T H È S E

présentée à

l'Université Paris 7 – Denis Diderot

pour obtenir le titre de

Docteur en Sciences

spécialité Informatique

Théorie, conception et réalisation d'un langage de programmation adapté à XML

soutenue par

Alain Frisch

le 13 décembre 2004, devant le jury composé de :

Monsieur	Pierre-Louis Curien	*Président*
Monsieur	Giuseppe Castagna	*Directeur de thèse*
Messieurs	Giorgio Ghelli	
	Martin Odersky	*Rapporteurs*
Madame	Mariangiola Dezani	*Examinatrice*
Monsieur	Xavier Leroy	*Examinateur*

Résumé

Cette thèse décrit les fondements théoriques d'un langage de programmation fonctionnel d'ordre supérieur, typé, adapté à la manipulation de documents XML. La première partie présente les bases sémantiques : algèbre de types avec types récursifs, combinaisons booléennes et constructeurs flèche et produit ; définition d'une relation de sous-typage sémantique en passant par une notion de modèle ensembliste des types ; présentation du noyau fonctionnel du langage, en particulier son système de types et sa sémantique dynamique dirigée par les types. La deuxième partie étudie les aspects algorithmiques : calcul de la relation de sous-typage et compilation optimisée du filtrage par motifs. La troisième partie présente le langage ℂDuce, construit au dessus du noyau fonctionnel, ainsi que certaines des techniques originales mises en œuvre dans son implémentation.

Abstract

This thesis describes the theoretical foundations of a type-safe and higher-order functional language, adapted to the manipulation of XML documents. The first part presents the semantical bases : type algebra with recursive types, boolean combination, arrow and product constructors ; definition of a semantic subtyping relation via a set-theoretic notion of model for types ; description of the functional kernel of the language, in particular its type system and its type-driven dynamic semantics. The second part focuses on the algorithmical aspects : computing the subtyping relation and compiling pattern matching with optimizations. The third part presents the ℂDuce language, built on top of the functional kernel, together with some of the original techniques used in its implementation.

Remerciements

Mes remerciements s'adressent, en premier lieu, à Giuseppe Castagna qui a bien voulu encadrer ma thèse et mon stage de DEA. Il m'a laissé libre de choisir les directions vers lesquelles mes travaux se sont orientés et la manière de les aborder ; il a toujours su me témoigner une grande confiance et il m'a conseillé et encouragé aux moments décisifs. Grâce à lui, ces années de thèse ont été un grand enrichissement pour moi, et j'ai pris goût à la recherche. Je remercie aussi Véronique Benzaken qui a accompagné mes travaux de thèse.

Giorgio Ghelli et Martin Odersky m'ont fait l'honneur d'être rapporteurs de cette thèse. Je leur suis très reconnaissant d'avoir accepté cette lourde tâche, malgré l'obstacle supplémentaire de la langue. Je tiens également à remercier Mariangiola Dezani, Pierre-Louis Curien, et Xavier Leroy d'avoir accepté de participer à mon jury.

Cette thèse n'aurait pas pu aboutir sans la souplesse du Conseil Général des Technologies de l'Information et de la direction des études de Télécom Paris, qui m'ont autorisé à mener mes travaux de recherche en parallèle à une formation d'ingénieur spécialement aménagée. L'INRIA doit aussi être remercié pour m'avoir permis de consacrer mes premières semaines en son sein à finir la rédaction de la thèse.

Le langage ℂDuce, dont les fondations théoriques sont présentées dans cette thèse, est un travail d'équipe. Giuseppe Castagna et Véronique Benzaken ont participé avec moi à la définition du langage et à la rédaction de la documentation. Certaines parties de l'implémentation ont été réalisées par d'autres étudiants : Cédric Miachon pour le langage de requête CQL intégré à ℂDuce, Stefano Zacchiroli pour le support de XML Schema, et Julien Demouth pour l'interface avec le langage Objective Caml. Les chercheurs et les étudiants qui ont contribué d'une manière ou d'une autre à ℂDuce, ainsi que la petite communauté des premiers utilisateurs, ont permis de créer une dynamique positive autour du projet.

Mes collègues de bureau à l'ENS ont le mérite de m'avoir supporté. Je tiens à les remercier pour l'ambiance sympathique de ce bureau sous les toits où il est pourtant si facile de se cogner la tête. Francesco Zappa-Nardelli a proposé le nom ℂDuce ; je n'ai pas encore décidé si je dois le remercier pour cela.

Mes parents sont responsables de mon existence, et donc en quelque sorte de celle de cette thèse. Je n'ai jamais eu l'occasion de les remercier « formellement » pour tout ce qu'ils ont fait pour moi, pour le goût du savoir et de l'effort qu'ils m'ont transmis, pour leurs encouragements et leur amour. Je les remercie donc pour tout cela. De même, je remercie mes amis, et Abbey en particulier, pour leur soutien moral et tout le reste.

Avant-propos

Ce mémoire présente mes travaux de recherche effectués au sein de l'équipe *Langages* du Département d'Informatique de l'École Normale Supérieure de Paris, sous la direction de Giuseppe Castagna (stage de DEA : avril-août 2001 ; thèse : octobre 2002 - septembre 2004). Pendant cette période, j'ai également travaillé à l'Université de Turin (une semaine), chez Microsoft Research Cambridge (trois mois), et au PSD Laboratory à Tokyo (une semaine) ; la rédaction du mémoire a été achevée au sein du projet Cristal à l'INRIA Rocquencourt. La liste des publications auxquelles j'ai contribué est donnée plus bas (certains résultats obtenus ne sont pas présentés dans ce mémoire).

Convention Le mémoire est rédigé dans un mélange libre de première personne du pluriel « nous » et d'un neutre impersonnel « on », à l'exception des chapitres d'introduction et de conclusion, où la première personne du singulier est utilisée pour donner un ton plus personnel.

Publications

[BCF02] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce : a white paper. In *Programming Languages Technologies for XML (PLAN-X)*, 2002.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce : An XML-centric general-purpose language. In *ACM International Conference on Functional Programming (ICFP)*, 2003.

[CF04] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Second workshop on Programmable Structured Documents*, 2004.

[DCFGM02] Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems (ITRS)*. Electronic Notes in Theoretical Computer Science, 2002.

[Fri01] Alain Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées : application au typage de XML, 2001. Rapport de DEA (Université Paris 7).

[Fri04] Alain Frisch. Regular tree language recognition with static information. In *3rd IFIP International Conference on Theoretical Computer Science (TCS)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop.

[FC04] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146. IEEE Computer Society Press, 2002.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *Proceedings of the 32st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005.

Table des matières

Résumé / Abstract **1**

Remerciements **3**

Avant-propos **5**

Table des figures **13**

Notations **15**

Introduction 19

1 Introduction **19**

1.1 Motivations, objectifs 19
1.1.1 Langages de programmation 19
1.1.2 XML 20
1.1.3 Programmer avec XML 21
1.2 XDuce 23
1.2.1 Présentation 23
1.2.2 Codage interne 24
1.3 Survol rapide de la thèse 26
1.3.1 Fondations théoriques et sémantiques 26
1.3.2 Aspects algorithmiques 28
1.3.3 Le langage ℂDuce 29
1.4 Contributions principales 29
1.4.1 Fonctions d'ordre supérieur et sous-typage ensembliste . . 29
1.4.2 Typage du filtrage 31
1.4.3 Compilation efficace du filtrage 32
1.4.4 Algorithme de sous-typage 32
1.4.5 Attributs XML 33
1.4.6 Un calcul pour les fonctions surchargées 34

I Fondations théoriques et sémantiques 37

2 Un cadre formel pour les syntaxes récursives 39
2.1 Motivation . . . 39
2.2 La catégorie des F-coalgèbres . . . 41
2.3 Congruences, quotients . . . 43
2.4 Régularité et récursivité . . . 44
2.4.1 Régularité . . . 45
2.4.2 Récursivité . . . 46
2.4.3 Application : transfert entre signatures . . . 48
2.5 Constructions . . . 50
2.5.1 Partage optimal . . . 51
2.5.2 Partage minimal . . . 53
2.5.3 Partage modulo renommage et extension . . . 55

3 Algèbre de types 59
3.1 Combinaisons booléennes . . . 59
3.1.1 Motivation . . . 59
3.1.2 Combinaisons booléennes finies en forme normale disjonctive 60
3.2 Algèbres avec types récursifs et combinaisons booléennes . . . 62
3.3 L'algèbre minimale . . . 63

4 Sous-typage sémantique 67
4.1 Types de base . . . 69
4.2 Modèles . . . 69
4.3 Inclusions ensemblistes . . . 71
4.4 Analyse du sous-typage . . . 73
4.5 Modèle universel . . . 75
4.6 Modèles non universels . . . 79
4.7 Conventions de notation . . . 82
4.8 Raisonnements sémantiques . . . 82
4.8.1 Décompositions . . . 83
4.8.2 Construction de modèles équivalents . . . 84
4.9 Systèmes d'équations $\vee\times$ . . . 88

