée lors de la normalisation, en confrontant les différents champs d'expression. En cas d'incohérence, le δ-type $\perp$ sera produit.

Rappelons la syntaxe d'un δ-type issu de `Multivalués` :

$$t = \langle Multiv; [\chi(\delta t); d_1; d_2; c]\rangle$$

où d_1 et d_2 sont, soit $\emptyset$, soit le symbole TOUT, soit une énumération finie de valeurs $v_1, \cdots, v_n = v_1 + V$ où $+$ dénote la concaténation. Les valeurs de ces énumérations sont construites par χ, on note

$v_i = \chi(e_{1i}, \cdots, e_{ni}), n \geq 1$, et $Nb(v_i) = n$, qui représente le nombre d'éléments qui composent v_i. Enfin, c est un intervalle de valeurs entières positives $[a; b]$ où b peut être infini. Nous rappelons l'interprétation dénotationnelle d'un δ-type issu de `Multivalués` à partir de l'information contenue dans les champs :

$$
\begin{array}{rcl}
\|t\|^{\xi} & = & \{v \in (\|\chi\|^{\xi}(\|\delta t\|^{\xi}) \cap \|d_1\|^{\xi}) \backslash \|d_2\|^{\xi} \mid Nb(v) \in \|c\|^{\xi}\} \\
\|\text{TOUT}\|^{\xi} & = & \|\chi\|^{\xi}(\|\delta t\|^{\xi}) \\
\|\emptyset\|^{\xi} & = & \emptyset \\
\|v_1, \cdots, v_n\|^{\xi} & = & \{v_1, \cdots, v_n\} \\
\|c = [a; b]\|^{\xi} & = & \{e \in \|Entier\|^{\xi} \mid a \leq e \leq b\}
\end{array}
$$

Les quatre formes normales que nous cherchons à obtenir pour les δ-types issus de `Multivalués` sont les suivantes :

1. $\langle Multi; [\chi(\delta t); \text{TOUT}; \emptyset; [a; b]]\rangle$ qui dénote $\{v \in \|\chi\|^{\xi}(\|\delta t\|^{\xi}) \mid Nb(v) \in \|[a; b]\|^{\xi}\}$,

2. $\langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a; b]]\rangle$ qui dénote $\{v_1, \cdots, v_n\}$,

3. $\langle Multi; [\chi(\delta t); \text{TOUT}; v'_1, \cdots, v'_p; [a; b]]\rangle$ qui dénote
$$\{v \in \|\chi\|^{\xi}(\|\delta t\|^{\xi}) \mid Nb(v) \in \|[a; b]\|^{\xi}\} \backslash \{v'_1, \cdots, v'_p\}$$

4. $\perp$ qui dénote $\emptyset$.

Comme dans le cas des `Construits`, les formes normales 1 et 3 sont destinées à représenter des ensembles infinis de valeurs, les ensembles finis non vide seront automatiquement sous la forme normale 2. Nous chercherons toujours à minimiser δt (en supprimant, lorsque cela est possible, des valeurs qui lui appartiennent et qui n'appartiennent pas au domaine énuméré). Par ailleurs, dès qu'il sera possible d'inférer la finitude de l'ensemble dénoté, l'énumération sera effectuée (cette inférence sera rendue possible par confrontation de la cardinalité imposée des valeurs et du type référencé) ; par là-même, l'intervalle de cardinalités autorisées sera minimisé : autrement dit, une fois que le δ-type à normaliser dénote un ensemble fini de valeurs, les champs **type-ref** et **card** deviennent secondaires, mais on cherchera toutefois à les minimiser à partir de l'énumération, pour s'assurer que les formes terminales obtenues sont bien des formes normales. Les règles du système de transition établi répondent à ces objectifs. Elles peuvent être partitionnées en trois groupes à vocations distinctes.

- Les quatre premières éliminent des énumérations les valeurs qui soit ont des éléments n'appartenant pas au type référencé, soit ont une cardinalité n'appartenant pas à l'intervalle de cardinalités exigé. Leur rôle commun se compare à celui des règles A.5 et A.6 élaborées pour les construits n-aires.

- Les huit suivantes visent à éliminer des redondances qui surgissent de la confrontation des champs **domaine**, **comp-dom** et **card**, ainsi qu'à détecter des situations évidentes d'inconsistances. Leur rôle se compare à celui des règles A.7 à A.11 des construits n-aires. Notons le rôle particulier de la règle A.26 qui ne s'applique qu'aux δ-types issus du constructeur ensembliste : elle est nécessaire à la réduction d'un intervalle de cardinalités intrinsèquement borné lorsque les éléments possibles composant les ensembles à construire sont en nombre limité.

– Enfin, les quatre dernières règles ont pour rôle de minimiser le type référencé, avec comme principe de base l'énumération du domaine lorsque cela est possible (ce qui mène alors à une minimisation de l'intervalle de cardinalités), avec comme finalité l'obtention de formes normales. Leur rôle est comparable à celui des règles A.12 à A.14 du système de transition des construits n-aires. Notons que les prémisses de ces quatre dernières règles font qu'elles ne peuvent s'appliquer qu'à des δ-types sous forme terminale.

$$\frac{t = \langle Multi;[\chi(\delta t); v+V; d_2; c]\rangle, v = \chi(e_1, \cdots e_k), \exists j \in [1;k] : e_j \notin \delta t}{t = \langle Multi;[\chi(\delta t); V; d_2; c]\rangle} \tag{A.15}$$

$$\frac{t = \langle Multi;[\chi(\delta t); d_1; v'+V'; c]\rangle, v' = \chi(e'_1, \cdots, e'_k), \exists j \in [1;k] : e'_j \notin \delta t}{t = \langle Multi;[\chi(\delta t); d_1; V'; c]\rangle} \tag{A.16}$$

$$\frac{t = \langle Multi;[\chi(\delta t); v+V; d_2; [a;b]]\rangle, v = \chi(e_1 \cdots e_k), k \notin [a;b]}{t = \langle Multi;[\chi(\delta t); V; d_2; [a;b]]\rangle} \tag{A.17}$$

$$\frac{t = \langle Multi;[\chi(\delta t); d_1; v'+V'; [a;b]]\rangle, v' = \chi(e'_1 \cdots e'_k), k \notin [a;b]}{t = \langle Multi;[\chi(\delta t); d_1; V'; [a;b]]\rangle} \tag{A.18}$$

$$\frac{t = \langle Multi;[\chi(\delta t); v+V; v+V'; c]\rangle}{t = \langle Multi;[\chi(\delta t); V; V'; c]\rangle} \tag{A.19}$$

$$\frac{t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; v'_1, \cdots v'_p; c]\rangle, p, n \geq 1, \text{t.q. } \forall i, j, v'_i \neq v_j}{t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; \emptyset; c]\rangle} \tag{A.20}$$

$$\frac{t = \langle Multi;[\chi(\delta t); \emptyset; d_2; c]\rangle}{t = \perp} \tag{A.21}$$

$$\frac{t = \langle Multi;[\chi(\delta t); d_1; \text{TOUT}; c]\rangle}{t = \perp} \tag{A.22}$$

$$\frac{t = \langle Multi;[\chi(\perp); d_1; d_2; c]\rangle}{t = \perp} \tag{A.23}$$

$$\frac{t = \langle Multi;[\chi(\delta t); d_1; d_2; \emptyset]\rangle}{t = \perp} \tag{A.24}$$

$$\frac{t = \langle Multi;[\chi(\delta t); d_1; d_2; [a;b]]\rangle, a > b}{t = \perp} \tag{A.25}$$

$$\frac{t = \langle Ensemble;[\text{Ens}(\delta t); d_1; d_2; [a;+inf]]\rangle, Card(\delta t) = m}{t = \langle Ensemble;[\text{Ens}(\delta t); d_1; d_2; [a;m]]\rangle} \tag{A.26}$$

$$\frac{\begin{array}{c} t = \langle Multi;[\chi(\delta t); \text{TOUT}; v'_1, \cdots, v'_p; [a;b]]\rangle, p \geq 0, b \neq +inf \\ \exists m = Card(\delta t), \text{Soit } n = \sum_{z=a}^{b} \mathcal{C}_m^z \\ \text{Soient } v_i = \chi(e_{1i}, \cdots, e_{ki}) \text{ t.q. } k \in [a;b], i \in [1;n], \text{ et } \forall j \in [1;k], e_{ji} \in \delta t \end{array}}{t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; v'_1, \cdots, v'_p; [a;b]]\rangle} \tag{A.27}$$

$$\frac{\begin{array}{c} t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a;b]]\rangle, n \geq 1, \\ \delta t' = NormalForm_T(\textbf{domaine} \longleftarrow \{e_{ji}\}_{j \in [1;Nb(v_i)], i \in [1;n]}) \\ \text{et } \delta t' <_\tau \delta t \end{array}}{t = \langle Multi;[\chi(\delta t'); v_1, \cdots, v_n; \emptyset; [a;b]]\rangle} \tag{A.28}$$

$$\frac{\begin{array}{c} t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a;b]]\rangle, a \leq b, \nexists i \in [1;n] : \\ v_i = \chi(e_{1i}, \cdots, e_{ki}) \text{ où } k = a \end{array}}{t = \langle Multi;[\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a+1;b]]\rangle} \tag{A.29}$$

$$\frac{\begin{array}{c} t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a; b]] \rangle, a \leq b, \\ \text{Soit } m = Max(\{Nb(v_i)\}_{i \in [1;n]}) \text{ et } m < b \end{array}}{t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a; m]] \rangle} \qquad (A.30)$$

où $\mathcal{C}_m^z$ est la combinaison de z éléments parmi n.

Comparaisons avec le système de transition précédent

De même que dans le cas de la normalisation des δ-types issus de `Construits`, le système de transition réservé aux δ-types issus de `Multivalués` s'observe selon deux axes distincts.

- les douze premières règles qui, prises ensemble, amènent automatiquement à l'une des quatre formes terminales souhaitées, en éliminant toute redondance lorsque les champs **comp-dom**, **domaine** et **card** sont considérés simultanément. Mais ces règles ne garantissent en aucun cas que les formes terminales obtenues sont des formes normales vis-à-vis des ensembles de valeurs dénotés par les δ-types réécrits. De même que dans le cas des construits n-aires, ces douzes règles se divisent en trois groupes (détections d'incohérences ne menant pas à l'inconsistance directe du δ-type, élimination des redondances évidentes et détection d'inconsistances directes).
- les quatre dernières règles, qui ont justement pour rôle d'assurer l'unicité de la forme obtenue.

Même si ces deux groupes de règles ont des rôles bien distincts, les applications des règles du premier groupe n'ont pas à précéder automatiquement celles des règles du second groupe, et vice-versa. En réalité, ce système de transition est issu de celui des `Construits`, qui a été tout d'abord simplifié pour prendre en compte le fait qu'un seul δ-type est référencé dans **type-ref**, puis complété pour intégrer la présence du nouveau champ **card** :

A.5	devient	A.15
A.6	devient	A.16
		A.17
		A.18
A.7	devient	A.19
A.8	devient	A.20
		A.26
A.9	devient	A.21
A.10	devient	A.22
A.11	devient	A.23
		A.24
		A.25
A.12	devient	inutile
A.13	devient	A.27
A.14	devient	A.28
		A.29
		A.30

Le tableau ci-dessus illustre la correspondance des règles entre les deux systèmes de transition élaborés dans le cas des construits n-aires d'une part, et unaires d'autre part (multivalués). Toutes les règles du premier trouvent une règle équivalente dans le second, sauf A.12 puisqu'il n'y a plus qu'un seul type référencé par le constructeur. Les règles du second système, qui n'ont pas d'équivalence directe avec une règle parmi celles du premier système, sont cependant placées dans les cases des règles avec lesquelles elles partagent leur objectif principal.

Les modifications apportées au système des `Construits` ne remettent pas en cause la nature des règles concernées, et les règles ajoutées ici (parce qu'il fallait tenir compte de l'intervalle de cardinalités autorisées) sont écrites selon le même principe d'atteinte des objectifs :

- quand les nouvelles règles s'attachent à éliminer des incohérences et des redondances (règles A.17, A.18, et A.24 et A.25), elles ne font qu'éliminer des valeurs ;
- quand les nouvelles règles s'attachent à cerner au mieux la finitude (ou l'infinitude) de l'ensemble de valeurs dénoté (règles A.26, A.29 et A.30), elles respectent les principes établis dans le cas des construits n-aires : minimisation des champs secondaires en cas d'énumération possible.
- quand des règles existantes pour les construits n-aires sont modifiées dans le cas des multivalués (règles A.15 et A.16, A.19 à A.23, A.27 et A.28), elles ne changent rien à leur but initial, se contentant d'adapter la transition à la présence d'un intervalle de cardinalités, et au fait qu'un seul δ-type est référencé.

Il est toutefois important de noter que la règle A.12 n'a pas d'équivalence dans le cas des multivalués, car il n'est intrinsèquement pas possible de minimiser le δ-type référencé (de même que l'intervalle de cardinalités) lorsque ce dernier est infini, et dans le cas d'un δ-type de cardinalité finie, son énumération est privilégiée.

Terminaison et confluence

Puisque le système de transition dans le cas des multivalués est directement issu de celui élaboré dans le cas des construits n-aires, comme nous l'avons expliqué dans la section précédente, il n'est pas pertinent d'imposer au lecteur, même intéressé, les preuves de terminaison et de confluence du dernier système de transition établi. En effet, ces preuves, très similaires aux précédentes, n'amèneraient aucun nouvel indice quant à la compréhension du système, dans la mesure où ces deux systèmes sont comparables dans leurs objectifs et leurs méthodes. De ce fait, de même que la normalisation des `Construits`, le système de transition correspondant à la normalisation des `Multivalués` termine et conflue.

Toutefois, il est intéressant d'observer en quoi les formes terminales, obtenues lorsque plus aucune règle du système ne peut s'appliquer, correspondent effectivement à des formes *normales*. À vrai dire, il s'agit d'une illustration du rôle secondaire que joue l'intervalle de cardinalités (au même titre que le type référencé) lorsque le δ-type dénote un ensemble fini de valeurs. Étudions donc ci-dessous les quatre formes normales qui peuvent être obtenues.

1. $FN_1 = \langle Multi; [\chi(\delta t); \text{TOUT}; \emptyset; [a;b]] \rangle$, qui dénote $\{v \in \|\chi\|^{\xi}(\|\delta t\|^{\xi}) \mid Nb(v) \in \|[a;b]\|^{\xi}\}$. Cet ensemble dénoté est forcément infini : si δt est de cardinalité finie, et si l'intervalle de cardinalités est aussi fini, – donc l'ensemble dénoté est fini malgré **domaine** à TOUT – (sauf si l'intervalle de cardinalités est vide, ce qui mène à l'application de la règle A.24 ou A.25 pour réécriture vers le δ-type inconsistant comme il se doit), la règle A.27 s'applique : la réécriture n'est pas terminée. Donc FN_1 est dédiée à la représentation d'ensembles infinis.

2. $FN_2 = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a; b]]\rangle$, qui dénote $\{v_1, \cdots, v_n\}$. L'ensemble dénoté ici est forcément fini et toutes les valeurs de l'énumération sont pertinentes, sinon des règles du premier groupe pourraient s'appliquer. En outre, la règle A.28 (resp. les règles A.29 et A.30) vise à minimiser le type référencé (resp. la cardinalité) à partir de l'énumération. À partir de ce constat, on montre aisément par l'absurde que deux δ-types dénotant le même ensemble fini de valeurs sont réécrits vers FN_2, qui est la seule à représenter un ensemble fini, et les deux réécritures de ces δ-types sont syntaxiquement égales.

3. $FN_3 = \langle Multi; [\chi(\delta t); \text{TOUT}; v'_1, \cdots, v'_p; [a; b]]\rangle$, qui dénote

$$\{v \in \|\chi\|^{\xi}(\|\delta t\|^{\xi}) \mid Nb(v) \in \|[a; b]\|^{\xi}\} \backslash \{v'_1, \cdots, v'_p\}$$

   Cet ensemble de valeurs dénoté est obligatoirement infini :

   - s'il était vide, cela signifierait que soit δt est inconsistant (règle A.23), soit que la cardinalité est vide (règle A.24 ou A.25) ;
   - s'il était fini, cela signifierait que, dans le même temps, δt est de cardinalité finie et que $b \neq +inf$, d'où l'application de la règle A.27. Notons toutefois que dans le cas des ensembles, il est procédé systématiquement à la réécriture de l'intervalle de cardinalités lorsque celui ci est infini alors que la cardinalité de δt est finie (règle A.26).

   Par ailleurs, les règles du premier groupe garantissent que l'énumération de **comp-dom** ne contient pas d'incohérence avec les champs **type-ref** et **card**. Dans ce cas de figure, il n'est pas effectué de minimisation de **type-ref** ni de **card**, car si c'était possible, cela signifierait que le δt serait de cardinalité finie, la règle A.27 s'appliquant alors. Donc FN_3 est réservée à l'expression d'ensembles de valeurs infinis. Enfin, si l'on remarque que les δ-types réécrits vers FN_3 n'ont jamais vu leur champs **type-ref** et **card** modifiés, il est alors aisé de montrer que FN_3 et FN_1 ne peuvent jamais dénoter le même ensemble infini de valeurs (car dans le cas d'un δt de cardinalité infinie comme c'est le cas ici, il ne peut y avoir d'énumération de valeurs interdites qui soit suffisamment exhaustive pour minimiser le δt : à partir du moment où le δ-type a été créé avec des valeurs interdites, soit elles sont toutes incohérentes par rapport à δt et donc s'éliminent grâce à la règle A.16, soit elles ne seront jamais éliminées de **comp-dom** par réduction de **type-ref**). Donc FN_3 est bien une forme normale pour l'expression d'ensembles de valeurs infinis.

4. $FN_4 = \perp$, qui dénote $\emptyset$. Nous avons vu que les trois formes précédentes ne dénotent jamais l'ensemble vide, donc d'après la complétude (*cf.* section suivante) du système, FN_4 est la seule forme possible pour exprimer l'ensemble vide.

Pour récapituler, disons qu'il n'existe qu'une forme d'expression des ensembles de valeurs finis non vides (FN_2), une seule forme d'expression d'un ensemble vide (FN_4), mais deux formes d'expressions d'ensembles de valeurs infinis (FN_1 et FN_3), qui pourtant ne dénoteront jamais d'ensembles égaux.

Complétude et correction

Pour les mêmes raisons que précédemment, dans la mesure où les preuves de correction et de complétude n'apporteraient aucun élément nouveau quant à la compréhension du système de transition, nous ne les présentons pas dans ce document. Mais nous garantissons la complétude et la correction de la normalisation des `Multivalués` au même titre que la normalisation des `Construits`.

Nous nous contentons de prouver la correction des règles nouvellement introduites : nous montrons que chacune de ces règles préserve l'ensemble de valeurs dénoté par le δ-type réécrit (tableau A.3).

Théorie et pratique de la normalisation des `Construits`

Nous avons présenté dans cette section sur les `Construits` en général, deux systèmes de transition possédant toutes les propriétés requises pour garantir la fiabilité de la normalisation.

Pourtant, cette fiabilité est toute relative, dans la mesure où en pratique, les principes que nous avons adopté ne sont pas toujours à même d'être appliqués. En effet, nous procédons, en théorie, à l'énumération systématique de l'application des construits lorsque la cardinalité de l'ensemble dénoté est connue pour être finie (ceci est toujours calculable à partir de la cardinalité des δ-types référencés dans **type-ref** et éventuellement à partir de la cardinalité dans le cas des multivalués). Pourtant, en pratique, les limites de la machine interdisent de telles énumérations lorsqu'un seuil est dépassé.

De ce fait, dans l'implémentation de la normalisation, nous avons fixé ce seuil, plus ou moins arbitrairement. Mais ce qu'il faut en retenir, c'est que du fait de ces limitations, il peut arriver que deux δ-types dénotant un même ensemble fini de valeurs soient représentés sous deux formes différentes à l'issue de la normalisation. Plus précisément, les deux seront représentés sous deux formes infinies différentes (l'un sous FN_1, l'autre sous FN_3).

On constate donc que le caractère fini de la mémoire d'un ordinateur nous amène à relativiser la confluence de nos systèmes de transitions. Il s'agit là, encore une fois, d'une illustration de ce fossé qui persiste entre la machine de Turing – et son ruban infini – et la machine à calculer dotée d'une pile mémorisante malheureusement bornée.

La normalisation dans son ensemble

Dans la mesure où, quel que soit le C-type dont est issu un δ-type à normaliser, la normalisation mise en œuvre est assurée (complétude), est correcte vis-à-vis des ensembles de valeurs dénotés, termine et conflue vers une forme obligatoirement normale, nous sommes en droit d'affirmer que la normalisation dans son ensemble est un processus *théoriquement* fiable, même si, dans la pratique, la confluence est remise en cause dans certains cas – lorsque l'on manipule des constructeurs de données.

A.5	$t = \langle CS; [T_c; v + V; d_2]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap (\{v\} \cup \Vert V\Vert^\xi))\backslash\Vert d_2\Vert^\xi$ or $v \notin \Vert T_c\Vert^\xi$ car $e_j \notin \Vert t_j\Vert^\xi$ donc $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert V\Vert^\xi)\backslash\Vert d_2\Vert^\xi$	$t = \langle CS; [T_c; V; d_2]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert V\Vert^\xi)\backslash\Vert d_2\Vert^\xi$
A.6	$t = \langle CS; [T_c; d_1; v' + V']\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash(\{v'\} \cup \Vert V'\Vert^\xi)$ or $v' \notin \Vert T_c\Vert^\xi$ car $e'_j \notin \Vert t_j\Vert^\xi$ donc $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert V'\Vert^\xi$	$t = \langle CS; [T_c; d_1; V']\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert V'\Vert^\xi$
A.7	$t = \langle CS; [T_c; v + V; v + V']\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap (\{v\} \cup \Vert V\Vert^\xi))\backslash(\{v\} \cup \Vert V'\Vert^\xi)$ donc $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert V\Vert^\xi)\backslash\Vert V'\Vert^\xi$	$t = \langle CS; [T_c; V; V']\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert V\Vert^\xi)\backslash\Vert V'\Vert^\xi$
A.8	$t = \langle CS; [T_c; v_1, \cdots, v_n; v'_1, \cdots, v'_p]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_n\})\backslash\{v'_1, \cdots, v'_p\}$ or $\forall i, j, v_i \neq v'_j$ donc $\Vert t\Vert^\xi = \Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$	$t = \langle CS; [v_1, \cdots, v_n; \emptyset]\rangle$ $\Vert t\Vert^\xi = \Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$
A.9	$t = \langle CS; [T_c; \emptyset; d_2]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert\emptyset\Vert^\xi)\backslash\Vert d_2\Vert^\xi$ donc $\Vert t\Vert^\xi = \emptyset$	$t = \bot$ $\Vert t\Vert^\xi = \emptyset$
A.10	$t = \langle CS; [T_c; d_1; \text{TOUT}]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert\text{TOUT}\Vert^\xi$ donc $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert T_c\Vert^\xi = \emptyset$	$t = \bot$ $\Vert t\Vert^\xi = \emptyset$
A.11	$t = \langle CS; [t_1\chi_1 \cdots \bot \cdots \chi_{k-1}t_k; d_1; d_2]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert d_2\Vert^\xi$ donc $\Vert t\Vert^\xi = (\Vert\bot\Vert^\xi \cap \Vert d_1\Vert^\xi)\backslash\Vert d_2\Vert^\xi = \emptyset$	$t = \bot$ $\Vert t\Vert^\xi = \emptyset$
A.12	$t = \langle CS; [t_1\chi_1 \cdots t_j \cdots \chi_{k-1}t_k; \text{TOUT}; v'_1, \cdots, v'_p]\rangle$ $\Vert t\Vert^\xi = \Vert t_1\chi_1 \cdots t_j \cdots \chi_{k-1}t_k\Vert^\xi\backslash\{v'_1, \cdots, v'_p\}$ or $m = \frac{Card(\Vert t_1\Vert^\xi) * \cdots * Card(\Vert t_k\Vert^\xi)}{Card(\Vert t_j\Vert^\xi)} \leq p$ et $\exists e \in \Vert t_j\Vert^\xi$ t.q. $\forall i \leq m, e'_{ji} = e$ donc $\forall v \in \Vert T_c\Vert^\xi$ t.q. $e_j = e, v \in \{v'_1, \cdots, v'_m\}$ donc $\Vert T_c\Vert^\xi = \Vert t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k\Vert^\xi$ or $\forall T_j, \Vert NormalForm_{T_j}(\text{domaine} \longleftarrow \{e\})\Vert^\xi = \{e\}$ donc $\Vert t'_j\Vert^\xi = \Vert t_j\Vert^\xi\backslash\{e\}$ alors : $\Vert t\Vert^\xi = (\Vert t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k\Vert^\xi)\backslash\{v'_1, \cdots, v'_p\}$ or $\forall i \in [1; m], e'_{ji} = e \notin \Vert t'_j\Vert^\xi$ donc : $\Vert t\Vert^\xi = \Vert t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k\Vert^\xi\backslash\{v'_{m+1}, \cdots, v'_p\}$ donc, avec $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ $\Vert t\Vert^\xi = \Vert T'_c\Vert^\xi\backslash\{v'_{m+1}, \cdots, v'_p\}$	$t = \langle CS; [T'_c; \text{TOUT}; v'_{m+1}, \cdots, v'_p]\rangle$ avec $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ $\Vert t\Vert^\xi = \Vert T'_c\Vert^\xi\backslash\{v'_{m+1}, \cdots, v'_p\}$
A.13	$t = \langle CS; [T_c; \text{TOUT}; d_2]\rangle$ donc $\Vert t\Vert^\xi = \Vert T_c\Vert^\xi\backslash\Vert d_2\Vert^\xi$, or $Card(T_c) = m$ donc $\Vert T_c\Vert^\xi = \{v_1, \cdots, v_m\}$ donc $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_m\})\backslash\Vert d_2\Vert^\xi$	$t = \langle CS; [T_c; v_1, \cdots, v_m; d_2]\rangle$ $\Vert t\Vert^\xi = (\Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_m\})\backslash\Vert d_2\Vert^\xi$
A.14	$t = \langle CS; [t_1\chi_1 \cdots \chi_{k-1}t_k; v_1, \cdots, v_n; \emptyset]\rangle$ avec $T_c = t_1\chi_1 \cdots t_j \cdots \chi_{k-1}t_k$ $\Vert t\Vert^\xi = \Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$ or $\forall T_j, \Vert NF_{T_j}(\text{domaine} \leftarrow \{e_{j1}, \cdots, e_{jn}\})\Vert^\xi$ $= \{e_{j1}, \cdots, e_{jn}\}$ donc $\Vert t'_j\Vert^\xi = \{e_{j1}, \cdots, e_{jn}\}$ et $t'_j <_\tau t_j \Leftrightarrow \Vert t'_j\Vert^\xi \subset \Vert t_j\Vert^\xi$ on a $\forall e \in \Vert t_j\Vert^\xi\backslash\Vert t'_j\Vert^\xi : e \neq e_{ji}, \forall i \in [1; n]$ donc $\Vert t_j\Vert^\xi \cap \{e_{j1}, \cdots, e_{jn}\} = \Vert t'_j\Vert^\xi \cap \{e_{j1}, \cdots, e_{jn}\}$ donc si $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ alors $\Vert T_c\Vert^\xi \cap \{v_1, \cdots, v_n\} = \Vert T'_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$ donc $\Vert t\Vert^\xi = \Vert T'_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$	$t = \langle CS; [t_1\chi_1 \cdots t'j \cdots \chi_{k-1}t_k; v_1, \cdots, v_n; \emptyset]\rangle$ avec $T'_c = t_1\chi_1 \cdots t'_j \cdots \chi_{k-1}t_k$ $\Vert t\Vert^\xi = \Vert T'_c\Vert^\xi \cap \{v_1, \cdots, v_n\}$

TAB. A.2 - : Correction de la normalisation des `Construits`.

A.17	$t = \langle Multi; [\chi(\delta t); v_1 + V; d_2; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap (\{v_1\} \cup \|V\|^\xi)\backslash\|d_2\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$ or $Nb(v_1) \notin [a;b]$ donc : $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \|V\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$	$t\langle Multi; [\chi(\delta t); V; d_1; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \|V\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$
A.18	$t = \langle Multi; [\chi(\delta t); d_1; v' + V'; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash(\{v'\} \cup \|V'\|^\xi) \mid Nb(v) \in \|[a;b]\|^\xi\}$ or $Nb(v') \notin [a;b]$ donc $v' \notin \{v \in \|\chi(\delta t)\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$ donc $\|t\|^\xi = \{v \in \|\chi(\delta t) \cap \|d_1\|^\xi\backslash\|V\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$	$t\langle Multi; [\chi(\delta t); d_1; V'; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t) \cap \|d_1\|^\xi\backslash\|V\|^\xi \mid Nb(v) \in \|[a;b]\|^\xi\}$
A.24	$t = \langle Multi; [\chi(\delta t); d_1; d_2; \emptyset]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in \emptyset\}$ donc $\|t\|^\xi = \emptyset$	$t = \bot$ $\|t\|^\xi = \emptyset$
A.25	$t = \langle Multi; [\chi(\delta t); d_1; d_2; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in [a;b]\}$ or $a > b$ donc $[a;b] = \emptyset$ donc $\|t\|^\xi = \emptyset$	$t = \bot$ $\|t\|^\xi = \emptyset$
A.26	$t = \langle Ensemble; [\mathrm{Ens}(\delta t); d_1; d_2; [a;+inf]]\rangle$ $\|t\|^\xi = \{v \in \|\mathrm{Ens}(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \geq a\}$ or $Card(\delta t) = m$ donc $\nexists v \in \mathrm{Ens}(\delta t) : Nb(v) > m$ donc $\|t\|^\xi = \{v \in \|\mathrm{Ens}(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in [a;m]\}$	$t = \langle Ensemble; [\mathrm{Ens}(\delta t); d_1; d_2; [a;m]]\rangle$ avec $m = Card(\delta t)$ $\|t\|^\xi = \{v \in \|\mathrm{Ens}(\delta t)\|^\xi \cap \|d_1\|^\xi\backslash\|d_2\|^\xi \mid Nb(v) \in [a;m]\}$
A.29	$t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \{v_1, \cdots, v_n\} \mid Nb(v) \in \|[a;b]\|^\xi\}$ or $\nexists i \in [1;n] : Nb(v_i) = a$ donc $\forall i \in [1;n], \{v_i \mid Nb(v_i\|^\xi \in \|[a;b]\|^\xi\} = \{v_i \mid Nb(v_i) \in \|[a+1;b]\|^\xi\}$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \{v_1, \cdots, v_n\} \mid Nb(v) \in \|[a+1;b]\|^\xi\}$	$t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a+1;b]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \{v_1, \cdots, v_n\} \mid Nb(v) \in \|[a+1;b]\|^\xi\}$
A.30	$t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a;b]]\rangle$ or $\forall i \in [1;n], Nb(v_i) \leq m < b$ donc $\forall i \in [1;n], \{v_i \mid Nb(v_i\|^\xi \in \|[a;b]\|^\xi\} = \{v_i \mid Nb(v_i) \in \|[a;m]\|^\xi\}$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \{v_1, \cdots, v_n\} \mid Nb(v) \in \|[a;m]\|^\xi\}$	$t = \langle Multi; [\chi(\delta t); v_1, \cdots, v_n; \emptyset; [a;m]]\rangle$ $\|t\|^\xi = \{v \in \|\chi(\delta t)\|^\xi \cap \{v_1, \cdots, v_n\} \mid Nb(v) \in \|[a;m]\|^\xi\}$

TAB.A.3: Correction des règles nouvellement introduites dans le cas de la normalisation des δ-types issus de `Multivalués`.

Annexe B

Preuves

Cette annexe contient les preuves de correction et complétude du typage et du sous-typage, au regard des interprétations en intension ; les preuves des diverses propriétés, lemmes et théorèmes enoncés dans le document, sont aussi données.

Correction et complétude du typage brut

Le typage des classes et des concepts est directement issu du typage des attributs, dans la mesure où la structure de la description d'une classe est identique à celle d'un type *record* au sens de L. Cardelli. De ce fait, seules les preuves de correction et de complétude du typage des attributs sont nécessaires. Nous prouvons donc le théorème 2, page 143.

Correction du typage brut des attributs

D'une part, nous constatons que la procédure de typage brut, donnée dans la section 4.4.2 (page 140), considère l'intersection des informations associées aux descripteurs, lors de l'élaboration du δ-type à normaliser : l'ensemble de valeurs dénoté par l'attribut est ici élaboré comme l'*intersection des ensembles de valeurs dénotés par chacun des descripteurs appliqués sur l'ensemble de valeurs dénoté par le type principal.* D'autre part, l'interprétation informelle qui a été faite des descripteurs, lors de la présentation de TROPES (page 55), précise que *les différents descripteurs peuvent être combinés par conjonction* ; on retrouve alors le principe d'intersection des ensemble de valeurs dénotés par les descripteurs informés pour obtenir l'ensemble de valeurs final.

Ensuite, la comparaison deux à deux de chacun des descripteurs avec la façon dont il est interprété lors de la procédure de typage (phase 4 et 5) confirme la correction de cette interprétation, au regard de l'interprétation en intension des descripteurs (donnée informellement page 55). Par ailleurs, et lorsqu'une erreur n'est pas générée par la procédure de typage), le δ-type obtenu est normalisé, et nous avons montré, dans l'annexe précédente, la correction, complétude et confluence de la normalisation.

Nous admettons donc, sans fournir de preuve formelle (qui serait peu pertinente vue la simplicité de son énoncé informel), que la procédure de typage est correcte au regard de l'interprétation intensionnelle des spécifications des attributs. Le typage des classes, concepts et instances revenant à la simple agrégation des types de leurs attributs, nous le considérons aussi correct.

Complétude du typage brut des attributs

La complétude du typage brut d'un attribut est immédiate. En effet, les cas de figures non traités lors de la phase 5 du typage sont en réalité des situations d'échec (par exemple, un descripteur non adapté à un type principal ou à un constructeur) qui ont mené à la production d'une erreur lors de la phase 4. Tous les autres cas non traités (lors de la phase 2), sont eux-aussi des erreurs de spécifications de types (récursivité dans les déclarations ou application d'un constructeur de données à des valeurs d'un autre constructeur). À part ces deux types de situations, détectés lors des phases 2 et 4 et gérés comme des erreurs (donc menant au δ-type inconsistant $\perp$), tous les cas sont traités et le typage est ainsi complet.