5 Calcul 91
5.1 Syntaxe . . . 91
5.2 Système de types . . . 92
5.3 Propriétés syntaxiques du typage . . . 94
5.4 Valeurs . . . 96
5.5 Sémantique . . . 99
5.6 Sûreté du typage . . . 100
5.7 Inférence de types . . . 103

5.7.1 Filtres, schémas . . . 103
5.7.2 Typage des opérateurs . . . 106
5.7.3 Algorithme de typage . . . 107
5.7.4 Interprétation ensembliste des schémas . . . 109
5.8 Opérateurs . . . 110
5.9 Variante du calcul sans fonctions surchargées . . . 112

6 Filtrage **115**
6.1 Motifs . . . 115
6.2 Sémantique . . . 116
6.3 Typage . . . 116
6.4 Extension du calcul . . . 120
6.5 Normalisation . . . 123
6.5.1 Méthode . . . 123
6.5.2 Élimination de l'intersection . . . 125

II Aspects algorithmiques **127**

7 Algorithme de sous-typage **129**
7.1 Calcul d'un prédicat inductif . . . 129
7.1.1 Formules, systèmes, induction . . . 129
7.1.2 Cache . . . 131
7.1.3 Suppression du retour-arrière . . . 134
7.1.4 Application : test de vide pour un automate d'arbres . . . 143
7.2 Algorithme de sous-typage . . . 143
7.3 Variante et optimisation . . . 145
7.3.1 Autre formules . . . 145
7.3.2 Approximations . . . 147

8 Compilation du filtrage **149**
8.1 Évaluation naïve . . . 149
8.2 Idées d'optimisation . . . 150
8.3 Formalisation . . . 152
8.3.1 Requêtes . . . 153
8.3.2 Résultats . . . 153
8.3.3 Compilation des requêtes . . . 154
8.3.4 Langage cible : syntaxe . . . 156
8.3.5 Langage cible : sémantique . . . 157
8.3.6 Information statique . . . 158
8.3.7 Élaboration de stratégies . . . 160
8.3.8 Exemples de constructions . . . 165
8.4 Factorisation des captures . . . 168

III Le langage CDuce **171**

9 Enregistrements **173**
9.1 Types . . . 173
9.1.1 Algèbre de types . . . 173
9.1.2 Modèles . . . 174
9.1.3 Sous-typage . . . 174
9.1.4 Décomposition, projection . . . 177
9.2 Motifs . . . 178
9.3 Calcul . . . 180
9.4 Opérateur de concaténation . . . 180
9.5 Fonctions partielles . . . 186

10 Présentation du langage **189**
10.1 Types de base, constantes . . . 189
10.2 Types, motifs . . . 190
10.3 Séquences, expressions régulières . . . 192
10.3.1 Séquences . . . 192
10.3.2 Expressions régulières de types et de motifs . . . 192
10.3.3 Traduction des expressions régulières . . . 193
10.3.4 Décompilation des types séquences . . . 195
10.3.5 Opérateurs . . . 196
10.3.6 Chaînes de caractères . . . 198
10.4 XML . . . 198
10.5 Constructions dérivées . . . 199
10.6 Traits impératifs . . . 200
10.7 Un exemple . . . 202

11 Techniques d'implémentation **205**
11.1 Typeur . . . 205
11.2 Représentation des valeurs . . . 208
11.3 Représentation des combinaisons booléennes . . . 210
11.3.1 Simplification par tautologies booléennes . . . 210
11.3.2 Atomes disjoints . . . 211
11.3.3 Représentation alternative : arbres de décision . . . 212
11.3.4 Représentation des types de base . . . 214
11.4 Interface avec Objective Caml . . . 214
11.5 Performances . . . 216

Conclusion **221**

12 Conclusion et perspectives **221**

12.1 Perspectives 222

12.2 Autour de ℂDuce 225

Table des figures

4.1 Axiomatisation du typage de l'application fonctionnelle 86

5.1 Expressions (termes) du calcul 91
5.2 Système de type 93
5.3 Algorithme de typage 108
5.4 Axiomatisation du typage de la première projection 111

6.1 Sémantique du filtrage 117

7.1 Algorithme avec cache 132
7.2 Algorithme sans retour-arrière 137

8.1 Sémantique du langage cible 158
8.2 Production d'expressions cible 160
8.3 Production de stratégies 162
8.4 Squelette de production d'expressions cible 163

9.1 Axiomatisation du typage de l'opérateur $\oplus_t$ 184

10.1 Un programme $\mathbb{C}$Duce 202

Notations

Symbole	Description	Section
$:=$	Définition d'un objet ou d'une notation	
$\mathcal{P}(X)$	Ensemble des parties de X	
$\mathcal{P}_f(X)$	Ensemble des parties finies de X	
Set	Catégorie des ensembles et des fonctions totales	
$X+Y$	Réunion disjointe de deux ensembles	
$\overline{X}^Y$	Complémentaire ensembliste de X par rapport à Y	
$\mathbb{N}$,$\mathbb{Z}$	Entiers naturels (resp. relatifs)	
$i_{X,Y}$	Inclusion canonique $X \subseteq Y$	
$i = 1..n$	$i \in \{1, \ldots, n\}$	
$\mathbf{Alg}_F$	Catégorie des F coalgèbres	2.2
$\mathcal{B}$	Foncteur « combinaisons booléennes »	3.1
$\vee$,$\wedge$,$\setminus$,$\neg$	Opérateurs booléens sur les types	3.1
$\sqsupset$	Socle	3.3
$\mathbb{T}$	Algèbre des types	3.2
T	Ensemble des nœuds de type	3.2
$\widehat{T}$	Ensemble des expressions de type	3.2
τ	Description des nœuds de type $\tau : T \to \widehat{T}$	3.2
T_A,T_u	Atomes de types (resp. atomes de types de genre u)	3.2
$\mathbb{B}$	Types de bases	3.3
$\mathcal{C}$	Constantes	4.1
$\times$,$\to$	Constructeurs de types	3.3
$\mathbb{0}, \mathbb{1}$	Type vide (resp. universel)	3.1
$\mathbb{1}_{\mathbf{prod}}$	Type produit universel $\mathbb{1}_{\mathbf{prod}} = \mathbb{1} \times \mathbb{1}$	6.5.2
$[\![\_]\!]$	Interprétation ensembliste	3.1
$\mathbb{E}[\![\_]\!]$	Interprétation extensionnelle associée à $[\![\_]\!]$	4.2
$\pi(t)$	Décomposition en réunion finie de produits	4.8
$\pi_1[t], \pi_2[t]$	Projections des types	5.8
$\mathrm{Dom}(t), \rho(t)$	Décomposition des types fonctionnels	4.8
G	Foncteur « motifs »	6.1
$\mathbb{P}$	Algèbre des motifs	6.1
P	Ensemble des nœuds de motif	6.1

Symbole	Description	Section
$\widehat{P}$	Ensemble des expressions de motif	6.1
σ	Description des nœuds de motif $\sigma P \to \widehat{P}$	6.1
$\mathbf{|}$,$\mathbf{\&}$	Constructeurs de motif (alternative et conjonction)	6.1
$\mathrm{Var}(p)$	Variables de capture du motif p	6.1
v/p	Résultat de l'application du motif p à la valeur v	6.2
Ω	Échec du filtrage, erreur de type	6.2
$\|, \oplus$	Combinaison des résultats de filtrage	6.2
$(\!\|p|\!)$,$\lceil p \rfloor$	Ensemble des valeurs acceptées par p	6.3
$\lceil p \rfloor$	Type des valeurs acceptées par p	6.3
$(t /\!\!/ p)(x)$	Résultat de l'application de p au type t	6.3
$(t / p)(x)$	Type du résultat de l'application de p au type t	6.3
$\mathcal{E}$	Ensemble des expressions du calcul	5.1
$\mathcal{O}$	Ensemble des opérateurs	5.1
app	Opérateur d'application fonctionnelle	5.1
$\mathcal{V}$	Ensemble des valeurs	5.4
$o : t_1 \mathbf{\to} t_2$	Typage axiomatique de l'opérateur o	5.2
$v \overset{o}{\rightsquigarrow} e$	Sémantique de l'opérateur o	5.5
$(o : t_1 \Longrightarrow t_2)$	Typage sémantique de l'opérateur o	5.6
$\mathbin{\mathbb{V}}$,$\mathbin{\mathbb{\Lambda}}$,$\otimes$	Constructeurs et opérateurs sur les schémas	5.7
$\mathbf{\{}\mathbb{t}\mathbf{\}}$	Filtre défini par le schéma $\mathbb{t}$	5.7
$\mathcal{F}$	Formules booléennes	7.1.1
$p \xrightarrow{t} x$	Factorisation d'une variable de capture	8.3
$\mathcal{L}$	Ensemble des étiquettes d'enregistrement	9
$\mathcal{L} \xrightarrow{c} Z$	Fonctions presque constantes de $\mathcal{L}$ dans Z	9
$\mathrm{Dom}(r)$	Domaine d'un enregistrement	9
$\mathtt{def}(r)$	Champ infiniment répété d'un enregistrement	9

Introduction

Chapitre 1

Introduction

1.1 Motivations, objectifs

L'objectif initial du travail de recherche présenté dans cette thèse était de proposer un langage de programmation fonctionnel adapté à la manipulation de documents XML [XML], avec des traits généralistes. Ce travail poursuit l'effort de recherche initié dans la thèse d'Hosoya [Hos01]. Il a débouché sur la définition du langage ℂDuce et son implémentation. Cette thèse présente les bases théoriques, sémantiques et algorithmiques, sur lesquelles reposent ce langage.