Correction et complétude du δ-sous-typage

Nous montrons la correction et la complétude du δ-sous-typage présenté page 3.4.3. Autrement dit, nous montrons que la procédure de vérification du δ-sous-typage assure que

$$\forall t_1, t_2, \|t_1\|^\xi \subseteq \|t_2\|^\xi \iff t_1 \leq_\tau t_2$$

Nous prouvons ce résultat pour chacune des définitions du δ-sous-typage correspondant à chaque groupe de C-types partageant le même mode de représentation de leurs δ-types. Ces preuves s'appuient sur la normalisation des δ-types.

Univers, Types de base, Ordonnés Dans ce cas, le δ-type est exprimé par un seul champ, le domaine explicite de valeurs : $t = \langle TU; D\rangle$.

$$\begin{aligned} t_1 \leq_\tau t_2 &\Leftrightarrow \langle TU; D_1\rangle \leq_\tau \langle TU; D_2\rangle \\ &\Leftrightarrow \|D_1\|^\xi \subseteq \|D_2\|^\xi \\ &\Leftrightarrow D_1 \subseteq_{TU} D_2 \end{aligned}$$

Construits et Records Dans ce cas, le δ-type est exprimé par trois champs, le type des éléments (**type-ref**), le domaine de valeurs explicites et le domaine de valeurs interdites. Compte-tenu de la normalisation de ces δ-types, nous devons considérer deux cas de tests : le sur-type dénote un ensemble fini de valeurs (il est sous FN_2) ou un ensemble infini (il est sous FN_1 ou FN_3). Dans le premier cas, le seul sous-type possible est un autre δ-type fini ; dans le second cas, ce peut être un δ-type fini ou infini.

$$\begin{aligned} t_1 \leq_\tau t_2 &\Leftrightarrow \langle TC; [t'_1; D_1; C_1]\rangle \leq_\tau \langle TC; [t'_2; D_2; C_2]\rangle \\ &\Leftrightarrow \|t'_1\|^\xi \cap \|D_1\|^\xi \backslash \|C_1\|^\xi \subseteq \|t'_2\|^\xi \cap \|D_2\|^\xi \backslash \|C_2\|^\xi \\ &\Leftrightarrow \left\{ \begin{array}{l} \|t'_{1i}\|^\xi \subseteq \|t'_{2i}\|^\xi, i \in [1;k],\ \text{si } t'_j = t'_{j1}\chi_1 \cdots \chi_{k-1} t'_{jk}, j \in [1;2] \\ \text{et } \|D_1\|^\xi \subseteq \|D_2\|^\xi \\ \text{et } \|C_2\|^\xi \subseteq \|C_1\|^\xi \text{ ou } (\forall v \in \|C_2\|^\xi, v \notin \|D_1\|^\xi \text{ ou } v \notin \|t'_1\|^\xi \text{ ou } v \in \|C_1\|^\xi) \end{array} \right. \\ &\Leftrightarrow \left\{ \begin{array}{l} \forall i \in [1;k], t'_{1i} \leq_\tau t'_{2i} \\ \text{et } D_1 \subseteq_{TC} D_2 \\ \text{et } C_2 \subseteq_{TC} C_1 \text{ ou } (\forall v \in C_2, v \notin D_1 \text{ ou } v \notin t'_1 \text{ ou } v \in C_1 \end{array} \right. \end{aligned}$$

La troisième équivalence est due aux propriétés de la normalisation des `Construits`, qui mène à trois formes normales susceptibles d'être comparées entre elles. Les règles de normalisation assurent les faits suivants :

1. si t est sous FN_1, $t = \langle TC; [t'; \text{TOUT}; \emptyset]\rangle$, alors $\|t\|^\xi = \|t'\|^\xi$

2. si t est sous FN_2, $t = \langle TC; [t'; v_1, \cdots, v_n; \emptyset]\rangle$, alors $\|t\|^\xi = \|D\|^\xi$ et $\not\exists t''$ t.q. $\|d\|^\xi \subseteq \|t''\|^\xi \subseteq \|t'\|^\xi$ (règles A.13 et A.14).
3. si t est sous FN_3, $t = \langle TC; [t'; \text{TOUT}; v_1, \cdots v_p]\rangle$, alors $\|t\|^\xi = \|t'\|^\xi \backslash \{v_1, \cdots, v_p\}$ et $\not\exists t''$ t.q. $\|t'\|^\xi \backslash \{v_1, \cdots, v_p\} \subseteq \|t''\|^\xi \backslash \{v_1, \cdots, v_h\} \subseteq \|t'\|^\xi, h \in [1; p]$ (règle A.12).

Nous examinons dans le tableau B.1, pour chaque cas de comparaisons possibles, la validité de cette équivalence, en faisant appel aux propriétés sous-jacentes aux faits énoncés ci-dessus. Ce tableau nous prouve la validité du δ-sous-typage tel qu'il est conçu sur les δ-types issus de `Construits`. Les δ-types issus de `Record` obéissent aux mêmes règles dans la mesure où il n'est pas introduit de nouveau champ de représentation.

Multi-valués Dans ce cas, il s'agit de prendre en compte les trois mêmes champs que dans le cas des `Construits`, et de considérer en plus le champ indiquant l'intervalle de cardinalité. Lors de la normalisation des multivalués, le champ **card** est en fait considéré comme un indicatif supplémentaire concernant le caractère fini des δ-types. Les conditions nécessaires et suffisantes de δ-sous-typage entre deux δ-types issus de `Multivalués` ont donc été élaborées de façon similaire à celles des `Construits`, et la preuve de la correction et complétude de ce δ-sous-typage est donc similaire ; c'est pourquoi nous ne la présentons pas dans ce document.

$$
\begin{aligned}
t_1 \leq_\tau t_2 &\Leftrightarrow \langle Multi; \chi(t'_1); D_1; C_1; c_1]\rangle \leq_\tau \langle Multi; [\chi(t'_2); D_2; C_2; c_2]\rangle \\
&\Leftrightarrow \begin{array}{l} \{v \in \|\chi(t'_1)\|^\xi \cap \|D_1\|^\xi \backslash \|C_1\|^\xi \mid Nb(v) \in c_1\} \\ \quad \subseteq \{v' \in \|\chi(t'_2)\|^\xi \cap \|D_2\|^\xi \backslash \|C_2\|^\xi \mid Nb(v') \in c_2\} \end{array} \\
&\Leftrightarrow \left\{ \begin{array}{l} \|t'_1\|^\xi \subseteq \|t'_2\|^\xi \\ \text{et } \|D_1\|^\xi \subseteq \|D_2\|^\xi \\ \text{et } \|C_2\|^\xi \subseteq \|C_1\|^\xi \text{ ou } \forall v \in \|C_2\|^\xi \left\{ \begin{array}{l} v \notin \|D_1\|^\xi \\ \text{ou } v \notin \|t'_1\|^\xi \\ \text{ou } v \in \|C_1\|^\xi \\ \text{ou } Nb(v) \notin \|c_1\|^\xi) \end{array} \right. \\ \text{et } \|c_1\|^\xi \subseteq \|c_2\|^\xi \end{array} \right. \\
&\Leftrightarrow \left\{ \begin{array}{l} t'_1 \leq_\tau t'_2 \\ \text{et } D_1 \subseteq_{Multi} D_2 \\ \text{et } C_2 \subseteq_{Multi} C_1 \text{ ou } (\forall v \in C_2, v \notin D_1 \text{ ou } v \notin t'_1 \text{ ou } v \in C_1 \text{ ou } Nb(v) \in c_1) \end{array} \right.
\end{aligned}
$$

Énumérés Dans le cas des δ-types issus de `Énumérés`, enfin, deux champs de représentation sont à prendre en considération : $t = \langle Enumeres; D; C\rangle$.

$$
\begin{aligned}
t_1 \leq_\tau t_2 &\Leftrightarrow \langle TU; [D_1; C_1]\rangle \leq_\tau \langle TU; [D_2; C_2]\rangle \\
&\Leftrightarrow \|D_1\|^\xi \backslash \|C_1\|^\xi \subseteq \|D_2\|^\xi \backslash \|C_2\|^\xi \\
&\Leftrightarrow \left\{ \begin{array}{l} \|D_1\|^\xi \subseteq \|D_2\|^\xi \\ \text{et } \|C_2\|^\xi \subseteq \|C_1\|^\xi \end{array} \right. \\
&\Leftrightarrow \left\{ \begin{array}{l} D_1 \subseteq_{Enum} D_2 \\ \text{et } C_2 \subseteq_{Enum} C_1 \end{array} \right.
\end{aligned}
$$

La troisième équivalence est valide, dans la mesure où, grâce à la normalisation des δ-types issus de `Énumérés`, le domaine de valeurs et le domaine complémentaire ne sont jamais énumérés en même temps.

Preuves du chapitre 3

Propriété 1 *La relation de δ-sous-typage, sur un C-type T, définit un ordre partiel complet sur $\Delta(T)$.*

	$[t'_1;\text{TOUT};\emptyset]$	$[t'_1;v_1,\cdots,v_n;\emptyset]$	$[t'_1;\text{TOUT};v_1,\cdots,v_p]$
$[t'_2;\text{TOUT};\emptyset]$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\subseteq\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et TOUT}\subseteq\text{TOUT}\\ \text{et }\emptyset\subseteq\emptyset\end{array}\right\}\Box$	$\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}\backslash\emptyset$ $\subseteq\lVert t'_2\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}\subseteq\lVert t'_2\rVert^\xi$ donc d'après le fait 2 $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\{v_1,\cdots,v_n\}\subseteq\lVert t'_2\rVert^\xi\end{array}\right\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\{v_1,\cdots,v_n\}\subseteq\text{TOUT}\\ \emptyset\subseteq\emptyset\end{array}\right\}\Box$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v_1,\cdots,v_p\}$ $\subseteq\lVert t'_2\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\backslash\{v_1,\cdots,v_p\}\subseteq\lVert t'_2\rVert^\xi$ donc d'après le fait 3 $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\emptyset\subseteq\{v_1,\cdots,v_p\}\end{array}\right\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et TOUT}\subseteq\text{TOUT}\\ \emptyset\subseteq\{v_1,\cdots,v_p\}\end{array}\right\}\Box$
$[t'_2;v'_1,\cdots,v'_m;\emptyset]$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}$ donc d'après le fait 2 $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\lVert\text{TOUT}\rVert^\xi\subseteq\{v'_1,\cdots,v'_m\}\\ \text{et }\cdots\end{array}\right\}$ (impossibilité détectée) $\Box$	$\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}\backslash\emptyset$ $\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}$ $\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}$ donc d'après le fait 2 $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\{v_1,\cdots v_n\}\subseteq\{v'_1,\cdots,v'_m\}\end{array}\right\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\{v_1,\cdots,v_n\}\subseteq\{v'_1,\cdots,v'_m\}\\ \emptyset\subseteq\emptyset\end{array}\right\}\Box$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v_1,\cdots,v_p\}$ $\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}\backslash\emptyset$ $\Leftrightarrow\lVert t'_1\rVert^\xi\backslash\{v_1,\cdots,v_p\}$ $\subseteq\lVert t'_2\rVert^\xi\cap\{v'_1,\cdots,v'_m\}$ donc d'après le fait 2 $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et TOUT}\subseteq\{v'_1,\cdots,v'_m\}\\ \text{et }\cdots\end{array}\right\}$ (impossibilité détectée) $\Box$
$[t'_2;\text{TOUT};v'_1,\cdots,v'_q]$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\emptyset$ $\subseteq\lVert t'_2\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\forall i\in[1;q],v'_i\notin\lVert t'_1\rVert^\xi\end{array}\right\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et TOUT}\subseteq\text{TOUT}\\ \text{et }\forall i\in[1;q],v'_i\notin\lVert t'_1\rVert^\xi\end{array}\right\}\Box$	$\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}\backslash\emptyset$ $\subseteq\lVert t'_2\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\lVert t'_1\rVert^\xi\cap\{v_1,\cdots,v_n\}$ $\subseteq\lVert t'_2\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et }\{v_1,\cdots,v_n\}\subseteq\text{TOUT}\\ \text{et }\forall i\in[1;q],\\ \qquad\left\{\begin{array}{l}v'_i\notin\lVert t'_1\rVert^\xi\\ \text{ou }v'_i\notin\{v_1,\cdots,v_n\}\end{array}\right.\end{array}\right\}\Box$	$\lVert t'_1\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v_1,\cdots,v_p\}$ $\subseteq\lVert t'_2\rVert^\xi\cap\lVert\text{TOUT}\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\lVert t'_1\rVert^\xi\backslash\{v_1,\cdots,v_p\}$ $\subseteq\lVert t'_2\rVert^\xi\backslash\{v'_1,\cdots,v'_q\}$ $\Leftrightarrow\left\{\begin{array}{l}\lVert t'_1\rVert^\xi\subseteq\lVert t'_2\rVert^\xi\\ \text{et TOUT}\subseteq\text{TOUT}\\ \text{et }\forall i\in[1;q],\\ \qquad\left\{\begin{array}{l}v'_i\in\{v_1,\cdots,v_p\}\\ \text{ou }v'_i\notin\lVert t'_1\rVert^\xi\end{array}\right.\end{array}\right\}\Box$

TAB.B.1: Validité du δ-sous-typage entre δ-types issus de `Construits`: conditions nécessaires et suffisantes pour que $t_1 \leq_\tau t_2$, avec $t_1 = \langle TC; E_1(TC)\rangle$ et $t_2 = \langle TC; E_2(TC)\rangle$.

Preuve. Cette propriété est due à l'interprétation ensembliste du δ-sous-typage (définition 21). Soit T un C-type, et $\Delta(T)$ l'ensemble des δ-types associé à T. La réflexivité, la transitivité et l'anti-symétrie proviennent directement de l'interprétation ensembliste du δ-sous-typage. Cet ordre est complet car pour toute paire de δ-types dans $\Delta(T)$, il existe un plus petit δ-type maximum calculé par union des ensembles dénotés, donc unique, de même pour le plus grand δ-type minimum calculé à partir de l'intersection. Comme l'intersection et l'union de deux δ-types sont définies pour représenter strictement l'intersection et l'union ensemblistes, le δ-sous-typage est un ordre complet. □

Propriété 2 *$\forall T \in \mathcal{T}, \widehat{\mathcal{G}(T)}$ est un treillis complet.*

Preuve. ce résultat vient du fait que la relation de δ-sous-typage est un ordre partiel complet. □

Propriété 3 *Soient $T_1, T_2 \in \mathcal{T}$ tels que $T_2 \leq_C T_1$. Il existe un homomorphisme $h : \Delta(T_2) \rightarrow \Delta(T_1)$ tel que*

$$\forall t_1, t'_1 \in \Delta(T_1) \text{ tels que } t_1 \leq^{\delta}_{T_1} t'_1,$$
$$h(t_1) \in \Delta(T_2), h(t'_1) \in \Delta(T_2) \text{ et } h(t_1) \leq^{\delta}_{T_2} h(t'_1)$$

Preuve. Nous définissons h tel que $\|h(t_1)\|^{\xi} = \|t_1\|^{\xi} \backslash \{e \in \|t_1\|^{\xi} \mid e \notin \|T_2\|^{\xi}\}$. Ainsi, l'homomorphisme h peut être vu comme une projection. Puisque $h(t_1)$ doit être une expression ne dénotant que des éléments de T_2, $h(t_1)$ s'exprime par le mode de représentation de T_2. De même, $\|h(t'_1)\|^{\xi} = \|t'_1\|^{\xi} \backslash \{e \in \|t'_1\|^{\xi} \mid e \notin \|T_2\|^{\xi}\}$. Or $t_1 \leq^{\delta}_{T_1} t'_1$, donc $\|t_1\|^{\xi} \subseteq \|t'_1\|^{\xi}$, et donc $\{e \in \|t_1\|^{\xi} \mid e \notin \|T_2\|^{\xi}\} \subseteq \{e \in \|t'_1\|^{\xi} \mid e \notin \|T_2\|^{\xi}\}$. Par conséquent, $h(t_1) \subseteq h(t'_1)$, donc d'après le théorème 1, $h(t_1) \leq^{\delta}_{T_2} h(t'_1)$. □

Preuves du chapitre 4

Théorème 3 *Soit E une entité de représentation considérée par les deux procédures de typage (un attribut, une classe, un concept ou une instance). Si `Typing-b`$(E) \neq \perp$ alors :*

$$\texttt{Typing-b}(E) = \texttt{Typing-i}(E)$$

Preuve. Rappelons la différence entre ces deux typages, dans le cas d'un attribut : le premier récupère, avant le typage, les facettes de typage associées à l'attribut de même nom dans les sur-classes (appelons cet ensemble $F(sup(\tilde{a}))$), et joint cet ensemble de facettes récupérées à celui des facettes définies localement pour lui (appelons cet ensemble $F(\tilde{a})$) ; le second récupère le δ-type associé à l'attribut de même nom le plus proche dans la hiérarchie : ce δ-type correspond en fait au typage de $F(sup(\tilde{a}))$. Ainsi, la différence des deux typages est la suivante :

`Typing-i`$(a) =$ `Typing`$(F(sup(\tilde{a}))) \sqcap$ `Elaborate-type`$(F(\tilde{a}))$

`Typing-b`$(a) =$ `Elaborate-type`$(F(sup(\tilde{a})) \cup F(\tilde{a}))$

où $\cup$, ici, désigne en fait l'ajout de $F(\tilde{a})$, exprimé sous forme de couples $(< descr >_i < info >_i)$, dans la liste des couples de $F(sup(\tilde{a}))$. Or la phase d'intégration des couples $(< descr >_i < info >_i)$ (réécrits en syntaxe MetéO) est effectuée séquentiellement à partir de la liste des couples (phase 5). Or pour chaque intitulé de descripteur, il est trivial de voir que l'intégration de l'information contenue dans ce descripteur est réalisée selon un processus de réduction du domaine : chaque descripteur est une contribution à la réduction du domaine de valeurs exprimé par le δ-type. Plus qu'une réduction, il s'agit d'intersections des domaines de valeurs. Ainsi, soit t_i le δ-type obtenu après intégration du couple i, alors $t_{i+1} = t_i \cap (< descr >_i < info >_i)$. Si t est le type final obtenu, $t = \bigcap_{i=0}^{i=n} (< descr >_i < info >_i)$, avec $t_0 = \top_T$ si T est le C-type de création de t.