1.1.1 Langages de programmation

Lorsqu'il s'agit de développer une application informatique, une des questions à se poser est celle du choix du langage de programmation. Ce choix peut être guidé, entre autres, par les considérations suivantes :

- Puissance expressive du langage par rapport au domaine d'application considéré : le langage permet-il d'exprimer facilement les opérations et les calculs que doivent effectuer l'application ?
- Sûreté : le langage aide-t-il le programmeur à éviter les erreurs de programmation (par exemple, en mettant en œuvre des vérifications automatiques, ou en encourageant un certain style de programmation) ?
- Efficacité : les applications développées dans le langage sont-elle suffisamment rapides pour l'usage qui leur est destiné ?

Ces questions ne sont pas en général complètement indépendantes. Une plus grande puissance expressive passe en général par la présence dans le langage de constructions de haut niveau, très déclaratives. Ces constructions permettent d'exprimer de manière concise et idiomatique des opérations complexes, ce qui permet au programmeur de ne pas avoir à se soucier de problèmes de bas niveau et de se concentrer sur l'architecture et la correction de son programme. Ces constructions de haut niveau posent des questions intéressantes de compilation et d'efficacité : leur aspect déclaratif rend nécessaire la mise en place d'une stratégie d'évaluation efficace. Enfin, les techniques d'analyses automatiques utilisées pour vérifier la sûreté des programmes peuvent donner au compilateur des informations précises pour « comprendre » les programmes et ainsi les compiler plus efficacement.

Cette thèse part de l'idée qu'un langage de programmation dédié, ou au moins conçu spécifiquement pour un usage particulier, peut manipuler certains concepts spécifiques d'une manière plus adaptée qu'un langage généraliste. Une variante de ce point de vue consiste à dire qu'en étudiant ce qui est spécifique à un domaine particulier en terme de programmation, on gagne en compréhension sur ce domaine, ce qui peut inspirer des extensions des langages généralistes, ou encourager une certaine méthodologie de développement pour ce domaine.

Le domaine considéré dans cette thèse est celui des applications qui manipulent des documents XML.

1.1.2 XML

Extensible Markup Language [XML] est un format de fichier destiné à représenter sous forme textuelle des données arborescentes. On peut considérer XML comme une syntaxe concrète pour un modèle abstrait d'arbre. Il n'y a pas de consensus sur la nature exacte de ce modèle de document, mais dans une vision simplifiée, les arbres XML sont des arbres d'arité variable, dans lequel les nœuds (appelés éléments) sont étiquetés et possèdent chacun un ensemble fini d'attributs textuels, et dont les feuilles sont des caractères.

Voici un exemple de document XML :

```
<animal kind="insect">
 <name>Bumble bee</name>
 <comment>There are about <strong>19</strong> different species
          of bumblebee.</comment>
</animal>
```

Dans cet exemple, `animal`, `name`, `comment` et `strong` sont des étiquettes (*tags*). Les balises `<...>` et `</...>` indiquent respectivement le début et la fin d'un élément. Une des conditions de bonne formation des documents XML est celle du bon parenthésage de ces balises, qui permet de voir les documents comme des arbres. Dans l'exemple ci-dessus, la racine est l'élément d'étiquette `animal`, il possède deux éléments fils `name` et `comment`, ainsi que des caractères blancs. L'élément `comment` possède comme fils des caractères et un élément `strong`. La racine possède de plus un attribut `kind` dont la valeur est la chaîne de caractères `"insect"`. Dans un document XML bien formé, le même élément ne peut définir deux fois le même attribut.

La spécification XML n'attribue aucune sémantique aux documents, aux étiquettes de leurs éléments, aux attributs. Elle se contente de spécifier quelles suites de caractères représentent effectivement un arbre XML. Ce faisant, elle encourage un certain modèle de document abstrait (les arbres). Une application qui peut représenter ses données sous forme d'arbre XML peut utiliser XML comme format de stockage externe. De même, deux applications qui doivent s'échanger des données arborescentes peuvent utiliser XML comme format d'échange. Dans les deux cas, les applications ne travaillent pas sur n'importe quels documents XML, mais seulement sur une certaine classe, définie par les étiquettes autorisées pour les éléments, leurs possibles structures d'imbrication, les attributs qui doivent être définis, qui peuvent l'être ou non, et la forme de leurs valeurs le cas échéant, la présence autorisée de données textuelles (caractères) à certains endroits, ...

On peut donc distinguer deux niveaux de *correction* pour les documents XML :

- une notion de bonne formation syntaxique, définie de manière universelle par la spécification XML, et qui est la condition nécessaire et suffisante pour pouvoir interpréter un document comme un arbre XML ;
- une notion de validité par rapport à un certain nombre de contraintes sur la structure et le contenu des documents, spécifiques à une application ou à un groupe d'applications qui se mettent d'accord.

Un ensemble de contraintes de validité est appelé un schéma, ou un type XML. La validité suppose la bonne formation, puisque les contraintes portent sur le modèle abstrait (arbre) et non sur la représentation textuelle des documents XML. Le fait de séparer ces deux notions de correction permet de développer certains outils génériques, qui travaillent sur n'importe quel type de documents XML (comme des *parsers*, des éditeurs, des outils de requête, de transformations, de mise en page, ...), et de se placer à un niveau suffisamment abstrait (les arbres) pour exprimer les contraintes des schémas. Les questions syntaxiques, comme la manière d'écrire des commentaires, d'inclure des fragments externes, d'encoder les jeux de caractère, de protéger les caractères spéciaux sont réglées une fois pour toutes par la spécification XML, et les applications qui veulent utiliser XML peuvent se placer au niveau conceptuel du modèle abstrait.

Si les outils XML génériques travaillent par nature sur des documents XML arbitraires, les applications qui utilisent XML comme format de stockage ou d'échange d'information travaillent en général avec un ou plusieurs types XML. Décrire de manière formelle ces types (ou une certaine approximation des contraintes sur les documents) permet de les voir comme des spécifications non ambiguës, et d'utiliser, par exemple, des outils de validation ou d'édition qui prennent en compte les types. Il existe plusieurs langages pour décrire de manière formelle des types XML : DTD (partie intégrante de la spécification XML), XML-Schema [SCH], Relax-NG [REL], ... Dans une DTD, on spécifie les étiquettes d'éléments autorisés dans le document, et pour chaque étiquette, on indique les attributs optionnels ou obligatoire, et on contraint le contenu par une expression régulière (avec une contrainte de non-ambiguïté forte sur les expressions régulières). Dans une spécification XML-Schema, le modèle de contenu ne dépend pas seulement de l'étiquette de l'élément, mais aussi de sa position (de son contexte) dans l'arbre ; cela permet d'exprimer des contraintes plus fines. XML-Schema possède de plus une notion de « type simple », pour spécifier la forme des valeurs atomiques dans les documents (les attributs, par exemple) et la manière de les interpréter (comme un entier, un date, ...).

1.1.3 Programmer avec XML

Lorsque l'on développe une application qui manipule des documents XML, on peut se placer à différents niveaux conceptuels pour appréhender les documents :

- Niveau 0 : l'application considère les documents XML comme des fichiers textes. C'est à ce niveau que se situent les *parsers*, qui vérifient les contraintes de bonne formation des documents XML, et exposent leur structure au reste de l'application. Il est à peu près impossible de faire quelque autre traitement intéressant que ce soit à ce niveau, sans casser l'abstraction du modèle de document abstrait. Ainsi, une simple recherche

de texte dans un document XML ne peut s'effectuer à ce niveau, car le même texte peut être représenté de nombreuses manières dans un document XML.