L'opération d'intersection étant associative, à partir du moment où les mêmes facettes sont considérées lors du typage et que le typage est correct et complet (ce que nous avons montré dans la première section de ce chapitre), alors les deux procédures de typage, brut et incrémental, sont équivalentes. □

Propriété 4 (Instanciation intensionnelle et appartenance à un type *record*)

$$\forall O = C \text{ } ou \text{ } K, I \in_I O \iff \textit{compute-value}(I) \in^{\delta}_{Record} \textit{ get-type}(O)$$

Preuve. Nous supposons que *compute-value*(I) associe à l'instance I la valeur *record* construite en associant à chaque attribut du concept, la valeur que l'instance donne à cet attribut ; dans le cas d'une valeur inconnue pour un attribut, ou d'un attribut non considéré par l'instance, l'étiquette n'apparaît pas dans la valeur *record* de l'instance. En ce sens, la longueur de la valeur *record* de l'instance est inférieure à celle du type *record* du concept.

Par confrontation des définitions 4 (page 65) et de la définition de l'appartenance d'une valeur à un δ-type *record* (page 112), en l'absence de récursivité et puisque le typage des attributs est correct est complet, il est immédiat de constater l'équivalence des deux tests effectués dans ces définitions :

- les champs **domaine** et **comp-dom** du type d'une classe (ou d'un concept) étant respectivement à *all* et *nothing*, seule l'appartenance d'une valeur *record* à un type *record* (agrégat) est testée,
- quelles que soient les longueurs des valeur et type *record*, seuls les champs en communs sont testés dans $\in^{\delta}_{Record}$: les valeurs inconnues sont ignorées, de même que les valeurs de l'instance allouées à des attributs que la classe ne définirait pas.

L'équivalence des deux définitions étant évidente, la propriété est montrée. $\square$

Propriété 5 (Spécialisation intensionnelle et δ-sous-typage entre *records*)

$$C_1 \leq_{\sigma,I} C_2 \iff \textit{get-type}(C_1) \leq^{\delta}_{Record} \textit{ get-type}(C_2)$$

Preuve. La preuve de cette propriété est similaire à la précédente : si nous reprenons, d'une part la définition du δ-sous-typage entre *records* (page 112) à laquelle nous éliminons les tests sur les champs **domaine** et **comp-dom** (qui sont respectivement à *all* et *nothing* pour les types de classes), et d'autre part la définition de la spécialisation intensionnelle (définition 6, page 65), nous constatons l'équivalence des deux, en l'absence de récursivité :

- les deux δ-types associés à la même étiquette dans les δ-types *records* associés aux deux classes, sont comparés au regard du δ-sous-typage : le δ-sous-typage étant correct et complet, ces vérifications sont fiables au regard de l'affinement des descriptions d'attributs ; cette comparaison est équivalente au test d'inclusion entre les domaines des attributs, du fait de la correction et de la complétude du δ-sous-typage au regard de l'inclusion des domaines.
- dans les deux définitions, il est fait état du fait que C_1 donne une définition à tous les attributs définis dans C_2 (dans la définition de la spécialisation intensionnelle, il s'agit de la condition $p \leq n$; dans la définition du δ-sous-typage entre δ-types *records*, il s'agit de la condition de terminaison de l'induction sur les agrégations de δ-types composant les champs **type-ref** de chacun des types de classes : la première des deux conditions d'arrêt aurorise une plus grande agrégation pour le sous-type).

Les deux définitions étant équivalentes, la propriété est montrée. $\square$

Propriété 6 (plongement) *L'arbre de spécialisation de tout point de vue d'un concept est* plongé *dans le treillis des δ-types* records *issus du typage de la base de connaissances dans laquelle est défini le concept.*

Preuve C'est une conséquence directe de la propriété 5. $\square$

Propriété 7 (Passerelles, INF et δ-sous-typage) $\forall C_1, \cdots, C_n, \forall C$,

$$\{C_1, \cdots, C_n\} \gg_I C \iff \{\texttt{get-type}(C_i)\}_{i=1..n} \gg_\tau \texttt{get-type}(C)$$

Preuve. Examinons la décomposition de la définition intensionnelle d'une passerelle, qui comporte deux conditions (définition 8, page 67) :

- la première condition s'assure du fait qu'une instance pourra toujours appartenir à la source d'une passerelle, et ceci en vérifiant qu'aucun domaine d'attribut n'est vidé :

  $$\bigcap_{i\in[1..n],a\in A(C_i)} \tilde{a}^i \neq \emptyset$$

  on s'assure ici que pour tout attribut qui apparaît dans plusieurs des classes sources, l'intersection des domaines de cet attribut donnés dans les classes sources le possédant n'est pas vide. Cette condition correspond exactement à l'une des deux énoncées dans la définition de $\gg_\tau$:

  $$T_p \neq \perp$$

  c'est-à-dire que le type résultant de l'intersection des types des classes sources n'est pas inconsistant (si un seul champ de T_p, correspond au type d'un des attributs, s'était vidé, la normalisation aurait réécrit T_p vers $\perp$, règle A.11, page 229). Notons qu'ici, on réalise l'intersection des types des classes, et non ps seulement des domaines des attributs communs : cela ne change rien au résultat, dans la mesure où lorsqu'un attribut n'est pas défini dans une classe, le type de cette classe alloue quand même à cet attribut un δ-type, celui donné pour l'attribut dans le concept, autrement dit le δ-type maximum du C-type concerné : ce δ-type ne modifie pas le résultat d'une intersection. L'application de cette condition portant sur l'intersection des domaines sources est ainsi équivalente dans les deux définitions.
- la seconde condition s'assure que toute valeur d'attribut autorisée conjointement par toutes les classes sources, le sera automatiquement par la classe destination ; cette condition s'énonce en imposant que le domaine de valeurs d'un attribut, autorisé conjointement par les classes sources, est inclus dans le domaine de valeurs autorisé par la classe destination, et ce pour tous les attributs qui apparaissent dans au moins une des classes sources, et dans la classe destination : $\forall i \in [1..q], \forall j \in [1..p]$ tel que C_i possède a_j :

  $$\begin{cases} \forall I \in_I d^i_j, I \in_I d_j & \text{si } a_j \text{ est un attribut complexe} \\ (\bigcap_{i,C_i \text{ possède } a_j} d^i_j \subseteq) d_j & \text{sinon} \end{cases}$$

  L'inclusion des domaines d'attributs étant représentée fidèlement par le γ-sous-typage sur les δ-types des attributs, nous retrouvons la condition équivalente dans la définition de $\gg_\tau$:

  $$T_p \leq_{Record} (\prod_{e\in \mathtt{Etiq}(T_p)} T)$$

  puisque le δ-sous-typage sur les records est fondé sur le γ-sous-typage sur les types des attributs : ici, le sous-typage est réduit à des tests de sous-typage portant sur les types des attributs apparaissant dans la source. La projection (qui est réalisée en réalité sur l'union des attributs apparaissant dans les classes sources) assure que la condition ne porte que sur des attributs risquant de remettre en cause l'existence d'une passerelle. Notons ici que si un attribut d'une classe source n'apparaît pas dans la destination, alors le type de cet attribut dans T (et donc dans la projection) est celui donné dans le concept pour cet attribut, soit $\top_{CT}$ si CT est le C-type concerné ; dans la mesure où il est demandé à T_p d'être un *sous-type* de T, les attributs manquants dans la classe destination, et pourtant représentés dans son type, ne changent rien au résultat de test de sous-typage. L'application de cette condition, qui porte sur l'inclusion du domaine conjoint des sources dans le domaine de la destination, est ainsi équivalente dans les deux définitions.

Les définitions de $\gg_\tau$ et $\gg_I$ portent sur des conditions traitées de façons équivalentes, donc ces définitions sont équivalentes, et la propriété est montrée. $\square$

Preuves du chapitre 5

Propriété 9 *Tout attribut a de toute super-classe proposée pour C dans spc_C existe aussi dans C, et le type de a dans C est un sous-type du type de a dans la super-classe :*

$$\forall C' \in spc_C, \forall a \in A(C'), \exists a \in A(C) \text{ et } t(a_C) \leq_\tau t(a_{C'})$$

Preuve La construction même de l'ensemble des super-classes possibles pour C relativement à un attribut a typé par C, soit sp, assure que seules les classes donnant pour a un sur-type de celui donné par C sont retenues, ainsi que les classes ne possédant pas a (algorithme de `search-slottype-position`, page 168). De plus, le traitement principal (page 167) réalise l'intersection de tous les sp_i déterminés pour chacun des attributs définis par C : ainsi, seules les classes du point de vue qui, à chaque attribut a de C donnent un sous-type de $t(a_C)$ ou ne le définissent pas, sont dans spc_C. Le sous-typage est donc garanti entre les types des super-classes proposées pour C et le type de C. Mais le traitement principal de la classification de C n'opère que sur l'ensemble des attributs de C. Donc à l'issue de ce traitement effectué par MetÉo, il peut demeurer des classes de spc_C possédant des attributs que ne définit pas C : ces classes sont éliminées lors de la première phase de post-traitement (phase 1, page 169). □

Propriété 10 *Tout attribut a de C existe aussi dans les sous-classes proposées pour C par ssc_C, et le type de a dans C est un sur-type du type de a dans les sous-classes proposées :*

$$\forall a \in A(C), \forall C' \in ssc_C, a \in A(C') \text{ et } t(a_{C'}) \leq_\tau t(a_C)$$

Preuve La construction même de l'ensemble des sous-classes possibles pour C relativement à un attribut de a typé par C, soit ss, assure que seules les classes donnant pour a un sous-type de celui donné par C sont retenues (algorithme de `search-slottype-position`, page 168). De plus, le traitement principal (page 167) réalise l'intersection de tous les ss_i déterminés pour chacun des attributs définis par C : ne sont donc gardées dans ssc_C que les classes qui possèdent tous les attributs de C à chacun desquels elles donnent un sous-type de celui donné par C. Le traitement principal de la classification de C n'opérant que sur l'ensemble des attributs de C, ssc_C ne peut contenir de classes qui ne possèdent pas tous les attributs de C ; en effet, lorsqu'une classe ne possède pas l'attribut a_i défini dans C, elle n'apparaît pas dans ssc_i, et trivialement n'apparaîtra pas dans l'intersection des ssc_i opérée par le traitement principal, donc n'apparaîtra pas dans la solution finale (les deux phases de post-traitement se contentant d'*enlever des solutions*). □

Lemme 1 (Correction de la spécialisation)

$$\forall C' \in spc_C, C \leq_{\sigma,I} C' \text{ et } \forall C'' \in ssc_C, C'' \leq_{\sigma,I} C$$

Preuve Ce lemme est trivial : il provient des deux propriétés précédentes (9 et 10) qui sont toutes deux équivalentes au regard de la définition de la spécialisation intensionnelle (définition 6, page 65) : héritage complet et inclusion des domaines sont assurés d'une sous-classe vers sa super-classe. □

Lemme 2 (Complétude de la spécialisation) *Le processus de classification d'une classe C par les types de ses attributs identifie toutes les classes et sous-classes possibles pour C, modulo la réduction transitive de la spécialisation, et ce avant que l'élimination de super-classes potentielles menant à la violation de l'exclusivité soit effectuée :*

$$\forall C', C' \leq_{\sigma,I} C \Longrightarrow C' \in ssc_C$$

$$\forall C'', C \leq_{\sigma,I} C'' \Longrightarrow C'' \in spc_C$$

Preuve Soit $C = \langle a_i : d_i \rangle_{i \in [1;n]}$. Nous montrons successivement les deux implications, en passant par leurs contraposées.

- Soit $C' \notin ssc_C$, montrons que alors que $C' \not\leq_{\sigma,I} C$. L'ensemble ssc_C, au départ, contient toutes les classes du point de vue dans lequel se déroule la classification. D'après le traitement principal et l'algorithme de `Search-slottype-position`, il y a deux raisons pour lesquelles C' a pu être retirée du ssc_C initial :
  1. $\exists i \in [1;n] : \tilde{a_i}(C') \not\subseteq \tilde{a_i}(C)$, déterminé par le sous-typage entre les types de ces deux définitions d'attributs. Mais dans ce cas, il y a violation de la définition de la spécialisation intensionnelle, qui exige que $\forall a_i \in A(C), \tilde{a_i}(C') \subseteq \tilde{a_i}(C)$, donc $C' \not\leq_{\sigma,I} C$;

2. $\exists i \in [1;n]$: $a_i \notin A(C')$ (`search-slottype-position` élimine alors C' de ss_i donc de la solution finale). Cette condition viole aussi la définition de la spécialisation intensionnelle (violation de l'héritage complet), donc $C' \not\leq_{\sigma,I} C$.

Nous avons donc montré, par sa contraposée, que

$$\forall C', C' \leq_{\sigma,I} C \Longrightarrow C' \in ssc_C$$

– Soit $C'' \notin spc_C$, montrons alors que $C \not\leq_{\sigma,I} C''$. Soit $C'' = \langle a_i : d_i \rangle_{i \in [1;p]}$. L'ensemble ssc_C, au départ, contient toutes les classes du point de vue dans lequel se déroule la classification. D'après le traitement principal et l'algorithme de `Search-slottype-position`, il y a deux raisons pour lesquelles C'' a pu être retirée du spc_C initial :

1. $\exists i \in [1;p]$: $\tilde{a_i}(C) \not\subseteq \tilde{a_i}(C'')$, ce qui est déterminé par le sous-typage entre les types des deux définitions de l'attribut a_i (dans `Search-slottype-position`). Dans ce cas, il y a violation de la spécialisation intensionnelle (comme dans le cas précédent), donc $C \not\leq_{\sigma,I} C''$.
2. $\exists i \in [1;p]$: $a_i \notin A(C)$, autrement dit $n < p$, ce qui est déterminé et traité lors du post-traitement. Dans ce cas, la propriété de l'héritage complet de la spécialisation intensionnelle n'est pas vérifiée, il n'y a pas sous-typage entre les types des classes, donc $C \not\leq_{\sigma,I} C''$.

Nous avons donc montré, par sa contraposée, que

$$\forall C'', C \leq_{\sigma,I} C'' \Longrightarrow C'' \in spc_C$$

Dans les deux conditions précédentes, nous n'avons pas considéré la seconde phase du post-traitement, dans la mesure où elle concerne la réduction transitive de la solution (donc retire des classes parmi celles de ssc_C et spc_C), car elle est supposée avoir été effectuée (dans les hypothèses du lemme). Par ailleurs, cette propriété ne s'applique que si la phase 3 du post-traitement n'a pas encore été effectuée, puisque cette phase peut éliminer des super-classes intensionnellement candidates. Nous avons ainsi montré la complétude des inférences de liens de spécialisation inférés lors de la classification d'une classe. □

Propriété 11 *Il existe au plus une classe intensionnellement candidate à être super-classe de C.*

Preuve Nous prouvons cette propriété par l'absurde. Supposons qu'il existe $C_1, C_2 \in spc_C$; alors d'après la seconde phase du post-traitement (réduction transitive de spc_C), $C_1 \neq C_2$ et $C_1 \not\leq_{\sigma,I} C_2$ et $C_2 \not\leq_{\sigma,I} C_1$. Par ailleurs, nous supposons que C_1, C_2 et C ne sont pas inconsistantes. Donc

$$C_1, C_2 \in spc_C \Longrightarrow C \leq_{\sigma,I} C_1 \text{ et } C \leq_{\sigma,I} C_2$$

d'après le lemme 1. Donc, d'après l'équivalence entre sous-typage entre δ-types *records* et spécialisation intensionnelle, nous déduisons

$$t(C) \leq_\tau t(C_1) \text{ et } t(C) \leq_\tau t(C_2)$$

ainsi, on déduit

$$t(C) \leq_\tau t(C_1) \cap_{Record} t(C_2)$$

et a fortiori, $t(C_1) \cap_{Record} t(C_2) \neq \perp$ puisque les trois classes ici ne sont pas inconsistantes. Or une telle conclusion violerait l'hypothèse d'exclusivité supposée vérifiée avant la classification de C (C_1 et C_2 sont nécessairement cousines). Donc on ne peut pas avoir $t(C_1) \cap_{Record} t(C_2) \neq \perp$, ce qui nous mène à revoir l'hypothèse de départ, et donc à montrer la propriété : il existe au plus une classe intensionnellement candidate à être super-classe de C. □

Propriété 12 *Pour toute classe intensionnellement candidate à être sous-classe de C, sa super-classe avant insertion de C est celle proposée pour être super-classe de C.*

Preuve Soit $sps_C = \{C_s\}$ (d'après la propriété 11, il n'existe qu'une seule super-classe possible pour C, soit C_s). Soit PV le point de vue dans lequel se déroule la classification de C. Nous cherchons donc ici à montrer que, avant classification de C, $\forall C' \in ssc_C$, $\nexists C'_s \in PV$ telle que :

$$C'_s \not\geq_{\sigma,I} C_s \text{ et } C' \leq_{\sigma,I} C'_s$$

Montrons ceci par l'absurde. Supposons que $\exists C' \in ssc_C$ telle que $\exists C'_s \in PV : C'_s \not\geq_{\sigma,I} C_s$ et $C' \leq_{\sigma,I} C'_s$. Puisque $C' \in ssc_C$, nous avons $C' \leq_{\sigma,I} C$. De la même façon, nous savons que $C \leq_{\sigma,I} C_s$. Donc, par transitivité de la spécialisation intensionnelle : $C' \leq_{\sigma,I} C_s$, c'est-à-dire

$$t(C') \leq_\tau t(C_s)$$

Or les hypothèses affirment que $C' \leq_{\sigma,I} C'_s$, c'est-à-dire $t(C') \leq_\tau t(C_s)$. Donc :

$$t(C') \leq_\tau t(C_s) \sqcap_{Record} t(C'_s)$$

Or toujours d'après les hypothèses, qui supposent que C'_s et C_s sont cousines, nous savons que, d'après la propriété d'exclusivité intensionnelle, supposée vérifiée avant classification de C :

$$t(C'_s) \sqcap_{Record} t(C_s) = \perp$$

ce qui est en contradiction avec le fait précédent établi. Nous devons donc revoir nos hypothèses, c'est-à-dire que

1. soit $C_s = C'_s$
2. soit $C'_s \geq_{\sigma,I} C_s$
3. soit $C' \not\leq_{\sigma,I} C_s$
4. soit, enfin, C'_s n'existe pas.

Nous avons ainsi montré, par l'absurde, que pour toute classe intensionnellement candidate à être sous-classe de C, sa super-classe avant insertion de C est celle proposée pour être super-classe de C. □

Propriété 13 *Toutes les sœurs proposées pour C sont disjointes entre elles et disjointes à C. Soit C' l'unique super-classe proposée pour C dans spc_C.*

$$\forall C'' \leq_\sigma C' \ et\ C'' \notin ssc_C, ||C''||^I \cap ||C||^I = \emptyset$$

Preuve Cette propriété est évidente après la réalisation de la phase 3 du post-traitement (page 169). □

Lemme 3 (Correction et cohérence de l'exclusivité intensionnelle) *Le processus de classification d'une classe C par les types de ses attributs garantit le respect des conditions d'exclusion intensionnelle portées sur la hiérarchie de spécialisation, à partir du moment où ces conditions étaient respectées avant la classification de C :*

$$\forall C_1, C_2 \in spc_C, ||C_1||^I \cap ||C_2||^I = \emptyset$$

$$\forall C'_1, C'_2 \in ssc_C, ||C'_1||^I \cap ||C'_2||^I = \emptyset$$

$$\forall C'' \leq_\sigma C' \ et\ C'' \notin ssc_C, ||C''||^I \cap ||C||^I = \emptyset$$

Preuve La première partie est immédiate : il n'y a qu'une seule super-classe possible (propriété 11). La seconde partie est directement issue de la propriété 12. La troisième partie est montrée par la propriété 13 □

Théorème 4 *La classification d'une classe dans un point de vue, réalisée par la classification de types de ses attributs, est complète et cohérente au regard de l'interprétation intensionnelle.*

Preuve Directement à partir des lemmes 1, 2 et 3. □

Annexe C

Algorithmes

Cette annexe contient deux algorithmes parmi ceux que nous avons implémentés dans MÉTÉO, relatifs à la gestion des treillis de δ-types lors de l'insertion (resp. suppression) d'un δ-type. Nous commençons cette annexe par un algorithme du système TROPES dont l'objectif est la détection de cycles dans les descriptions de concepts d'une base de connaissances.

Le treillis de δ-types issus d'un C-type est maintenu à l'intérieur même des structures de δ-types. Plus généralement, un δ-type est constitué des champs de renseignement suivants :

- l'ensemble de valeurs dénoté, géré par les procédures d'accès `get-type(`t`)` et `put-type(`t`,`e`)`,
- liste des sur-types (resp. sous-types), gérée par les procédures d'accès `get-sur-type(`t`)` et `put-sur-type(`t_1`,`t_2`)` (resp. `get-sous-type(`t`)` et `put-sous-type(`t_1`,`t_2`)`),
- liste de dépendances, gérée par `get-depend(`t`)` et `put-depend(`t_1`,`t_2`)`,
- liste des entités de représentation qu'il type, gérée par `get-owners(`t`)` et `put-owner(`t_1`,`t_2`)`,
- indicatif de maintien de treillis, géré par `get-lattice(`t`)` et `put-lattice(`t`,`*bool*`)`.

Rappelons enfin que la notation $\lceil T \rceil$ dénote le δ-type maximum (top) du C-type T, de même que $\lfloor T \rfloor$ dénote le type inconsistant (bottom) de T. La notation $\tilde{a}$ indique la définition d'un attribut d'identificateur a dans un certain contexte de définition (concept ou classe).

L'écriture des algorithmes de cette annexe est proche des langages impératifs, et ceci pour plus de clarté et de compréhension. De ce fait, les fonctions et procédures décrites ici ne respectent pas toujours les signatures données pour elles dans le document, mais la cohérence entre les deux signatures fournies est respectée.

Détections de cycles

Cette section présente l'un des trois algorithmes de détection de cycles que nous avons élaboré : la détection d'un cycle lors de la définition d'un concept en termes d'attributs conduit à une erreur. Il s'agit d'un algorithme du système TROPES, qui est exploité pour éviter les cycles dans les définitions, car ces derniers n'ont pas encore été considérés par MÉTÉO. Soit $\{K_i\}_{i=1..n}$ l'ensemble des concepts d'une base de connaissances. On note $\tilde{a}^i_j$ la définition du $j^{\text{ème}}$ attributs du concept

K_i ($j \in [1..m]$). Il existe un cycle dans la base de connaissances si et seulement si il existe un i tel que K_i est atteignable à partir de lui-même, en suivant un chemin ponctué par des définitions d'attributs. Autrement dit, il existe un cycle si et seulement si :

$$\exists i \in [1..n], \exists a_{j_1}^{i_1}, \cdots, a_{j_s}^{i_p}, s \leq m, p \leq n, \text{ et } \exists k \leq m \text{ tels que}$$

$$\tilde{a}_k^i = \{a_k : K_{i_1}\} \text{ et } \tilde{a}_{j_1}^{i_1} = \{a_{j_1} : K_{i_2}\} \text{ et } \cdots \text{ et } \tilde{a}_{j_s}^{i_p} = \{a_{j_s} : K_i\}$$

L'algorithme que nous présentons ci-dessous teste une base de connaissances au regard des cycles, et termine dès qu'un cycle est trouvé. Il s'agit ainsi d'un simple parcours séquentiel de l'ensemble des concepts d'une base de connaissances, chaque chemin issu de chacun des concepts à partir des attributs complexes est a priori (dans le pire des cas) parcouru. Une variable booléenne globale (nommée `cycle`) est élaborée, qui est mise à `vrai` dès qu'un cycle est détecté.

detecte-cycle(KB) (procédure principale de détection d'un cycle)
Où KB est une base de connaissances.

```
Début
   cycle ⟵ faux
   Pour Chaque (Ki ∈ KB) et (cycle = faux) et (Ki.deja = non-visité) Faire
         Début
            liste ⟵ ∅
            Ki.marque ⟵ non-visité
            exist-cycle (Ki,cycle,liste)
         Fin
   FinPourChaque
   Si (cycle = vrai
      Alors ⟶ erreur
      Sinon ⟶ vrai
   FinSi
Fin
```

exist-cycle(K,cycle,liste) (algorithme de détection d'un cycle à partir d'un concept particulier)
Où **K** est un concept, **cycle** est une variable booléenne (élaborée dans la procédure) qui indique s'il existe ou non un cycle et **liste** est la liste des attributs complexes pouvant engendrer un cycle.

```
Début
   Si (cycle = faux)
      Alors Début
               Si (K.marque = visité)
                  Alors cycle ⟵ vrai
                  Sinon Début
                           K.marque ⟵ visité
                           K.deja ⟵ visité
                           Pour Chaque ((ãt ∈ Ã(K)) et (cycle = faux)) Faire
                                 Début
                                    soit ã_t = a : d
                                    Si (tr-conceptp(d))
                                       Alors Début
                                                liste ⟵ liste ∪ {ã_t}
                                                exist-cycle (d,cycle,liste)
                                                Si (cycle = faux)
                                                   Alors liste ⟵ liste \ {ã_t}
                                                FinSi
```

```
                                                Fin
                                         Fin
                                     Fin
                              FinPourChaque
                              K.marque ⟵ non-visité
                          Fin
              FinSi
          Fin
  FinSi
Fin
```

Où `tr-conceptp(d)` est une fonction de TROPES qui teste si d est un concept de la base de connaissances.

La complexité de cet algorithme est, dans le pire des cas, $\mathcal{O}(k * n^2)$, où k est le nombre de concepts dans la base de connaissances testée, et n est le nombre maximum d'attributs dans la définition d'un de ces concepts.

Les deux autres algorithmes de détection de cycles sont proches de celui-ci : les spécifications sont différentes. Le premier consiste à détecter *tous* les cycles dans les définitions de concepts d'un base de connaissances, tandis que le second s'attache à tester si la définition d'un nouvel attribut de concept n'engendre pas un cycle dans une définition.

Insertion d'un δ-type

L'algorithme d'insertion d'un nouveau δ-type dans un treillis comporte trois phases principales (section 4.4.8), à savoir :

- la recherche de la position du nouveau type
- la maintenance de la structure de treillis
- la mise à jour des liens

La maintenance du treillis est anticipée dès la recherche de la position adéquate, lors de laquelle on distingue deux cas, le premier indiquant une possibilité de violation de la structure de treillis : la position du nouveau type propose des frères pour ce type, ou n'en propose pas. Ainsi, nous proposons deux procédures d'insertion différentes, la première (`insert-and-check-lattice`) qui insère avec résolution d'incorrection potentielle, et la seconde (`insert`) qui est activée lorsqu'il n'y a aucune possibilité d'incorrection. Dans le premier cas, l'algorithme peut être amené à créer deux nouveaux types pour maintenir la structure de treillis. Dans le second cas, l'algorithme peut être amené à supprimer des δ-types qui n'étaient présents que pour maintenir cette structure.

La procédure principale d'insertion est `insert-new-type(`T`,`t`)`, où T est un C-type, et t le δ-type à insérer.

`insert-new-type(`T`,`t`)`
Où T est un C-type, et t le type à insérer. Nous noterons SS l'ensemble des δ-sous-types les plus généraux de t (élaboré), SP l'ensemble des δ-sur-types les plus spécifiques de t (élaboré), FF l'ensemble des δ-types qui ne sont pas comparables avec t au regard du sous-typage, mais qui partagent avec lui un δ-sur-type direct, et `existe` est une variable booléenne qui indique si t

dénote le même ensemble de valeurs qu'un δ-type déjà existant dans le treillis. Enfin, la variable SS' contient les δ-sous-types possibles pour t dans le cas où FF n'est pas vide.

Début
  $SP \longleftarrow \emptyset$
  $SS \longleftarrow \emptyset$
  $SS' \longleftarrow \emptyset$
  $FF \longleftarrow \emptyset$
  `existe` $\longleftarrow$ `faux`
  `search-type-position` (T,`existe`,$\lceil T \rceil$,t,SP,SS,SS',FF)
  Si (`existe` = `faux` et $SS \neq \emptyset$)
    Alors `insert`(T,t,SP,SS)
    Sinon `insert-and-check-lattice`(T,t,SP,SS',FF)
  FinSi
Fin

La procédure `search-type-position` réalise la descente en profondeur d'abord du δ-type t dont on cherche la position. Elle comporte trois tests principaux : t est un δ-sous-type du type testé, il en est un δ-sur-type, ou encore ils sont égaux au regard de leurs dénotations. Si aucun de ces tests n'est vérifié, cela signifie qu'ils sont incomparables. Cette procédure traite chacun de ces cas, et élabore les variables SS, SP, SS' et FF.

`search-type-position` (T,`existe`,t_C,t,SP,SS,SS',FF)
Où T est un C-type, t le type à insérer, t_C le type courant lors du parcours du treillis. Les autres variables sont élaborées, elles correspondent à celles de mêmes noms dans la procédure principales. Les δ-types d'un treillis sont marqués au regard de leur observation lors du parcours, la marque d'un δ-type δt est notée δt.`marque` ; ils sont en outre marqués au regard de la relation de sous-typage entretenu avec le type à insérer, cette marque est notée δt.`marqueType`.

Début
  Si (`existe` = `faux`
    et t_C.`marque` $\neq$ `déjà-visité`
    et t_C.`marque` $\neq$ `incomparable`)
    Alors Début
        t_C.`marque` $\longleftarrow$ `déjà-visité`
        Choix Selon (comparaison entre t_C et t selon l'ordre $\leq_T$)
          • `get-type`(t_C) $=_T$ `get-type`(t) :
            Début
              `put-owners`(t_C, `get-owners`(t_C) $\cup$ `get-owners` (t))
              `existe` $\longleftarrow$ `vrai`
            Fin
          • `get-type`(t_C) $\geq_T$ `get-type`(t) :
            Début
              Pour Chaque ($t_i \in$ `get-sous-types`(t_C)) Faire
                `search-type-position` (T,`existe`,t_i,t,SP,SS,SS',FF)
              FinPourChaque
            Fin
          • `get-type`(t_C) $\leq_T$ `get-type`(t) :
            Début
              Si ($SS' = \emptyset$)
                Alors Début
                    $SS \longleftarrow SS \cup \{t_C\}$
                    t_C.`marqueType` $\longleftarrow$ `soustype`
                    Pour Chaque ($t_i \in$ `get-sur-types` (t_C)) Faire

Si (get-type(t_i) $\geq_T$ get-type (t)
et t_i.marqueType $\neq$ surtype
et t_i.marqueType $\neq$ soustype)
Alors Début
$SP \longleftarrow SP \cup \{t_i\}$
t_i.marqueType $\longleftarrow$ surtype
Fin
FinSi
Fin
FinSi
Fin
• get-type(t_C) et get-type(t) ne sont pas comparables :
Début
Si ($SS \neq \emptyset$)
Alors Début
t_C.marque $\longleftarrow$ incomparable
t_C.marqueType $\longleftarrow$ incomparable
Fin
Sinon Début
$FF \longleftarrow FF \cup \{t_C\}$
t_C.marque $\longleftarrow$ déjà-visité
t_C.marqueType $\longleftarrow$ incomparable
Pour Chaque ($t_i \in$ get-sur-types(t_C)) Faire
Si (get-type(t_i) $\geq_T$ get-type(t)
et t_i.marqueType $\neq$ surtype
et t_i.marqueType $\neq$ soustype)
Alors Début
$SP \longleftarrow SP \cup \{t_i\}$
t_i.marqueType $\longleftarrow$ surtype
Fin
FinSi
FinPourChaque
search-incomparable-subtypes(T,t_C,t,SS')
Fin
FinSi
Fin
FinChoixSelon
Fin
FinSi
Fin

La procédure search-incomparable-subtypes est celle qui retrouve les frères potentiels du type à insérer, en vue de réaliser une insertion effective future, compte-tenu de la possibilité de violation de la structure de treillis. Elle revient à élaborer les sous-types du type t à partir des frères potentiels de t.

search-incomparable-subtypes(T,t_C,t,SS')
Où T est un C-type, t_C le δ-type courant, t le δ-type à insérer, et SS' est l'ensemble des δ-sous-types potentiels de t, qui est élaboré dans cette procédure.

Début
Pour Chaque ($t_i \in$ get-sous-types(t_C)) Faire
Début
Si t_i.marque = pas-visité
Alors Début

```
                Si get-type(t_i) ≤_T get-type(t)
                   Alors Début
                            t_i.marqueType ⟵ soustype
                            t_i.marque ⟵ déjà-visité
                            SS' ⟵ SS' ∪ {t_i}
                         Fin
                   Sinon   search-incomparable-subtypes(T,t_i,t,SS')
                FinSi
             Fin
       FinSi
    Fin
  FinPourChaque
Fin
```

Une fois la position du nouveau δ-type t déterminée par `search-type-position`, l'insertion proprement dite peut être activée. Il existe deux procédures pour cela, comme nous l'avons dit précédemment, qui sont `insert-and-check-lattice` et plus simplement `insert`. La première est activée lorsque t admet des frères (donc le graphe devra peut-être se voir ajouter des δ-types pour le maintien de la structure de treillis). La seconde est activée lorsque t n'admet pas de frère, et l'on doit alors vérifier si t ne remplace pas deux δ-types qui n'étaient là que pour maintenir le treillis.