- Niveau 1 : l'application considère les documents XML au travers d'un *parser*, mais sans prendre en compte les types XML. Celui-ci fournit une certaine interface pour accéder aux documents et à leur structure. Par exemple, DOM [DOM] permet de voir les documents XML comme des graphes, de naviguer dedans, et de les modifier en place. L'interface SAX expose à l'application un flux d'événements syntaxiques rencontrés dans le document XML (comme le début ou la fin d'un élément XML, ou une zone de texte), c'est-à-dire une suite de commandes qui permettrait de reconstruire progressivement l'arbre XML associé au document. L'application est cependant libre de ne pas construire explicitement cet arbre, et de le traiter « en flux ».
- Niveau 2 : l'application manipule des documents XML en en connaissant le type. Le langage de programmation doit permettre de suivre ces types au cours du programme, soit en traduisant d'une certaine manière les types XML en des types généralistes (langages généralistes), soit en prenant en compte directement les types XML (langages dédiés). L'application utilise le système de types du langage pour garantir certaines propriétés sur le type des documents manipulés.

Une application développée dans un langage classique peut naturellement se placer aux niveaux 0 ou 1. Si elle utilise un *parser* XML validant, celui-ci pourra se charger de vérifier que les documents en entrée de l'application sont bien du type attendu, mais cette information est alors oubliée, et l'application ne voit qu'un arbre XML générique, *a priori* quelconque, et elle ne peut pas bénéficier du fait que le document est bien typé. Il est également possible de travailler au niveau 2 avec un langage généraliste : il suffit d'établir une mise en correspondance entre les types XML et ceux du langage de programmation. C'est l'approche dite *data-binding* qui consiste à utiliser les constructions d'un langage de programmation généraliste pour représenter les données XML en fonction de leur type. Un outil externe peut par exemple prendre une spécification de types XML (DTD, XML-Schema, ...), et produire des déclarations de types (ou classes) dans le langage de programmation. Cette traduction des types introduit en général plus de structure dans les données que leur représentation XML, ce qui limite la souplesse des types XML, et elle impose en général des approximations importantes. Par exemple, on peut spécifier dans une expression régulière qu'une séquence n'est pas vide (en remplaçant une étoile de Kleene $*$ par $+$), mais si l'on veut pouvoir manipuler dans le langage des séquences potentiellement vides et des séquences nécessairement non vides avec les mêmes opérations, ces deux catégories d'objets seront représentées avec le même type de données, ce qui signifie que l'on oublie une des contraintes spécifiées par le type XML. Il est parfois possible d'utiliser des caractéristiques particulières du langage hôte (polymorphisme paramétrique, surcharge, ...) pour encoder plus finement les contraintes issues des types XML.

Une autre approche consiste à définir un nouveau langage avec un système de types conçu dès le départ pour représenter les contraintes des types XML. C'est cette approche qui est retenue dans cette thèse. Lorsque les traits de langage et le système de types adapté à XML sont suffisamment bien compris, on peut envisager d'étendre un langage existant et son système de types pour le faire

bénéficier de ces nouvelles caractéristiques.

Considérons un scénario simple d'application XML. L'application prend un document XML en entrée, et effectue certains calculs pour produire finalement un nouveau document XML. La spécification de cette application donne un type XML t_1 de documents en entrée, et un type XML t_2 en sortie. Il s'agit d'un contrat qu'elle passe avec le monde extérieur : le monde extérieur s'engage à lui fournir un document de type t_1 et elle s'engage, si c'est bien le cas, à produire un document de type t_2. En pratique, comme il n'est jamais possible de faire confiance au monde extérieur, l'application commencera par vérifier que le document qu'elle reçoit est bien de type t_1, et indiquera une erreur si ce n'est pas le cas.

Si l'application est écrite au niveau 2 ci-dessus, on peut utiliser le système de types du langage pour vérifier statiquement, au moment de compiler l'application, que la transformation vérifie bien ce contrat (que l'on peut noter $t_1 \rightarrow t_2$). En fait, si le système de types du langage n'est pas suffisamment adapté à XML pour représenter exactement les contraintes exprimées par les types XML t_1 et t_2, il sera nécessaire d'utiliser une approximation, donc de garantir un contrat moins fort, mais cela permettra tout de même d'avoir une certaine garantie statique.

1.2 XDuce

1.2.1 Présentation

Cette thèse s'inscrit dans la ligne de recherche initiée par Hosoya dans sa thèse [Hos01] et dans le projet XDuce [HP00, HVP00, HP03]. XDuce est un langage de programmation dédié à XML, construit sur une algèbre de types expression régulière, proche des langages de schéma XML classiques (DTD, XML-Schema, Relax-NG). Il repose sur :

- un codage interne [HVP00] des documents XML qui permet d'aborder les problèmes de typage avec des méthodes issues de l'algorithmique des langages réguliers d'arbre ;
- un système de types du genre DTD, mais dans lequel le contenu d'un élément n'est pas imposé uniquement par son étiquette, mais aussi par sa position, et sans la restriction de déterminisme des DTD ;
- une interprétation des types comme des ensembles de documents, qui mène naturellement à définir le sous-typage $\leq$ comme l'inclusion des ensembles dénotés (on parle de sous-typage sémantique, ou ensembliste) ;
- une opération de filtrage (*pattern-matching*) [HP01, HP02], avec une algèbre de motifs adaptée à l'extraction d'information de documents XML.

XDuce est un langage fonctionnel, dans la mesure où les programmes sont essentiellement des *expressions* à évaluer. Les résultats des calculs sont des *valeurs*, qui sont en l'occurrence des arbres XML.

Dans XDuce, les types sont interprétés comme des *ensembles* de valeurs. Cette vision des types n'est pas classique dans le monde des langages de programmation, où les types sont généralement considérés comme une spécification de la forme des valeurs, de leur représentation physique et de leur comportement. Elle est néanmoins tout à fait naturelle dans le cadre de XML : en effet, les types représentent des contraintes sur la structure des documents, mais les documents

existent indépendemment de leur type, et certaines opérations génériques ont tout leur sens sur n'importe quel type XML. Les types XML ne sont donc là que pour spécifier, voire seulement partiellement, la structure des documents XML. Il faut pouvoir voir le même document sous plusieurs types différents, car cela ne revient en fait qu'à garder en tête plus ou moins de contraintes. Le corollaire de cette vision des choses est une relation de sous-typage ensembliste : un type t_1 est décrété sous-type de t_2 si l'ensemble des valeurs de type t_1 est inclus dans l'ensemble des valeurs de type t_2, ou, autrement dit, si les contraintes exprimées par t_1 sont au moins aussi fortes que les contraintes exprimées par t_2. Le système de types du langage est alors construit de sorte à ce que le programmeur puisse utiliser une *expression* de type t_1 partout où une expression de type t_2 est attendue, et cela de manière transparente, sans devoir explicitement introduire une coercion de types.

En particulier, toute la théorie équationnelle des expressions régulières peut être utilisée de manière transparente. On peut voir une valeur de type `A+` au choix comme une valeur de type `A A*` ou de type `A * A`, c'est-à-dire isoler dans une séquence non vide soit son premier élément, soit son dernier élément. Ces trois types sont équivalents, et une fonction définie sur l'un des trois pourra être appliquée aux deux autres.

Le modèle de calcul de XDuce est fonctionnel : un programme est un ensemble de fonctions mutuellement récursives, chacun étant définie par des filtrages à base de motifs, inspirés de ceux de ML. De même qu'en ML les constructeurs de motifs correspondent aux constructeurs de types et de valeurs, en XDuce, les motifs sont des généralisations des types : ce sont des expressions régulières, avec des variables de capture. Cela encourage un style de programmation dans lequel les applications sont structurées autour du type des données XML qu'elles doivent traiter. Les fonctions récursives correspondent aux définitions de types XML récursifs, et chaque filtrage permet d'inspecter un sous-arbre (sa racine, ses fils, ...). L'opération de filtrage est un exemple d'opération déclarative, de haut niveau, que je mentionnais dans la Section 1.1.1.

Cette thèse reprend les idées de XDuce, résout certains des problèmes ouverts mentionnés dans la thèse d'Hosoya, et propose un nouveau langage, appelé $\mathbb{C}$Duce, qui étend XDuce dans un certain nombre de directions. Je parlerai très peu de XML dans cette thèse. Le lien qui existe entre XML et $\mathbb{C}$Duce est le même que celui qui existe entre XML et XDuce, et le lecteur intéressé est donc invité à se reporter aux travaux sur XDuce.

1.2.2 Codage interne

XDuce travaille avec deux représentations des documents XML. La forme externe représente les documents comme des arbres d'arité quelconque ; la forme interne utilise des arbres binaires, ce qui est plus agréable techniquement (voir également [MSV00]). Cette dernière est définie par la syntaxe (on fixe un ensemble d'étiquettes d'éléments, dont on note l un élément générique) :

$$v ::= \epsilon \mid l(v, v)$$

Ces objets représentent des *séquences* d'éléments ; la séquence vide est représentée par ϵ et $l(v_1, v_2)$ est la séquence qui commence par un élément de *tag* l, de contenu v_1, suivi par la séquence v_2.

Par exemple, le document :

```
<person>
 <name></name>
 <email></email>
</person>
```

est codé en forme interne par :

$$\texttt{person}(\texttt{name}(\epsilon, \texttt{email}(\epsilon, \epsilon)), \epsilon)$$

Les types sont des expressions qui représentent des ensembles de séquences d'éléments. Les types ont également deux représentations dans XDuce. Dans la forme externe, la syntaxe des types est :

T	::=		
		()	séquence vide
	|	X	type déclaré
	|	l[T]	élément XML
	|	T, T	concaténation
	|	T|T	réunion

Pour chaque nom de type X, on suppose donnée une déclaration de la forme :

$$\texttt{type X} = \texttt{T}$$

L'interprétation de ces types comme ensembles de séquences est claire, et il est possible de définir l'étoile de Kleene comme du sucre syntaxique dans cette représentation.

Pour la forme interne, on utilise des automates d'arbre binaire. Notons $X, X_0, X_1, \ldots$ les états, et supposons donnée pour chacun la description de ses transitions, c'est-à-dire un *type interne*. Les types de l'algèbre interne sont définis par les productions ci-dessous :

T	::=		
		$\emptyset$	type vide
	|	ϵ	singleton séquence vide
	|	$l(X, X)$	élément XML
	|	$T\|T$	réunion

Par exemple, considérons l'automate d'états X_0, X_1, X_2, défini par :

$$\begin{array}{lcl} M(X_0) & = & \epsilon \\ M(X_1) & = & \texttt{name}(X_0, X_2) \\ M(X_2) & = & \texttt{email}(X_0, X_2) \mid \epsilon \end{array}$$

Le type $\texttt{person}(X_1, X_0)$ représente les séquences constituées d'un seul élément person, qui a pour fils exactement un élément vide name, suivi d'un nombre quelconque d'éléments vides email. Autrement dit, ce type correspond à l'élément person défini par la DTD :

```
<!ELEMENT person (name,mail*)>
<!ELEMENT name   EMPTY>
<!ELEMENT email  EMPTY>
```

La forme interne des documents et des types de XDuce est assez spécifique à XML, où les étiquettes jouent un rôle particulier. En fait, la forme interne n'utilise que des caractéristiques bien connues en théorie des types : types de base singletons (pour les étiquettes et la séquence vide), types produit (pour la structure d'arbre), types récursifs et types réunion (pour coder les expressions régulières). On pourrait d'ailleurs réinterpréter la théorie sous-jacente à XDuce en termes de types produit, types réunion, et types récursifs. Au lieu de noter $l(X_1, X_2)$, on noterait $l \times X_1 \times X_2$, où l est un type de base singleton, et $\times$ est le constructeur de type produit cartésien. Le type donné en exemple ci-dessus serait écrit :

$$\texttt{person} \times (\texttt{name} \times \epsilon \times (\mu\alpha.(\texttt{email} \times \epsilon \times \alpha) \vee \epsilon)) \times \epsilon$$

où la notation $\mu\alpha.\ldots$ introduit un type récursif.

$\mathbb{C}$Duce travaille directement au niveau de la forme interne, et utilise effectivement des constructions généralistes, comme le constructeur de type produit cartésien, ou le constructeur d'enregistrement (pour représenter les attributs XML). Les notations XML ne sont vues que comme du sucre syntaxique. La Section 10.4 présente l'encodage de XML en $\mathbb{C}$Duce.

1.3 Survol rapide de la thèse

1.3.1 Fondations théoriques et sémantiques

Un des apports majeurs de $\mathbb{C}$Duce par rapport à XDuce est l'ajout des fonctions de première classe et d'un constructeur de type flèche $\rightarrow$, sans stratifier les types, de sorte à pouvoir mélanger librement types flèche et types XML. J'ai choisi de travailler au niveau de la représentation interne des types XDuce, dans laquelle les types XML sont représentés par des types réunion et produit, et des types récursifs.

J'ai donc été amené à étudier un λ-calcul avec types flèche, types produit, types réunion et types récursifs. La difficulté technique principale a été de préserver l'approche ensembliste pour définir le sous-typage. Cette approche ensembliste, qui consiste à définir le sous-typage comme une inclusion ensembliste, permet de mener de manière confortable l'étude méta-théorique du système. La démarche adoptée consiste à faire un détour par une notion de modèle purement ensembliste.

Pour pouvoir ajouter par la suite un filtrage inspiré de celui de XDuce, qui peut effectuer des tests de type à l'exécution, ce calcul possède une opération de test de type. La sémantique est donc dirigée par les types, et pour en rendre compte au niveau du typage, le calcul et l'algèbre de types permettent d'exprimer des fonctions surchargées. Intuitivement, si l'on interprète le type $t \rightarrow s$ ensemblistement comme l'ensemble des fonctions dont le résultat est de type s dès que l'argument est de type t, il est naturel de définir la surcharge des fonctions par une simple intersection. Une fonction de type $(t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2)$ est simplement une fonction surchargée, qui vérifie en même temps la contrainte du type $(t_1 \rightarrow s_1)$ et la contrainte du type $(t_2 \rightarrow s_2)$. Quitte à raisonner ensemblistement, et à introduire l'intersection pour les types flèche, il semble naturel de travailler de manière uniforme, avec des combinaisons booléennes arbitraires (réunion, intersection, différence) pour tous les constructeurs de types. Dans

XDuce, les connecteurs d'intersection et de différence sont utilisés en interne dans l'algorithme d'inférence de type pour le filtrage ; les exposer dans l'algèbre de types permet aussi de simplifier la présentation de cet algorithme.

Les quatre premiers chapitres techniques de cette thèse étudient cette algèbre de types et le calcul correspondant. Cette étude constitue les fondements théoriques du langage ℂDuce.

Le Chapitre 2 introduit un formalisme abstrait pour manipuler des algèbres de types (ou d'autres objets) récursifs. Ce formalisme permet de s'affranchir de détails d'implémentation et d'optimisation (comme le partage plus ou moins poussé de structures cycliques) ou d'artifices de présentation (représentation syntaxique avec des lieurs récursifs et les problèmes de renommage associés ; environnements de définitions récursives globales qu'il faut traîner dans tous les énoncés).

Le Chapitre 3 définit l'algèbre de types, en utilisant une représentation non syntaxique. La récursion est prise en compte par le formalisme du Chapitre 2, et les combinaisons booléennes sont représentées sous une forme qui permet d'obtenir une présentation agréable des algorithmes. Cette représentation est choisie de sorte que tout ensemble fini de types puisse être saturé en un ensemble fini stable par combinaison booléenne et par décomposition des constructeurs (flèche et produit).

Le Chapitre 4 s'attaque à la définition d'une relation de sous-typage ensembliste pour l'algèbre de types. Le défi posé par les fonctions d'ordre supérieur provient de la circularité que l'on introduit entre le système de types, la relation de sous-typage et la définition ensembliste des types. Dans XDuce, il est possible de donner directement une sémantique ensembliste des types comme des ensembles de valeurs, car les types ne sont rien d'autre que des automates d'arbre. Le sous-typage correspond alors simplement à l'inclusion de langages réguliers d'arbre, et on peut l'utiliser pour typer le calcul lui-même. Dans ℂDuce, les valeurs et les expressions du calcul sont intimement liées ; en effet, les fonctions sont des valeurs, mais pour savoir si une fonction est dans l'interprétation d'un type, il est nécessaire de disposer de l'intégralité du système de types (pour pouvoir typer le corps de la fonction).

Je lève cette circularité en introduisant une notion de modèle ensembliste de l'algèbre de types, qui permet de considérer des interprétations ensemblistes avant même d'introduire le calcul (et son système de types). Cette définition de modèle capture l'intuition que l'on a du comportement des fonctions (en fait, des types flèche), sans figer la nature concrète des éléments du modèle. Les conditions pour qu'une interprétation ensembliste soit un modèle doivent capturer les propriétés sémantiques du calcul qui ont de l'importance au niveau du typage. Tous les raisonnements sur la relation de sous-typage se font de manière sémantique en travaillant dans des modèles, et cela permet de dissocier les aspects algorithmiques liés au calcul de cette relation et son utilisation dans le système de types. En particulier, toute l'étude méta-théorique de la relation de sous-typage et des opérateurs associés peut se faire sans considérer les règles très complexes de l'algorithme de sous-typage.

Le Chapitre 5 introduit enfin le calcul à proprement parler, son système de types (qui utilise la relation de sous-typage), et sa sémantique (qui est dirigée par les types). Outre le résultat classique de sûreté (préservation du typage par réduction et absence d'erreur de type à l'exécution), le théorème principal (Théorème 5.6) est que l'ensemble des valeurs du calcul est bien un modèle,

au sens de la définition du Chapitre 4, et qu'il induit la même relation de sous-typage que celle qui a été utilisée pour définir le système de type. Autrement dit, la boucle est bouclée, car la relation de sous-typage correspond bien *a posteriori* à l'inclusion des ensembles de valeurs dénotés par les types.