`insert-and-check-lattice(`T,t,SP,SS,FF`)`
Où T est un C-type, **t** le type à insérer, SS l'ensemble des sous-types proposés, SP l'ensemble des sur-types proposés, et FF l'ensemble des frères de t. Notons que SP et SS contiennent chacun des δ-types incomparables.

```
Début
  Si ((| SP | = 1) et (| SS | = 1))
     Alors Début
              soit t_p ∈ SP
              soit t_s ∈ SS
              put-lattice(t, faux)
              put-sous-types (t_p,get-sous-types(t_p) ∪{t}\SS )
              put-sur-types(t,{ t_p})
              put-sur-types(t_s, get-sur-types(t_s) ∪{t}\SP)
              put-sous-types(t, { t_s })
           Fin
     Sinon Début
              Si (maintain-lattice(T,t,SP,SS,FF) = faux)
                 Alors Début
                          put-lattice(t, faux)
                          put-sur-types(t,SP)
                          Pour Chaque (t_p ∈ SP) Faire
                               put-sous-types(t_p,get-sous-types(t_p) ∪{t}\SS )
                          FinPourChaque
                          put-sous-types(t,SS)
                          Pour Chaque (t_s ∈ SS) Faire
                               put-sur-types(t_s,get-sur-types(t_s) ∪{t}\SP)
                          FinPourChaque
                       Fin
              FinSi
           Fin
  FinSi
Fin
```

`Maintain-lattice` est une fonction qui traite le cas où l'insertion d'un nouveau δ-type viole la structure de treillis : sont créés en conséquence les deux (voire quatre) δ-types nécessaires au respect de cette structure, par les opérations SUP et/ou INF. Lorsqu'il n'y a pas de violation, cette fonction rend `faux`, et l'insertion est directement traitée dans `insert-and-check-lattice`.

`maintain-lattice(`T`,`t`,`SP`,`SS`,`FF`)`
Où T est un C-type, t est le δ-type à insérer, SP sont les sur-types, SS les sous-types et FF les frères de t.

```
Début
  maintenirSP ⟵ faux
  Pour Chaque ((t_p1 ∈ SP) et maintenirSP = faux) Faire
        Pour Chaque ((t_p2 ∈ SP) et (maintenirSP = faux)) Faire
              Si ((t_p1 ≠_T t_p2) et |get-sous-types(t_p1) ∩ get-sous-types(t_p2)|> 1)
                 Alors Début
                          put-lattice(t, faux)
                          maintenirSP ⟵ vrai
                          t_sup ⟵ create-type(T)
                          t_inf ⟵ create-type(T)
                          put-lattice(t_sup,vrai)
                          put-type(t_sup, ⌈T⌉)
                          put-sur-type(t_sup,SP)
                          put-sous-type(t,{ t_inf })
                          Pour Chaque (t_p ∈ SP) Faire
                                put-type(t_sup,get-type(t_sup) ⊓_T get-type(t_p))
                                put-sous-type(t_p,{ t_sup})
                          FinPourChaque
                          put-lattice(t_inf,vrai)
                          put-type(t_inf, ⌊T⌋)
                          put-sous-type(t_inf,{ t } ∪FF)
                          put-sur-type(t_inf,{ t })
                          Pour Chaque (t_s ∈ {t} ∪ FF) Faire
                                put-type(t_inf,get-type(t_inf) ⊔_T get-type(t_s))
                                put-sur-type(t_s,{t_inf})
                          FinPourChaque
                       Fin
              FinSi
        FinPourChaque
  FinPourChaque
  maintenirSS ⟵ faux
  Pour Chaque ((t_s1 ∈ SS) et maintenirSS = faux) Faire
        Pour Chaque ((t_s2 ∈ SS) et (maintenirSS = faux)) Faire
              Si ((t_s1 ≠_T t_s2) et |get-sur-types(t_s1) ∩ get-sur-types(t_s2)|> 1)
                 Alors Début
                          put-lattice(t, faux)
                          maintenirSS ⟵ vrai
                          t_sup ⟵ create-type(T)
                          t_inf ⟵ create-type(T)
                          put-lattice(t_sup,vrai)
                          put-type(t_sup, ⌈T⌉)
                          put-sur-type(t_sup,{t} ∪FF)
                          put-sous-type(t_sup,{t_inf})
                          Pour Chaque (t_p ∈ {t} ∪FF) Faire
                                put-type(t_sup,get-type(t_sup) ⊓_T get-type(t_p))
                                put-sous-type(t_p,{t_sup})
                          FinPourChaque
```

```
                put-lattice($t_{inf}$,vrai)
                put-type($t_{inf}$, $\lfloor T \rfloor$)
                put-sous-type($t_{inf}$,SS)
                put-sur-type($t_{inf}$,{$t_{sup}$})
                Pour Chaque ($t_s \in SS$) Faire
                    put-type($t_{inf}$,get-type($t_{inf}$) $\sqcup_T$ get-type($t_s$))
                    put-sur-type($t_s$,{$t_{inf}$})
                FinPourChaque
              Fin
        FinSi
    FinPourChaque
  FinPourChaque
  ⟶ (maintenirSP ∨ maintenirSS)
Fin
```

Enfin, l'insertion sans risque de violation de la structure de treillis (lorsque le type à insérer n'a pas de frère) est relativement simple ; il s'agit tout de même de vérifier que le treillis ne comporte pas, après l'insertion, de δ-types qui n'étaient là que pour maintenir la structure de treillis. La procédure suivante traite ce cas.

insert(T,t,SP,SS)
Où T est un C-type, t est le δ-type à insérer, SS et SP sont respectivement les sous-types et sur-types proposés pour t.

```
Début
  Si ((| SP |= 1) et (| SS |= 1))
    Alors Début
          soit t_p ∈ SP
          soit t_s ∈ SS
          Si (get-lattice(t_p) = vrai
            et get-lattice(t_s) = vrai
            et get-owner(t_p) = ∅
            et get-owner(t_s) = ∅
            et get-depend(t_p) = ∅
            et get-depend(t_s) = ∅)
            Alors Début ;;; ces types n'étaient là que pour maintenir le treillis
                  put-sur-types(t,get-sur-types(t_p))
                  put-sous-types (t,get-sous-types(t_s))
                  Pour Chaque (t_i ∈ get-sur-types(t_p)) Faire
                      put-sous-types(t_i,t)
                  FinPourChaque
                  Pour Chaque (t_i ∈ get-sous-types(t_s)) Faire
                      put-sur-Types(t_i,t)
                  FinPourChaque
                  put-lattice(t,vrai)
                  put-sur-types(t_s,∅)
                  put-sur-types(t_p,∅)
                  put-sous-types(t_s,∅)
                  put-sous-types(t_p,∅)
              Fin
            Sinon Début
                  put-sous-types(t_p,get-sous-types(t_p) ∪{t}\SS)
                  put-sur-types(t,{ t_p})
                  put-sur-types(t_s, get-sur-types(t_s) ∪{t}\SP)
                  put-sous-types (t, { t_s })
```

```
                put-lattice(t,faux)
              Fin
        FinSi
      Fin
   Sinon Début
        Pour Chaque (t_p ∈ SP) Faire
            Début
              Pour Chaque (t_s ∈ get-sous-types(t_p)) Faire
                  Début
                    Si (t_s.marqueType = soustype)
                      Alors put-sous-types(t_p, get-sous-types(t_p) \{t})
                    FinSi
                  Fin
              FinPourChaque
              put-sous-types(t_p,get-sous-types(t_P) ∪{t})
            Fin
        FinPourChaque
        Pour Chaque (t_s ∈ SS) Faire
            Début
              Pour Chaque (t_p ∈ get-sur-types(t_s)) Faire
                  Début
                    Si (t_p.marqueType = surtype
                      Alors put-sur-types(t_s, get-sur-types(t_s) \{t})
                    FinSi
                  Fin
              FinPourChaque
              put-sur-types(t_s, get-sur-types(t_s) ∪{t})
            Fin
        FinPourChaque
        put-sous-types(t, SS)
        put-sur-types(t, SP)
        put-lattice(t,faux)
      Fin
  FinSi
Fin
```

On remarque que lorsqu'un δ-type n'est pas lié à une entité de représentation, ni ne sert à la construction d'un autre δ-type, alors il n'est accessible que le long des liens de δ-sous-typage : il suffit de supprimer ces liens afin de supprimer le δ-type du treillis.

Complexité de l'insertion

La complexité, dans le pire des cas, de l'insertion d'un δ-type dans un treillis est $\mathcal{O}(n^3)$, où n est le nombre de nœuds du treillis dans lequel est faite l'insertion. Elle est calculée comme suit, où $\mathcal{C}(\texttt{fct})$ est la complexité dans le pire des cas de la procédure `fct`.

$$\begin{aligned}\mathcal{C}(\texttt{insert-new-type}) = \max(&\mathcal{C}(\texttt{search-type-position}),\\ &\mathcal{C}(\texttt{insert}),\\ &\mathcal{C}(\texttt{insert-and-check-lattice})).\end{aligned}$$

Avec :

- $\mathcal{C}(\texttt{search-type-position}) = \mathcal{O}(n^3)$

- $\mathcal{C}(\texttt{insert}) = \mathcal{O}(n^2)$
- $\mathcal{C}(\texttt{insert-and-check-lattice}) = \mathcal{O}(n^2)$

Donc finalement, $\mathcal{C}(\texttt{insert-new-type}) = \mathcal{O}(n^3)$.

Suppression d'un δ-type

La suppression d'un δ-type dans un treillis, telle que nous la présentons ici, a pour pré-condition le fait que le δ-type en question n'a plus de raison d'être dans ce treillis, et que ceci a déjà été vérifié antérieurement. Il s'agit ainsi d'une procédure très simple qui consiste simplement à mettre à jour les liens de δ-sous-typage dans lesquels est impliqué le δ-type à supprimer.

`Remove-type-from-lattice(`T`,`t`)`
Où T est un C-type, et t le δ-type à supprimer.

```
Début
   Pour Chaque (t_p ∈ get-sur-types(t)) Faire
         put-sous-types(t_p, (get-sous-types(t) ∪ get-sous-types(t_p)) \t))
   FinPour
   Pour Chaque (t_s ∈ get-sous-types(t)) Faire
         put-sur-types(t_s, (get-sur-types(t) ∪ get-sur-types(t_s)) \t))
   FinPour
   put-sous-types(t,∅)
   put-sur-types(t,∅)
Fin
```

De cette façon, le δ-type t n'est plus accessible, puisqu'un δ-type ne l'est qu'à partir des liens de δ-sous-typage, s'il n'est pas lié à un autre δ-type ou à une entité de représentation.

La complexité de la procédure de suppression est, dans le pire des cas, $\mathcal{O}(n)$, où n est le nombre de δ-types dans le treillis dans lequel se déroule la suppression.

Annexe D

Indications relatives à l'implémentation de Metéo

Le langage hôte

METÉO, tout comme TROPES a été développé tout d'abord en Lisp version 16, puis a été traduit en TALK, un produit de la société ILOG, qui est un langage intégrant, dans un environnement à la Lisp, les fonctionnalités d'un langage à objets. TALK est compatible avec CommonLisp [ILO94].

Les possibilités de la couche "objet" de TALK sont classiques : distinction classe / instance, définition de champs et de méthodes associés aux classes, sélection dynamique des méthodes, héritage multiple, etc. En outre, l'environnement de programmation favorise la programmation dynamique modulaire, et ceci est en partie dû à l'existence d'un interpréteur et d'un compilateur incrémental.

C-types et δ-types

Les C-types sont implémentés comme des classes TALK, tandis que les δ-types sont des instances de ces classes, et les valeurs restent des valeurs construites à partir des structures de base du langage hôte, d'après les primitives de construction définies pour chaque C-type. Nous verrons que les méthodes sont systématiquement paramétrées par un δ-type, qu'elles s'appliquent aux valeurs de C-types ou aux δ-types.

C-types : classes Talk

Chaque C-type de METÉO est une classe TALK, dont les champs contiennent les informations statiques pour la gestion de ce C-type et de chacun de ses δ-types. Ainsi, une classe TALK correspondant à un C-type est définie de façon minimale comme suit (nous donnons ci-dessous la structure de la classe `Univers`) :

```
(defclass %univers%
    (object)
    (   ;; δ-type top du C-type
        (type-max keyword: type-max: accessor: .val-type-max)
        ;; liste de sur-types d'un δ-type du C-type
```

```
        (l-sur-type keyword: l-sur-type: accessor: .val-l-sur-type)
        ;; liste de sous-types d'un δ-type du C-type
        (l-sous-type keyword: l-sous-type: accessor: .val-l-sous-type)
        ;; liste de δ-types construit sur un δ-type du C-type
        (depend keyword: depend: accessor: .val-depend)
        ;; liste d'entités de représentation typées par un δ-type du C-type
        (owner keyword: owner: accessor: .val-owner)
        ;; booléen indiquant si un δ-type permet de maintenir la structure de treillis
        (lattice keyword: lattice: accessor: .val-lattice)
        ;; marquage lors des parcours de treillis de δ-types
        (marque keyword: marque: accessor: .val-marque)
    ))
```

Le C-sous-typage est représenté par la relation d'héritage entre classes TALK, les C-sous-types de `univers` héritent donc de `univers` les champs ci-dessus, et en définissent de nouveaux qui véhiculent en particulier l'information relative au mode de représentation des δ-types. Nous donnons ci-dessous l'exemple du C-type `multiv` qui hérite successivement les champs de `univers` et ceux de `constructeur` (ces derniers relatifs à la syntaxe de représentation des δ-types), qu'il complète par un champ indiquant la cardinalité des δ-types (les champs en italique sont ceux hérités de `constructeur`, ceux hérités de `univers` ne sont pas indiqués) :

```
(defclass %multiv%
    (%constructeur%)
    (   ;; δ-types de construction
        (type-ref keyword: type-ref: accessor: .val-type-ref)
        ;;domaine de valeurs explicite
        (domaine keyword: domaine: accessor: .val-domaine)
        ;;domaine complémentaire de valeurs
        (comp-dom keyword: comp-dom: accessor: .val-comp-dom)
        ;; cardinalité des valeurs du δ-type
        (card-type keyword: card-type: accessor: .val-card-type)
    ))
```

Chaque C-type définit ses deux prédicats (appartenance d'une valeur, et égalité de deux valeurs), comme des méthodes paramétrées obligatoirement par un δ-type du C-type, généralement le type top : c'est cette paramétrisation qui permet la sélection dynamique de la méthode le long du graphe d'héritage. Ainsi, les méthodes pour la définition de ces prédicats sont définies génériquement, puis redéfinies pour chaque classe correspondant à un C-type. Nous donnons ci-dessous l'exemple du C-type `entier`.

```
;; déclaration des méthodes
(defgeneric .member-val-adt (val type))
(defgeneric .values-equality (v1 v2 type))
;; définition des méthodes pour entier
(defmethod .values-equality (v1 v2 (type %entier%))
   (ignore type)
   (if (symbolp v1)
       (and (symbolp v2) (eq v1 v2))
     (unless (symbolp v2)
       (and (intp v1) (intp v2) (i= v1 v2)))))
(defmethod .member-val-adt (val (type %entier%))
   (or (intp val)
       (.values-equality val (.borne-sup-max type) type)
       (.values-equality val (.borne-inf-min type) type)))
```

où `.borne-sup-max type` (resp. `.borne-sup-min`) est la valeur maximum (resp. minimum) d'un C-type. Dans le cas d'`entier`, il s'agit des symboles TALK '+inf et '-inf.

Selon le même procédé, toutes les opérations portant sur les valeurs des C-types sont ainsi implémentées comme des méthodes dynamiquement sélectionnées. Par exemple, l'opération d'addition chez les entiers est définie comme suit :

```
(defmethod .addition (v1 v2 (type %entier%))
   (if (or (and (.values-equality v1 (.borne-sup-max type) type)
                (.values-equality v2 (.borne-inf-min type) type))
           (and (.values-equality v2 (.borne-sup-max type) type)
                (.values-equality v1 (.borne-inf-min type) type)))
       (.tropes.error.raise-error 'bad-op (list v1 v2) '.addition)
     (if (or (.values-equality v1 (.borne-sup-max type) type)
             (.values-equality v1 (.borne-inf-min type) type))
         v1
       (if (or (.values-equality v2 (.borne-sup-max type) type)
               (.values-equality v2 (.borne-inf-min type) type))
           v2
         (+ v1 v2)))))
```

En théorie, la dernière ligne de la méthode aurait dû être

```
(.succ (.addition v1 (.pred v2 type) type) type)
```

mais pour des raisons évidentes d'efficacité, nous avons préféré bénéficier des opérations optimisées du langage hôte.

Nous remarquons dans l'exemple ci-dessus l'intérêt de la redéfinition de ce type d'opération en ce qui concerne la gestion de l'infini.

δ-types : instances Talk

Les δ-types sont implémentés comme des instances de classes TALK, et plus précisément, comme des instances des classes correspondant aux C-types. Ces instances valuent donc naturellement les champs définis dans leur classe de création. Chaque δ-type est ainsi informé, au sein même de sa structure, d'une part des liens de sous-typage qu'il entretient avec les autres δ-types du même C-type, et d'autre part de l'ensemble de valeurs qu'il dénote.

Opérations sur les δ-types

De la même façon que les fonctions sur les valeurs, les opérations de gestion des δ-types sont définies, pour chaque C-type, à l'aide de méthodes de la classe correspondant au C-type. Rappelons que ces opérations sont principalement la normalisation, le δ-sous-typage, le GLB, le LUB et l'élimination, sans parler de l'initialisation de δ-types. Leur implémentation nécessite celle de nombreuses autres méthodes plus spécifiques, comme l'illustre l'exemple de la fonction de GLB définie de la même façon pour tous les C-types ordonnés :

```
(defgeneric .GLB (type1 type2))
(defmethod .GLB ((t1 %ordonne%) (t2 %ordonne%))
   (if (not (equal (class-name (class-of t1)) (class-name (class-of t2))))
```

```
      (tropes.erreur.raise-error 'c-it (list t1 t2) '.GLB-%ordonne%)
    (.creer-type t1 (list () () (.I-linter (.val-domaine t1) (.val-domaine t2) t1)))))
```

où `.creer-type` est la méthode d'initialisation d'un δ-type, et `.I-linter` est une méthode propre aux ordonnés qui calcule l'intersection de deux listes d'intervalles. La sélection de la méthode la plus spécifique s'effectue à l'exécution ; si `.GLB` est appelée avec des δ-types `entier`, ce sera la méthode `.creer-type` définie localement dans la classe des entiers qui sera activée.