Le Chapitre 6 ajoute au calcul une opération de filtrage à base de motifs, qui reprend l'opération correspondante de XDuce. En fait, cette opération généralise le test de type dynamique présent dans le calcul, et elle ne change donc pas fondamentalement le typage du calcul ou les propriétés de sa sémantique. Les motifs sont, comme les types, des objets récursifs, et le formalisme du Chapitre 2 est de nouveau utilisé pour définir l'algèbre des motifs. L'algèbre de motifs de XDuce est étendue avec des motifs conjonction ou intersection (qui ne changent pas le pouvoir expressif, mais permettent d'écrire des motifs plus compacts), des motifs constante (qui acceptent n'importe quelle valeur et renvoient une liaison constante pour une certaine variable). De plus, la condition de linéarité des variables de capture est assouplie, ce qui permet d'encoder des variables de capture de sous-séquence (dans une expression régulière) qui peuvent être repétées, c'est-à-dire apparaître plusieurs fois ou sous une étoile de Kleene ; la sémantique est alors de concaténer toutes les sous-séquences capturées. J'obtiens un algorithme de typage exact, y compris pour les variables de capture de séquence en position non-terminale, ce qui résout un problème ouvert de la thèse d'Hosoya (problème qu'il a résolu depuis, indépendemment, et avec une technique assez proche, voir la Section 1.4.2).

1.3.2 Aspects algorithmiques

Une des contributions majeures du projet XDuce est la mise au point d'un algorithme de sous-typage efficace en pratique, même s'il correspond au problème algorithmiquement complexe de l'inclusion de langages réguliers d'arbre.

Le Chapitre 7 étudie l'aspect algorithmique de la relation de sous-typage introduite au Chapitre 4, pour une certaine classe de modèles dits universels (ce sont ceux qui induisent la plus grande relation de sous-typage). Cette relation de sous-typage peut être vue comme une généralisation de celle de XDuce. En fait, l'algorithme de sous-typage a été bien préparé au Chapitre 4, où la relation de sous-typage dans les modèles universels est caractérisée de manière coinductive (via une notion de simulation). Le complémentaire de cette relation est donc simplement un prédicat inductif, et nous présentons de manière autonome deux algorithmes de calcul d'un prédicat inductif défini par des formules logiques. Le premier reprend le principe utilisé dans l'algorithme de XDuce, qui s'appuie sur un principe de *memoization*, classique dans les algorithmes de sous-typage avec types récursifs. L'algorithme présenté traite soigneusement la *memoization* pour éviter d'annuler inutilement des calculs. Je donne un autre algorithme, qui évite complètement le retour-arrière, en gardant trace des dépendances entre les hypothèses non encore vérifiées de manière sûre et les conséquences déduites. L'algorithme de sous-typage lui-même est obtenu en utilisant l'un ou l'autre de ces algorithmes, et des formules logiques directement inspirées de la notion de simulation du Chapitre 4. Je présente aussi des formules alternatives et des optimisations. Le fait d'avoir présenté le calcul de prédicat inductif comme un moteur de résolution séparé des formules logiques permet de bien découpler les problèmes et de ne pas refaire les preuves inutilement lorsque l'on considère ces variantes.

La Chapitre 8 considère l'évaluation effective de l'opération de filtrage. Du fait de la récursivité des motifs et des types, l'opération de filtrage s'apparente plus à l'évaluation d'un automate d'arbre qu'à un filtrage à la ML qui ne fait qu'extraire des sous-composantes des valeurs à profondeur finie. Je me place dans le cadre de la compilation, où il est acceptable de faire, à la compilation, des calculs éventuellement coûteux sur les motifs, pour préparer une évaluation efficace de ceux-ci lors de l'exécution. Je présente un cadre assez général pour exprimer plusieurs stratégies d'évaluations et d'optimisations. Je m'intéresse en particulier aux optimisations rendues possibles par les informations de types sur les valeurs filtrées, données par le système de types statique du langage.

1.3.3 Le langage ℂDuce

Le langage ℂDuce est défini au dessus du calcul du Chapitre 5. Je ne donne pas la syntaxe concrète du langage, ni même une description exhaustive. Je me contente d'indiquer comment les principaux traits du langage peuvent être exprimés dans le cadre du calcul.

Le Chapitre 9 montre comment étendre le calcul, l'algèbre de types et celle de motifs avec des enregistrements extensibles. Ces enregistrements permettent de coder dans ℂDuce les ensembles d'attributs d'un élément XML. Il faut bien entendu étendre l'approche ensembliste pour le sous-typage, et le typage d'un opérateur de concaténation donne lieu à un développement sémantique non trivial, pour caractériser sémantiquement et de manière exacte l'approximation que l'on fait dans son typage.

Le Chapitre 10 présente rapidement le langage, en particulier comment les types et motifs expression régulière sont encodés dans les algèbres correspondantes du calcul.

Le Chapitre 11 donne quelques aspects plus pratiques de l'implémentation de ℂDuce, comme la manière d'implémenter le typeur pour obtenir des messages d'erreurs bien localisés, des techniques pour représenter des valeurs à l'exécution pour optimiser certaines opérations, des variantes de représentation des combinaisons booléennes en interne dans le typeur, ou un système d'interface (qui préserve les types) avec le langage Objective Caml.

1.4 Contributions principales

Les travaux sur XDuce constituent le point de départ de la recherche présentée dans cette thèse. J'ai proposé des solutions à certains des problèmes ouverts présentés dans la thèse d'Hosoya, ainsi que des extensions de la théorie.

1.4.1 Fonctions d'ordre supérieur et sous-typage ensembliste

À l'origine de cette thèse est la constatation de l'absence des fonctions de première classe dans XDuce, qui ressemble pourtant, au moins superficiellement, à un langage fonctionnel de la famille ML. Disposer des fonctions de première classe, qui peuvent être passées en argument à d'autres fonctions, renvoyées comme résultat, ou stockées dans des structures de données permettrait

par exemple de définir des transformations XML paramétrées par des transformations élémentaires sur certaines parties des documents, ou de factoriser des transformations similaires. Par exemple, si `AddrBook` est un schéma XML utilisé pour représenter un carnet d'adresses, constitué disons d'éléments de type `Person` et d'éléments de type `Company`, on peut vouloir écrire une fonction générique de mise en page (`Person→Html`)→(`Company→Html`)→`AddrBook→Html`, où `Html` est le type des fragments de documents XHTML. Cette fonction générique prend en argument deux fonctions de mise en page pour les entrées de base, et renvoie une fonction de mise en page pour les carnets d'adresses. On peut donc réutiliser cette fonction générique pour créer plusieurs présentations différentes.

Outre cette utilisation des fonctions d'ordre supérieur dans le cadre des applications XML, il semble intéressant, d'un point de vue théorique, de voir comment interagissent l'interprétation ensembliste des types dans XDuce et l'interprétation des types que fait le λ-calcul.

Nous avons choisi d'intégrer complètement les types flèche au reste de l'algèbre de types, ce qui permet d'utiliser des combinaisons booléennes de types flèche. D'autres travaux ont adopté une approche plus légère pour intégrer les types expression régulière de XDuce dans un langage avec fonctions d'ordre supérieur ou objets. Le langage XHaskell [LS04b] est une extension de Haskell avec les types expression régulière de XDuce. L'algèbre de types est stratifiée : les types flèche ne peuvent pas apparaître à l'intérieur des expressions régulières, ce qui permet de définir le sous-typage d'une manière purement axiomatique pour ces types flèche. Le langage Xtatic [GP03] étend le langage orienté-objet C$^\#$ avec des types expression régulière. Dans Xtatic, les types expression régulière et les types d'objets (classes) sont stratifiés mais mutuellement récursifs. La définition de la relation de sous-typage reste cependant facile dans la mesure où le typage des objets est un typage par nom, avec sous-typage explicitement déclaré (héritage).

Damm [Dam94] étudie une algèbre de types avec types réunion, intersection, récursion, et les constructeurs produit et flèche. Cette algèbre est très proche de celle considérée dans cette thèse, mais elle n'a pas de connecteur différence ensembliste (dont nous avons besoin pour typer précisément le filtrage). Damm utilise également une approche sémantique, en introduisant deux interprétations des types : une interprétation sémantique des types dans un modèle à idéaux (les idéaux sont stables par réunion et intersection, mais pas par complémentaire), et un encodage des types comme des langages réguliers d'arbre. En mettant en relation ces deux interprétations des types, il ramène le problème de sous-typage coté sémantique à celui de l'inclusion de langages réguliers d'arbre. Cette démarche utilise des outils de sémantique dénotationnelle (domaines, espaces métriques complets), alors que l'approche présentée dans cette thèse reste purement ensembliste et se combine bien avec une sémantique opérationnelle pour le calcul. Dans la mesure où la sémantique de ℂDuce est dirigée par les types, il n'est pas clair de voir comment l'approche dénotationnelle de Damm pourrait être utilisée dans ce cadre. En effet, il existe plusieurs relations de sous-typage différentes que l'on peut déduire à partir de modèles ; les calculs construits sur ces relations auront une sémantique différente, et l'on voit donc bien qu'il n'est pas possible de partir comme Damm le fait d'une sémantique *a priori* pour définir le sous-typage.

Pour se ramener à des langages réguliers, Damm encode un type flèche

comme un ensemble de séquences qui représentent tous les graphes possibles pour des approximations finies des fonctions de ce type. Nous évitons cet encodage et ces approximations en interprétant directement un type flèche comme un ensemble de graphes (Définition 4.2). Cette interprétation plus directe permet, via des calculs ensemblistes simples, de déduire la règle de l'algorithme de sous-typage pour les combinaisons booléennes de types flèche (dans un modèle universel). Cela permet d'étendre l'algorithme de sous-typage de XDuce, qui a fait les preuves de son efficacité.

Vouillon et Melliès [VM04] présentent une généralisation du modèle à idéaux pour des types récursifs polymorphes. Leur objectif est d'interpréter les types comme des ensembles de termes de leur calcul (et non pas de valeurs), c'est-à-dire de classifier les termes en fonction de leurs types. Ils ne s'intéressent pas à l'étude effective de la relation de sous-typage. Le connecteur de type réunion est une approximation (clôture) par rapport à la réunion ensembliste. Une étude plus appronfondie des liens existant entre le système de Vouillon et Melliès et celui présenté dans cette thèse reste à mener.

1.4.2 Typage du filtrage

L'algorithme d'inférence de types pour le filtrage, présenté dans la thèse d'Hosoya [Hos01], ne donne pas un type exact pour des variables de capture de séquence qui ne sont pas en position terminale dans un motif expression régulière. Cela provient de la traduction des variables de capture de la syntaxe externe (expressions régulières) dans l'algèbre de motifs interne (automates). Dans la traduction de XDuce, une variable de capture de séquence donne lieu à deux variables dans la représentation interne, pour représenter le début et la fin de la sous-séquence capturée. La dépendance entre ces deux variables est perdue et cela oblige à inférer un type éventuellement trop large pour la variable de capture. J'ai résolu ce problème en adoptant une traduction différente des variables de capture de séquence dans les expressions régulières : elles sont propagées sur tous les éléments qu'elles capturent (et non plus seulement un marqueur de début et un marqueur de fin) dans l'expression régulière, en utilisant le même nom de variable. Cela permet de conserver la dépendance entre toutes ces variables. La sémantique des motifs internes est définie de sorte que cette traduction fournisse la bonne sémantique de capture : lorsque l'on applique un motif (q_1, q_2) à une valeur (v_1, v_2), et que chaque q_i capture une valeur v'_i pour la même variable x, le résultat pour x dans le motif (q_1, q_2) est (v'_1, v'_2).

Hosoya a résolu ce problème indépendamment. Il propose en fait une autre sémantique [Hos03] pour les motifs, qui rejette les motifs ambigus (ce qui simplifie les choses en évitant de devoir introduire des différences pour prendre en compte la politique *first-match* de l'alternation |). L'algorithme d'inférence qu'il propose dans ce cadre est exact (pour les motifs non ambigus). La technique consiste à utiliser une notion d'automate dont les transitions sont annotées par des variables de captures (qui enregistrent les éléments qui activent ces transitions). La traduction des motifs expression régulière dans les automates propage, comme je le fais, les variables de capture jusqu'aux transitions. Les deux solutions sont donc très semblables, et celle d'Hosoya pourrait s'adapter facilement aux motifs ambigus et à la politique *first-match*.

1.4.3 Compilation efficace du filtrage

La thèse d'Hosoya évoque une technique d'optimisation de l'évaluation des motifs, qui permet dans certains cas de ne considérer que l'étiquette d'un élément XML au lieu de considérer tout le sous-arbre correspondant, en utilisant les informations fournies par le système de types du langage. Je présente dans cette thèse des techniques efficaces d'évaluation des motifs qui permettent d'exprimer ce genre d'optimisations. Le formalisme introduit permet de garder trace précisément des informations de types (celles fournies par le système statique, et celles obtenues au cours des calculs), et d'exprimer des calculs en parallèle, ce qui permet d'éviter des passages multiples sur la valeur filtrée.

Levin et Pierce [Lev03, LP04] ont étudié en parallèle des méthodes semblables d'évaluation efficace des motifs dans des langages du style XDuce, en prenant en compte les informations de type. Ils introduisent une notion d'automate de filtrage (*matching automata*) qui pourrait servir de langage cible pour l'algorithme de compilation que je présente. Leur algorithme d'optimisation dirigé par les types [LP04] n'étant pas décrit formellement, il est difficile de le comparer à celui que je propose.

Dans un travail précédent [Fri04], j'ai présenté un algorithme d'optimisation pour un filtrage sans variables de capture. Le langage cible (automates non uniformes) impose un ordre de parcours séquentiel gauche-droite des arbres, et l'algorithme correspond ainsi à la stratégie gauche-droite présentée dans cette thèse (Section 8.3.8). Il possède de plus une propriété d'invariance lorsque les motifs (en fait, des types, puisqu'il n'y a pas de variables) sont remplacés par des motifs équivalents ; cela provient d'une décomposition canonique des types sous la forme d'une réunion minimale de types produit dont les premières composantes sont disjointes. Cet ingrédient pourrait s'intégrer au formalisme de cette thèse si l'on modifie la mise en forme normale des motifs (Section 6.5.2) pour utiliser une telle décomposition canonique, au moins pour les types (pour traiter les motifs avec capture, en toute généralité, il faudrait détecter l'équivalence de motifs, et je n'ai pas étudié ce problème ; on peut néanmoins se contenter de détecter certaines équivalences).

1.4.4 Algorithme de sous-typage

Amadio et Cardelli [AC93] ont proposé un algorithme de sous-typage dans un système avec types flèche et types récursifs. L'algorithme proposé est une variante de celui sans type récursifs ; il opère en gardant la trace (*memoization*) des paires de types rencontrées dans la descente récursive. Cette méthode est utilisée dans la plupart des travaux sur les types récursifs pour assurer la terminaison des algorithmes. L'aspect coinductif de l'algorithme d'Amadio et Cardelli a été souligné par Brandt et Henglein [BH97]. Kozen, Palsbergy et Schwartzbachy [KPMI95] proposent une méthode à base d'automate fini pour optimiser la complexité de l'algorithme d'Amadio et Cardelli. La constatation de base est que si l'on considère le constructeur produit $\times$ (covariant en ses deux arguments) au lieu du constructeur flèche (contravariant en son premier argument), l'algorithme d'Amadio et Cardelli est en fait un algorithme classique d'inclusion entre automates finis. Pour tenir compte de la contravariance de la flèche, il suffit d'adapter cet algorithme pour inverser l'ordre des arguments du coté contravariant. Gapeyev et al. [GLP00] font le point sur ces travaux.

L'algorithme de sous-typage de XDuce peut-être vu comme une extension de l'algorithme d'Amadio et Cardelli. Il reprend l'approche coinductive, implémentée par un ensemble d'hypothèses en train d'être vérifiées, et qu'il s'agit de saturer pour faire apparaître des contradictions éventuelles. À cause du connecteur de type réunion, l'algorithme de XDuce ne peut pas se contenter de saturer de manière monotone cet ensemble d'hypothèses. En effet, l'algorithme, présenté sous forme de règles, peut devoir essayer plusieurs dérivations différentes pour prouver un but donné (il suffit que l'une d'entre elle réussisse). Si l'algorithme essaie une branche et que celle-ci échoue, il faut procéder à un retour-arrière (*backtracking*) pour remettre à son état antérieur l'ensemble des hypothèses, car certaines de hypothèses qui ont été introduites dans la branche qui échoue n'ont pas été vérifiées (en fait, au moins l'une d'entre elles est fausse).

J'ai développé un algorithme (Chapitre 7) qui évite ce retour-arrière. J'ai choisi de dissocier la présentation les règles qui définissent la relation de sous-typage elle-même et celle du moteur de résolution qui en est indépendante. L'algorithme sans retour-arrière s'applique en fait à n'importe quelle relation inductive (ou coinductive, comme le sous-typage, en en considérant la négation). Par exemple, l'algorithme de factorisation des captures (Section 8.4) rentre bien dans ce cadre. L'algorithme sans retour-arrière est finalement assez proche, dans l'esprit, des algorithmes proposés par Dowling et Gallier [DG84] pour tester la satisfiabilité de formule de Horn en temps linéaire.

Tadahiro et Hosoya [SH04] ont abordé en parallèle le même problème (élimination du retour-arrière dans l'algorithme de sous-typage de XDuce). Leur solution consiste, comme la mienne, à garder trace des dépendances entre les hypothèses en cours de vérification et leurs conséquences. Une comparaison précise entre les deux algorithmes reste à mener.