Toutes les autres opérations locales définies pour la gestion de δ-types d'un C-type sont ainsi implémentées comme des méthodes de ce C-type, activées dynamiquement grâce au paramétrage par au moins un δ-type (une instance).

Module de contrôle de Metéo

Le module de contrôle de METÉO est assuré par des fonctions TALK, qui manipulent les objets par composition des méthodes.

Mécanismes spécifiques

Les mécanismes spécifiques sont les opérations qui centralisent certaines opérations locales, celles utiles pour l'interface avec TROPES. Elles sont donc implémentées comme des fonctions TALK qui sélectionnent la méthode appropriée à la sémantique de la fonction. De ce fait, les méthodes ne font pas partie de l'interface TROPES / METÉO.

Nous retrouvons dans ces mécanismes les sept opérations de manipulations de δ-types traduisant des manipulations ensemblistes (intersection, union, etc.). Par exemple, l'opération de normalisation est définie comme suit :

```
(defun normal-form (dt)
   (or (:normalize dt) (list dt 'type-vide)))
```

La méthode sélectionnée pour effectuer la normalisation proprement dite sera celle définie pour le C-type duquel est issu le δ-type `dt` ; il s'agit d'une fonctionnalité du langage hôte. Cette centralisation des opérations spécifiques permet la récupération homogène des erreurs de types.

Mécanismes globaux

Les mécanismes globaux du module de contrôle de METÉO sont ceux qui composent les mécanismes spécifiques, ainsi que ceux qui gèrent les treillis de δ-types. Nous trouvons d'ailleurs, parmi ces mécanismes, les opérations de suppression et d'insertion de δ-types. Il s'agit aussi d'opérations implémentées comme des fonctions TALK. Nous ne donnons pas d'exemple ici, en raison de la taille de ses opérations, mais les algorithmes de certaines d'entre elles sont donnés en annexe C.

Les opérations de l'interface, et en particulier les opérations de typage ou de vérifications de types, sont elles-aussi réalisées par des fonctions TALK qui composent les opérations spécifiques et globales.

Création de C-types

Dans l'état actuel de l'implémentation, il n'existe qu'une fonction de l'interface Tropes / Metéo réalisant l'extensibilité de Metéo, qui permet l'ajout de C-type correspondant à un type abstrait de données déjà existant dans le langage hôte. En outre, cette fonction ne permet pas de modifier le mode de représentation des δ-types, et ne modifie pas la syntaxe de Tropes.

Il s'agit de la fonction `tr-create-adt`, qui prend pour paramètre l'identificateur du C-sur-type, le nom du C-type à ajouter, ainsi que la donnée des différents prédicats à définir au minimum pour le C-type. Nous donnons ci-dessous un exemple de création du C-type `time` pour la représentation du temps, qui est prédéfini dans le langage hôte (`timep` teste l'appartenance d'une valeur Talk au type de données `time` de ce langage) :

```
(tr-create-adt %discret% %time%
   member: timep
   equality: equal
   superior: >
   successor: +1
   predecessor: -1
)
```

où `timep` est une fonction du langage hôte.

Cette instruction peut être compilée et chargée, le C-type sera alors disponible au sein de Tropes comme tout autre. Il est possible de définir par le biais de cette fonction plusieurs autres opérations pour le C-type, qui seront traduites, par Metéo, en méthodes du C-type. `tr-create-adt` est définie comme une macro qui s'assure de la présence des définitions exigées pour la création du C-type, et qui crée alors les méthodes associées avec l'identificateur adapté.

Actuellement, cette opération ne permet pas la création de méthodes qui ne sont pas déjà définies de façon générique ; pour que de telles fonctions soient intégrées à Tropes, cela nécessite la recompilation partielle de ce dernier, après écriture du code de ces fonctions dans des fichiers spécifiques.

Bibliographie

[ABB⁺83] M. Ahlsén, A. Björnerstedt, S. Britts, C. Hultén, et L. Söderlund. Making type changes transparent. *IEEE Workshop on Languages for Automation*, pages 110–117, Chicago (MI, US), Novembre 1983. IEEE Computer Society Press.

[AC91] R.M. Amadio et L. Cardelli. Subtyping recursive types. *18th POPL*, Orlando (FL, US), 1991.

[AK86] H. Aït-Kaci. An algebraic semantics approach to effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[AKBLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, et R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, Janvier 1989.

[AKP91] H. Aït-Kaci et A. Podelski. Towards a meaning of LIFE. *3rd International Symposium on Programming Language Implementation and Logic Programming*, éd. par W. Bibel et P. Jorrand, volume 528 de *Lecture Notes in Computer Science*, pages 255–274, Passau (Germany), 1991. Springer-Verlag.

[Att91] G. Attardi. An analysis of taxonomic reasoning. *Inheritance Hierarchies in Knowledge Representation and Programming Language*, éd. par D. Nardi M. Lenzerini et M. Simi, chapitre 3, pages 29–49. John Wiley and Sons, 1991.

[Bar91] G. Barbedette. Schema modification in the LISPO2 persistent object-oriented language. *European Conference on Object-Oriented Programming*, volume 512 de *Lecture Notes in Computer Science*, pages 77–96, Geneva (Switzerland), Juillet 1991.

[BB92a] D. Beneventano et S. Bergamaschi. Subsumption for complex object data model. *ERCIM Workshop on Theoretical and Experimental Aspects of Knowledge Representation*, pages 21–32, Pisa (Italy), Mai 1992.

[BB92b] A. Borgida et R.J. Brachman. Customizable classification inference in the ProtoDL description management system. Unpublished, Octobre 1992.

[BdBZ93] H. Balsters, R.A. de By, et R. Zicari. Typed sets as a basis for object-oriented database schemas. *7th European Conference on Object-Oriented Programming*, éd. par O.M. Nierstrasz, volume 707 de *Lecture Notes in Computer Science*, pages 161–184, Kaiserslautern (Allemagne), Juillet 1993. Springer-Verlag.

[BDG⁺88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E Keene, G. Kiczales, et D.A. Moon. Special issue, Common Lisp Object System Specification. *ACM Sigplan Notices*, 23, Septembre 1988.

[BDMN73] G. Birtwistle, O.J. Dahl, B. Myhrhaug, et K. Nygaard. SIMULA begin. Petrocelli Charter, New-York (NY,US), 1973.

[BDS93] M. Buchheit, F.M. Donini, et A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 704–709, Chambéry (France), Août 1993.

[BFL83] R.J. Brachman, R.E. Fikes, et H.J. Levesque. KRYPTON: A Functional Approach to Knowledge Representation. *IEEE Computer*, pages 67–73, Octobre 1983.

[BGN89] H.W. Beck, S.K. Gala, et S.B. Navathe. Classification as a query processing technique in the CANDIDE semantic data model. *International Data Engineering Conference, IEEE*, pages 572–581, Los Angeles (CA, US), 1989.

[BH91] F. Baader et B. Hollunder. KRIS: knowledge representation and inference system. *SIGART bulletin*, 2(3), Juin 1991.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, et H.K. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Conference*, San Francisco (CA, US), 1987.

[BL82] R.J. Brachman et H.J. Levesque. Competence in knowledge representation. *AAAI*, pages 189–192, 1982.

[Bor88] A. Borgida. Class hierarchies in information systems: sets, types or prototypes? *Data Types and Persistence*, éd. par M.P. Atkinson, P. Buneman, et R. Morrison, Topics in Information Systems, chapitre 9, pages 137–154. Springer-Verlag, 1988.

[Bor91] A. Borgida. Terminologic frames as types: inference rules and prospective applications. Rapp. Techn. 280, Rutgers University, (NJ, US), Mai 1991.

[Bor92] A. Borgida. From type systems to knowledge representation: natural semantic specifications for description logics. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):93–126, Avril 1992.

[BPS94] A. Borgida et P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, 1:277–308, Juin 1994.

[Bra77] R.J. Brachman. What's in a concept: structural foundations for semantic networks. *International Journal of Man-Machine Studies*, 9:127–159, 1977.

[Bra83] R.J. Brachman. What *is-a* is and isn't: an analysis of taxonomic links in sematic networks. *Computer*, 16(10):30–36, Octobre 1983.

[Bra85] R.J. Brachman. "I lied about the trees", or defaults and definitions in knowledge representation. *AI Magazine*, 6(3):80–93, 1985.

[Bra92] R.J. Brachman. Reducing CLASSIC to practice: knowledge representation theory meets reality. *3rd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge (MA, US), Octobre 1992.

[Bri85] J.P. Briot. Les métaclasses dans les langages orientés objets. *5ème CARFIA*, pages 755–764, Grenoble (France), 1985.

[BS83] D.G. Bobrow et M. Stefik. *The LOOPS manual: a data and object-oriented programming system for Interlisp*. Knowledge-based VLSI Design Group Memo KB-VLSI-81-13, Xerox PARC, Palo Alto (CA, US), 1983.

[BS85] R.J. Brachman et J.G. Schmoltze. An overview of the KL-ONE knowledge representation language. *Cognitive Science*, 9:171–216, 1985.

[BW77] D.G. Bobrow et T.W. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

[BW87] K.B. Bruce et P. Wegner. An algebraic model of subtypes and inheritance. Rapp. techn., Brown university, Septembre 1987.

[Cap93] C. Capponi. Classification des classes par les types. *Représentations Par Objets*, éd. par EC2, pages 215–224, La Grande Motte (France), Juin 1993.

[Car84] L. Cardelli. A semantics of multiple inheritance. *Lecture Notes in Computer Science*, 173, 1984.

[Cas87] Y. Caseau. Étude et réalisation d'un langage objet: LORE. Thèse d'État, université de Paris-Sud (France), 1987.

[CC93] C. Capponi et M. Chaillot. Construction incrémentale d'une base de classes correcte du point de vue des types. *journée Validation, JAVA*, Saint-Raphaël (France), Mars 1993.

[CDE+95] B. Carré, R. Ducournau, J. Euzenat, A. Napoli, et F. Rechenmann. Classification et objets :programmation ou représentation? *5ème journées nationales du PRC-GDR "Intelligence Artificielle"*, pages 213–237, Nancy (France), Février 1995.

[CEG95] C. Capponi, J. Euzenat, et J. Gensel. Objects, types and constraints as classification schemes. *Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz (CA, US), à paraître, Août 1995.

[CG90] B. Carré et J.M. Geib. The point of view notion for multiple inheritance. *OOPSLA-ECOOP*, volume 25 de *ACM Sigplan Notices*, pages 312–321, Ottawa (CDN), 1990.

[CHC90] W.R. Cook, W.L. Hill, et P.S. Canning. Inheritance is not subtyping. *7th annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco (CA, US), Janvier 1990.

[CM91] L. Cardelli et J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, Mars 1991.

[Coi87] P. Cointe. Metaclasses are first-class : the ObjVlisp model. *2nd OOPSLA*, pages 156–167, Orlando (FL, US), 1987.

[Coo88] W. Cook. The semantics of inheritance. Rapp. techn., Brown university, Mars 1988.

[Cra95] Y. Crampé. Suggestion de révisions dans une base de connaissances à objets. Rapport interne, INRIA Rhône-Alpes, Grenoble (France), Avril 1995.

[CW85] L. Cardelli et P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Décembre 1985.

[Dav87] H.E. Davis. VIEWS: muliple perspectives and structured objects in a knowledge representation language. Master's thesis, MIT, 1987.

[Dek94] L. Dekker. *FROME: représentation multiple et classification d'objets avec points de vue.* Thèse de doctorat, LIFL, Université des Sciences et Technologies de Lille, Lille (France), Juin 1994.

[DH89] R. Ducournau et M. Habib. La multiplicité de l'héritage dans les langages à objets. *Techniques et Sciences Informatiques*, 8(1):41–62, 1989.

[DLNN91a] F.M. Donini, M. Lenzerini, D. Nardi, et W. Nutt. The complexity of concept languages. *2nd International Conference on Principles of Knowledge Representation and Reasoning*, éd. par J. Allen, R. Fikes, et E. Sandewall, pages 151–162. Morgan Kaufmann, 1991.

[DLNN91b] F.M. Donini, M. Lenzerini, D. Nardi, et W. Nutt. Tractable concept languages. *12th IJCAI*, pages 458–463, Sydney (Australie), 1991.

[DLNS94] F.M. Donini, M. Lenzerini, D. Nardi, et A. Schaerf. Deduction in concept languages: from subsumption to instance checking. *Journal of Logical Computation*, 4(4):423–452, 1994.

[DMO92] R. Dionne, E. Mays, et F.J. Oles. A non-well-founded approach to terminological cycles. *AAAI'92*, San Jose (CA, US), Juillet 1992.

[DMO93] R. Dionne, E. Mays, et F.J. Oles. The equivalence of model-theoretic and structural subsumption in description logics. *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 710–716, Chambéry (France), Août 1993.

[DP91] J. Doyle et R.S. Patil. Two theses of knowledge representations: language restrictions, taxonomic classification and the utility of representation services. *Artificial Intelligence*, 48(3):261–297, Avril 1991.

[DT88] S. Danforth et C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, Novembre 1988.

[Duc88] R. Ducournau. *YAFOOL version 3.22.* SEMA METRA, Montrouge (France), 1988.

[Duc95] R. Ducournau. Les systèmes classificatoires. Cours de DEA, LIRMM, Montpellier (france), 1995.

[Dug88] P. Dugerdil. *OBJLOG II, Guide d'utilisation.* Université d'Aix-Marseille II, Aix (France), 1988.

[Dug89] P. Dugerdil. Inheritance mechanisms in the OBJLOG language: multiple selective and multiple vertical with points of view. *Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, pages 233–242, Viareggio (Italie), 1989.

[Ell93] G. Ellis. Efficient retrieval from hierarchies of objects using lattice operations. *1st International Conference on Conceptual Graphs for Knowledge Representation*, éd. par G.W. Mineau, B. Moulin, et J.F. Sowa, volume 699 de *Lecture Notes in Artificial Intelligence*, pages 274–293, Quebec (PQ, Canada), Août 1993. Springer-Verlag.

[Erw93] M. Erwig. Specifying type systems with multi-level order-sorted algebra. *Workshop on Algebraic Methodology and Software Technology*, éd. par C.J. van Rijsbergen, pages 177–184, Enschede (NL), 1993. Springer-Verlag.

[Euz93] J. Euzenat. On a purely taxonomic and descriptive meaning for classes. *IJCAI workshop on Object-based Representation Systems*, Chambéry (France), Août 1993.

[Euz94] J. Euzenat. Classification dans les représentations par objets : produits de systèmes classificatoires. *9ème RFIA*, éd. par AFCET, volume 2, pages 185–196, Paris (France), Janvier 1994.

[Fal95] A. Fall. An abstract framework for taxonomic encoding. *Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz (CA, US), à paraître, Août 1995.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hock, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Tyan, et M.C. Shan. IRIS: an object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, Janvier 1987.

[Fer84] J. Ferber. Quelques aspects du caractère self réflexif du langage MERING. *2èmes journées JLOO*, volume 41 de *Bigre+Globule*, pages 277–290, Brest (France), 1984.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, et J. Meseguer. Principles of OBJ2. *Principles of Programming Languages*, pages 52–66, 1985.

[Fin86] T.W. Finin. Interactive classification: a technique for acquiring and maintaining knowledge bases. *Proceedings of the IEEE*, 74(16), Octobre 1986.

[FK85] R. Fikes et T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, Septembre 1985.

[Gen93] J. Gensel. Expression et satisfaction de contraintes dans TROPES. *Représentations Par Objets*, pages 51–62, La Grande Motte (France), Juin 1993.

[Gen95] J. Gensel. Contraintes et modèle de connaissances à objets. Application aux relations et aux filtres. Thèse de doctorat (à paraître), Université Joseph Fourier, Grenoble, France, 1995.

[GH78] J. Guttag et J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GHW85] J.V. Guttag, J.J. Horning, et J.M. Wing. Larch in five easy pieces. Rapp. techn., Digital Systems Research Center, Palo Alto (CA, US), 1985.

[Gir95] P. Girard. Classification d'instances hypothétiques. Thèse de doctorat (à paraître), Université Joseph Fourier, Grenoble (France), 1995.

[GJM85] J.A. Goguen, J.P. Jouannaud, et J. Meseguer. Operational semantics for order-sorted algebra. *12th Colloqium on Automata, Languages and Programming*, éd. par G. Goos et J. Hartmanis, volume 194, pages 221–231. Springer-Verlag, Nafplion (Grèce), Juillet 1985.

[Got87] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, Janvier 1987.

[GR83] A. Goldberg et D. Robson. Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

[Gra93] M. Graves. *Theories and tools for designing application-specific knowledge base data models*. PhD thesis, University of Michigan, (US), 1993.

[Gua92] N. Guarino. Concepts, Attribute and Arbitrary Relations – Some linguistic and ontological criteria for structuring knowledge bases. *Data and Knowledge Engineering*, pages 249–261, 1992.

[Gut77] J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6), Juin 1977.

[HBFC+91] J.P. Haton, N. Bouzid, M.C. Haton F. Charpillet, B. Lâasri, H. Lâasri, P. Marquis, T. Mondot, et A. Napoli. *Le raisonnement en intelligence artificielle*. InterEditions, 1991.

[HKNP92] J. Heinsohn, D. Kudenko, B. Nebel, et H.J. Profitlich. An empirical analysis of terminological representation systems. Rapp. Techn. RR-92-16, DFKI, Kaiserslautern (Germany), Mai 1992.

[HN94] M. Habib et L. Nourine. Bit-vector encoding for partially ordered sets. *International Workshop on Order, Algorithms and Applications (ORDAL)*, éd. par V. Bouchitte et M. Morvan, volume 831 de *Lecture Notes in Computer Science*, Lyon (France), Juillet 1994. Springer-Verlag.

[ILO94] ILOG, Gentilly (France). *ILOG* TALK, *version 3.01 (Beta 1)*, 1994.

[JEG+95] M. Jarke, S. Eherer, R. Gallersdörfer, M.A. Jeusfeld, et M. Staudt. ConceptBase – a deductive object base manager. *Journal of Intelligent Information Systems, Special issue on Advances in Deductive Object-Oriented Databases*, 1995.

[Kay84] D. Kayser. Comment représenter la typicalité? Rapp. techn., Université de Paris-Sud, Orsay (France), Décembre 1984.

[Kay93] D. Kayser. Un point de vue cognitif sur représentation et référence. *Modèles et Concepts pour la Science Cognitive*, Sciences et Technologies de la Connaissance, pages 165–178. Presses Universitaires de Grenoble, 1993.

[KR91] C. Kindermann et P. Randi. Object recognition and retrieval in the BACK system. *International Working Conference on Cooperating Knowledge Based Systems*, éd. par S.M. Deen, pages 311–325, Berlin (Allemagne), 1991. Springer.

[LB85] H.J. Levesque et R.J. Brachman. A fundamental tradeoff in knowledge representation and reasoning (revised version). *Readings in Knowledge Representation*, éd. par R.J. Brachman et H.J. Levesque. Morgan Kaufmann, San Mateo (CA, US), 1985.

[Lip82] T.A. Lipkis. A KL-ONE classifier. *1981* KL-ONE *Workshop*, éd. par J.G. Schmolze et R.J. Brachman, pages 126–143, Jackson (NH, US), 1982.

[LNE89] J.A. Larson, S.B. Navathe, et R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Sofware Engineering*, 15(4):449–463, Avril 1989.

[LT94] F. Lemaire et N. Tayar. Construction incrimentale de bases consensuelles. *Journées Jeunes Chercheurs en Intelligence Artificielle*, Marseille (France), 1994.

[Mac82] B.J. MacLennan. Values and objects in programming languages. *SIGPLAN Notices*, 17(12):70–79, Décembre 1982.

[Mac91a] R. MacGregor. The evolving technology of classification-based knowledge representation systems. *Principles of Semantics Networks*, éd. par J.F. Sowa, chapitre 13, pages 385–400. Morgan Kaufmann, 1991.

[Mac91b] R. MacGregor. Inside the LOOM classifier. *SIGART Bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):88–92, Juin 1991.

[Mae87] P. Maes. Concepts and experiments in computational reflection. *2nd OOPSLA*, pages 147–155, Orlando (FL,US), 1987.

[Mar93] O. Marino. Raisonnement classificatoire dans une représentation à objets multi-points de vue. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Octobre 1993.

[MB91] R. MacGregor et M.H. Burstein. Using a description classifier to enhance knowledge representation. *IEEE Expert*, 6(3):41–46, Juin 1991.

[MB92] R. MacGregor et D. Brill. Recognition algorithms for the LOOM classifier. *AAAI*, San Jose (CA, US), Juillet 1992.

[MDW91] E. Mays, R. Dionne, et R. Weida. K-Rep system overview. *SIGART Bulletin*, 23((2-5)):329–342, Juin 1991.

[Mil84] R. Milner. A proposal for standard ML. *The Symposium on LISP and Functional Programming*, pages 184–197, Austin (TX, US), Août 1984. ACM.

[Min75] M. Minsky. A framework for representating knowledge. *The psychology of computer vision*, éd. par P. Winston, chapitre 6, pages 211–281. Mc Graw-Hill, 1975.

[Mit84] J. Mitchell. Type inference and type containment. *Semantics of Data Types*, volume 173 de *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, New-York (NY, US), 1984.

[ML79] P. Martin-Löf. Constructive mathematics and computer programming. *International Congress of Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover (Allemagne), 1979.

[MNC+89] G. Masini, A. Napoli, D. Colnet, D. Léonard, et K. Tombre. *Les langages à objets*. InterEditions, Paris (France), 1989.

[Mol93] P. Moller. MATISSE. *Génie Logiciel et Systèmes Experts*, (31):18–25, Juin 1993.

[MPS86] D.B. MacQueen, G. Plotkin, et R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[MRU90] O. Marino, F. Rechenmann, et P. Uvietta. Multiple perspectives and classification mechanism in object-oriented representation. *9th European Conference on Artificial Intelligence*, éd. par Springer-Verlag, pages 425–430, Stockholm (Sweden), Août 1990.

[MS82] D.B. MacQueen et R. Sethi. A higher order polymorphic type system for applicative languages. *Symposium on Lisp and Functional Programming*, pages 243–252, Pittsburgh (PA, US), 1982.

[Nap92a] A. Napoli. Représentations à objets et raisonnement par classification en intelligence artificielle. Thèse d'état, CRIN-INRIA Lorraine, Nancy (France), Janvier 1992.

[Nap92b] A. Napoli. Subsumption and classification-based reasoning in object-based representations. *10th European Conference on Artificial Intelligence*, éd. par B. Neumann, Vienna (Austria), Août 1992. John Wiley and Sons.

[ND92] A. Napoli et R. Ducournau. Subsumption in object-based representations. *ERCIM Workshop on Theoretical and Experimental Aspects of Knowledge Representation*, pages 1–9, Pisa (Italy), 1992.

[Neb90] B. Nebel. *Reasoning and revision in hybrid representation systems*, volume 422 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin (Allemagne), 1990.

[Neb91] B. Nebel. Terminological cycles. *Principles of semantic networks*, éd. par John Sowa, chapitre 11, pages 331–361. Morgan Kaufmann, 1991.

[NWW91] A.H.H. Ngu, L. Wong, et S. Widjojo. On canonical and non-canonical classifications. *2nd International Conference on Deductive and Object-Oriented Databases*, volume 566 de *Lecture Notes in Computer Science*, Munich (Allemagne), Décembre 1991.

[OK89] B. Owsnicki-Klewe. A general characterization of term description languages. *Workshop on Sorts and Types in Artificial Intelligence*, éd. par K.H. Bläsius, U. Hedtstück, et C.R. Rollinger, volume 418 de *Lecture Notes in Artificial Intelligence*, pages 183–189. Springer-Verlag, Eringerfeld (Allemagne), 1989.

[Old94] E. Oldberg. MultiPerspectives: the classification dimension of schema modification management for object-oriented databases. *TOOLS-USA*, 1994.

[Pag92] Lin Pagdham. Defeasible inheritance: a lattice-based approach. *Computer Math. Applic.*, 23(6-9):527–541, 1992.

[Pel91] C. Peltason. The BACK system: an overview. *SIGART Bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):114–119, Juin 1991.

[Per92a] G. Perrière. Application d'une représentation par objets des connaissances à la modélisation de certains aspects de l'expression des gènes chez *Escherichia coli*. Thèse de doctorat, Université Claude Bernard, Lyon (France), Octobre 1992.

[Per92b] J.F. Perrot. Langages à objets, programmation par objets. Rapp. Techn. 92/34, Institut Blaise Pascal, LAFORIA, Paris (France), Novembre 1992.

[PP87] B. Pivot et M. Prokop. Définition et réalisation d'une fonctionnalité de classification dans SHIRKA. Rapport d'année spéciale en intelligence artificielle, Institut National Polytechnique de Grenoble, Grenoble (France), 1987.

[PS84] P.F. Patel-Schneider. Small can be beautiful in knowledge representation. Rapp. Techn. 660, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto (CA, US), 1984.

[PS87] J. Penney et J. Stein. Class modification in the Gemstone object-oriented DBMS. *Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 111–117, Orlando (FL, US), Octobre 1987.

[PS89a] P.F. Patel-Schneider. A four-valued semantics for terminological cycles. *Artificial Intelligence*, 38(3):319–351, Avril 1989.

[PS89b] P.F. Patel-Schneider. Undecidability of subsumption in NIKL. *Artificial Intelligence*, 39(2):263–272, Juin 1989.

[PS90] P.F. Patel-Schneider. Pratical, Object-Based Knowledge Representation For Knowledge-Based Systems. *Informations Systems*, 15(1):9–19, 1990.

[PS91] J. Palsberg et M.I. Schwartzbach. Three discussions on object-oriented typing. *European Conference on Object-Oriented Programming*, Workshop on Types, Inheritance and Assignments, Genève (Suisse), Juillet 1991.

[PSMB+91] P.F. Patel-Schneider, D.L. McGuinness, R.J. Brachman, L.A. Resnick, et A. Borgida. The CLASSIC knowledge representation system : guiding principles and implementation rational. *SIGART Bulletin, Special issue on implemented krnowledge representation and reasoning systems*, 2(3):108–113, Juin 1991.

[Qui68] R. Quillian. *Semantic memory*, pages 227–270. Semantic Information Processing. MIT Press, Cambridge (MA, US), 1968.

[Qui93] A. Quintero. Parallélisation de la classification d'objets dans un modèle de connaissances multi-points de vue. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Juin 1993.

[RDP+91] Brachman R.J., McGuinness D.L., Patel-Schneider P.F., Resnick L.A., et Borgida A. Living with CLASSIC, When and how to use a KL-ONE-like language. *Principles of Semantic Networks*, éd. par J.F. Sowa, chapitre 14, pages 401–456. Morgan Kaufmann, 1991.

[Rec85] F. Rechenmann. SHIRKA: mécanismes d'inférences sur une base de connaissances centrée objets. *5ème CARFIA*, pages 1243–1254, Grenoble (France), 1985.

[Rec88] F. Rechenmann. *SHIRKA: système de gestion de bases de connaissances centrées objets.* INRIA/ARTEMIS, Grenoble (France), 1988.

[Rec89] F. Rechenmann. SHIRKA: un modèle de connaissance centré-objets. rapport interne, laboratoire ARTEMIS/IMAG, Grenoble (France), 1989.

[Rec93] F. Rechenmann. Integrating procedural and declarative knowledge in object-based knowledge models. *IEEE International Conference on Systems, Man and Cybernetics*, pages 98–101, Le Touquet (France), Octobre 1993.

[RG77] R.B. Roberts et I.P. Goldstein. *The FRL manual, AI Memo 409.* AI Lab, MIT, Cambridge (MA, US), 1977.

[Ros75] E. Rosch. Cognitive representations of semantic categories. *Journal of Experimental Psychology*, 104(3):192–233, 1975.

[RSS73] L.J. Rips, E.J. Shoben, et E.E. Smith. Semantic distance and the verification of semantic relations. *Journal of Verbal Learning and Verbal Behavior*, 12:1–20, 1973.

[SB85] M. Stefik et D.G. Bobrow. Object-oriented programming: themes and variations. *AI Magazine*, 6(4):40–62, 1985.

[Sch85] J.G. Schmoltze. The language and semantics of NIKL. Rapp. techn., BBN Laboratories, Cambridge (MA, US), 1985.

[Sch88] K. Schild. Undecidability of subsumption in $\mathcal{U}$. Rapp. Techn. 67, KIT-Report, FB Informatik, Technische Universität Berlin, Berlin (Allemagne), 1988.

[Sch93] A. Schaerf. Reasoning with individuals in concept languages. Rapp. Techn. 07.93, Dipartimento di Informatica e Sistemistica, Universtità La Sapienza, Roma (Italie), 1993.

[Sch95] O. Schmeltzer. Modélisation de cartes génomiques : une formalisation et un algorithme de construction fondé sur le raisonnement temporel. Rapp. techn., Université Joseph Fourier, Grenoble (France), Janvier 1995.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal Computing*, pages 522–587, 1976.

[SGD93] S. Scherrer, A. Geppert, et K.R. Dittrich. Schema evolution in NO^2. Rapp. Techn. 93.12, Institut für Informatik der Universität Zürich, Zürich (Germany), Avril 1993.

[SI83] J.G. Schmolze et D.J. Israel. KL-ONE: semantics and classification. Rapp. Techn. 5421, BBN Laboratories, Cambridge (MA, US), 1983.

[SL83] J.G. Schmolze et T.A. Lipkis. Classification in the KL-ONE knowledge representation system. *8th International Joint Conference on Artificial Intelligence*, Karlsruhe (Allemagne), 1983.

[SM86] R.E. Stepp et R.S. Michalski. Conceptual clustering : inventing goal-oriented classifications of structured objects. *Machine Learning, an Artificial Intelligence Approach*, éd. par R.S. Michalski et J.G. Carbonell, volume 2, pages 471–498. Morgan Kaufmann, Los Altos (CA, US), 1986.

[Sow84] J.F. Sowa. *Conceptual structures: information processing in mind and machines.* Addison-Wesley, Massachusetts (US), 1984.

[SS89] M. Schmidt-Schauß. Subsumption in KL-ONE is undecidable. *1st International Conference on Principles of Knowledge Representation and Reasoning*, éd. par R.J. Brachman, H.J. Levesque, et R. Reiter, pages 421–431. Morgan Kaufmann, 1989.

[ST88] D.T. Sannella et A. Tarlecki. Towards formal developments of programs from algebraic specifications : implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhague (DK), Août 1967.

[Str86] B. Stroustrup. An overview of C++. *ACM Sigplan Notices*, 21(10):7–18, 1986.

[SW83] D.T. Sannella et M. Wirsing. A kernel language for algebraic specification and implementation. *11th Colloquium on Foundations of Computation Theory*, éd. par M. Karpinsky, Lecture Notes in Computer Science, pages 413–427. Springer-Verlag, 1983.

[SZ89] L.A. Stein et S.B. Zdonik. Clovers: the dynamic behavior of types and instances. Rapp. Techn. CS-89-42, Brown University, Department of Computer Science, Novembre 1989.

[Tay95] N. Tayar. Gestion des versions pour la construction incrémentale et concourante de bases de connaissances. Thèse de doctorat (à paraître), Université Joseph Fourier, Grenoble (France), Septembre 1995.

[Tou86] D.S. Touretzky. *The Mathematical Theory of Inheritance.* Morgan Kaufman, 1986.

[TS93] C. Thieme et A. Siebes. Schema refinement and schema integration in object-oriented databases. Rapp. techn., CWI, Amsterdam (NL), 1993.

[Ull88] J. Ullman. Principles of Data Based and Knowledge Based Systems. Computer Science Press, Maryland, 1988.

[Uni83] United States Department of Defense. *Reference Manual for the Ada Programming Language ANSI/-MIL-std 1815-a*, 1983.

[VE95] P. Valtchev et J. Euzenat. Classification of concepts through products of concepts and abstract data types. *1st International Conference on Data Analysis and Ordered Structures*, pages 131–134, Paris (France), Juin 1995.

[Vil85] M. Vilain. The restricted language architecture of a hybrid representation system. *IJCAI-85*, Los Angeles (CA, US), 1985.

[Weg86] P. Wegner. Classification in object-oriented systems. *SIGPLAN Notices*, 21(10):173–182, Octobre 1986.

[Weg87] P. Wegner. The object-oriented classification paradigm. *Research Directions in Object-Oriented Programming*, éd. par Bruce Shiver et Peter Wegner, pages 479–560. MIT Press, Cambridge (MA, US), 1987.

[Wil94] J. Willamowski. Modélisation de tâches pour la résolution de problèmes en coopération système-utilisateur. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Avril 1994.

[Win85] T. Winograd. *Frame representation and the declarative/procedural controversy*, chapitre 20, pages 357–370. Readings in Knowledge Representation. Morgan Kaufmann, Los Altos (CA, US), 1985.

[Wir87] N. Wirth. *Algorithmes et structures de données.* Eyrolles, Paris (France), 1987.

[WM81] D. Weinreb et D. Moon. *LISP machine manual*, chapitre 20. Crambidge, Mass, 1981.

[Woo75] W.A. Woods. *What's in a link: foundations for semantic networks*, pages 35–82. Representation and Understanding: Studies in Cognitive Science. Academic Press, New-York (NY, US), 1975.

[Woo91] W.A. Woods. Understanding subsumption and taxonomy: a framework for progress. *Principles of Semantic Networks*, éd. par J.F. Sowa, chapitre 1, pages 45–94. Morgan Kaufmann, 1991.

[WSH77] J. Welsh, W.J. Sneeringer, et C.A.R. Hoare. Ambiguities and insecurities in Pascal. *Software Practice and Experience*, Novembre 1977.

[WZ88] P. Wegner et S.B. Zdonik. Inheritance as an incremental modification mechanism, or what *like* is and isn't *like*. *European Conference on Object-Oriented Programming*, volume 322 de *Lecture Notes in Computer Science*, pages 55–77, 1988.

[Zdo86] S.B. Zdonik. Maintaining consistency in a database with changing types. *SIGPLAN Notices*, 21(10):120–127, Octobre 1986.