1.4.5 Attributs XML

Lorsque ce travail de thèse a débuté, XDuce ne traitait pas les attributs XML. J'ai introduit dans ℂDuce les enregistrements extensibles (Chapitre 9) pour combler ce manque. La version présentée dans cette thèse est légèrement plus expressive que celle implémentée effectivement dans ℂDuce, en cela qu'elle permet de contraindre le type de tous les champs non spécifiés (la version implémentée permet seulement d'autoriser ou d'interdire la présence de champs hors de ceux spécifiés). Elle est aussi différente en cela que l'absence d'un champ est encodée par une valeur particulière, qui peut être manipulée et capturée par filtrage. Dans l'implémentation, cette valeur est cachée au programmeur.

Parallèlement, Hosoya et Murata [HM02, HM03] ont développé une solution alternative pour ajouter les attributs à XDuce, qui consiste à ajouter aux expressions régulières les spécifications d'attributs. Dans ce système, on peut facilement exprimer qu'une information est présente soit dans un sous-élément, soit dans un attribut, de manière exclusive, ou d'autres dépendances élément-attribut similaires. Ces dépendances peuvent être « développées », pour séparer les contraintes sur les sous-éléments des contraintes sur les attributs, en listant tous les cas possibles, au prix d'une croissance exponentielle de la taille des types. Hosoya et Murata ont développé des algorithmes spécifiques pour calculer la relation de sous-typage et les combinaisons booléennes, qui évite en pratique cette explosion combinatoire. Ainsi, si le pouvoir expressif des deux approches est comparable, celle d'Hosoya et Murata possède un avantage algorithmique

certain. Celle proposée dans cette thèse, à base d'enregistrements, a cependant l'avantage de la simplicité ; elle est complètement orthogonale au reste du système (les enregistrements sont en fait une généralisation des produits cartésiens) et contrairement à l'approche d'Hosoya et Murata, elle ne remet pas en cause, par exemple, les techniques de compilation efficace du filtrage, et ne complique pas significativement l'algorithme de sous-typage ou celui de typage du filtrage.

1.4.6 Un calcul pour les fonctions surchargées

Dans XDuce et dans ℂDuce, les calculs se font essentiellement par filtrage : c'est l'unique moyen de diriger les calculs suivant la forme des objets. Chaque branche est donnée par un motif et un corps. Le motif vérifie que l'argument est d'un certain type, et extrait également des sous-objets de l'argument, qui seront utilisés pour le calcul dans le corps. En fait, à chaque motif p, on peut associer un type $⌊p⌋$ qui représente l'ensemble des valeurs qui sont acceptées par ce motif : une valeur est acceptée si et seulement si elle est de type $⌊p⌋$. Autrement dit, chaque branche traite un type donné, et la sélection de la branche peut se faire uniquement sur le type de la valeur filtrée. En cas d'ambiguïté, le choix classique dans les langages de programmation est de choisir la première branche qui convient : c'est ce qui est fait dans les langages de type ML, dans XDuce, et dans ℂDuce.

Le filtrage est assez proche, opérationnellement, des calculs basés sur les fonctions surchargées avec liaison tardive, qui sont également des calculs dirigés par les types. Dans le λ&-calcul [CGL95], une fonction surchargée est définie par plusieurs branches, qui sont des fonctions « simples ». Lorsqu'on l'applique à une valeur, un mécanisme de sélection (*dispatch*) sélectionne la branche adéquate. Lorsque plusieurs branches sauraient traiter l'argument, la plus précise, celle dont le domaine est le plus petit, est choisie, et le typage assure qu'une telle branche existe.

La différence dans le mécanisme de sélection lorsque plusieurs branches conviennent n'est pas essentielle pour notre propos.

Par contre, il y a une différence fondamentale entre le filtrage à la XDuce et la surcharge à la λ&-calcul. Dans le filtrage toutes les branches doivent donner un résultat du même type, alors qu'avec la surcharge on peut avoir des branches qui opèrent sur des domaines disjoints ; dans ce cas, les résultats de ces branches peuvent avoir des types différents, et cette information plus précise peut être exploitée pour typer le reste du programme. Le filtrage est en ce sens moins flexible.

En ℂDuce, les fonctions surchargées permettent de conserver ce genre d'information dans le type des fonctions. Il n'y a pas de nouveau constructeur de type, et la surcharge est simplement représentée par un type intersection. Le type $(t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)$ dénote les fonctions qui ont simultanément tous les types $t_i \to s_i$. Le type intersection s'interprète effectivement comme une conjonction de contraintes (donc comme l'intersection des ensembles dénotés).

Donnons un exemple très simple d'utilisation des fonctions surchargées. Supposons qu'une entreprise reçoive de ses clients des documents XML qui sont des suites de commandes, et qu'il y a deux types de commandes t_1, t_2 (par exemple, pour des types de produits différents). Elle renvoie un accusé de réception qui reprend chaque commande, et, y ajoute des informations suivant le type. À une commande de type t_i, elle répond par un fragment XML de type s_i. Sans

fonction surchargée, la fonction de traitement a le type $(t_1 \vee t_2)* \rightarrow (s_1 \vee s_2)*$ (on utilise ici librement l'étoile de Kleene des expressions régulières). Elle procède en itérant simplement une fonction de réponse pour les commandes individuelles, de type $(t_1 \vee t_2) \rightarrow (s_1 \vee s_2)$. En fait, c'est une fonction surchargée, qui a le type : $(t_1 \rightarrow s_1) \wedge (t_2 \rightarrow s_2)$, et on peut donc donner à la transformation globale (sans récrire son code), le type plus précis : $((t_1 \vee t_2) * \rightarrow (s_1 \vee s_2)*) \wedge (t_1 * \rightarrow s_1*) \wedge (t_2 * \rightarrow s_2*)$. Ainsi, si un client n'envoie que des commandes de type t_1, donc un bon de commande de type t_1*, il peut supposer que ce qu'il reçoit est de type s_1* ; un client qui envoie des commandes des deux types doit pouvoir recevoir des réponses $(s_1 \vee s_2)*$. Si l'on ajoute un nouveau type de commandes, il suffit de modifier le code la fonction de traitement des commandes pour qu'elle puisse traiter un nouveau type (on change également l'interface des fonctions pour préciser les nouveaux types gérés).

On voit que les fonctions surchargées permettent de travailler simultanément sur des documents ayant des types plus ou moins différents, sans perdre d'information. Cela évite de devoir dupliquer du code.

Revenons un instant sur le $\lambda\&$-calcul. Dans cette théorie des fonctions surchargées, une des conditions de bonne formation des fonctions surchargées est la suivante. Si une fonction surchargée a deux branches de types $t_1 \rightarrow s_1$ et $t_2 \rightarrow s_2$, alors : $t_1 \leq t_2 \Rightarrow s_1 \leq s_2$. Pour comprendre cette condition, il faut se rappeler que le type le plus précis d'un programme diminue lors de l'évaluation. Si, statiquement, on attribue à une expression le type t_2 et que par conséquent on déduit que le résultat de la fonction surchargée appliquée à cette expression a le type s_2, il est possible qu'à l'exécution, on se rende compte que l'expression a en fait le type plus précis t_1, et c'est donc la branche $t_1 \rightarrow s_1$ qui est utilisée, donnant un résultat de type s_1. Pour que le typage soit sûr, il faut que $s_1 \leq s_2$.

Dans $\mathbb{C}$Duce, aucune condition ne restreint les types flèche donnés dans l'interface d'une fonction. Considérons les deux expressions :

$$\begin{aligned} e_0 &= \mu f(t_1 \rightarrow s_1; t_2 \rightarrow s_2).\lambda x.e \\ e'_0 &= \mu f(t_1 \rightarrow (s_1 \wedge s_2); t_2 \rightarrow s_2).\lambda x.e \end{aligned}$$

Supposons $t_1 \leq t_2$. La fonction e'_0 vérifie bien la condition du $\lambda\&$-calcul, puisque $s_1 \wedge s_2 \leq s_2$, mais ce n'est pas forcément le cas pour e_0 (si $s_1 \not\leq s_2$). Avec le système de types du Chapitre 5, cependant, l'une des expressions e_0,e'_0 est bien typée si et seulement si l'autre l'est, et elles ont alors exactement les mêmes types. Plus généralement, il est toujours possible de récrire l'interface d'une abstraction de sorte que la condition de covariance du $\lambda\&$-calcul soit vérifiée. La théorie développée permet donc de retrouver la condition posée *a priori* dans le $\lambda\&$-calcul ; dans un sens, elle donne une justification sémantique à cette condition.

Première partie

Fondations théoriques et sémantiques

Chapitre 2

Un cadre formel pour les syntaxes récursives

Dans ce chapitre, nous allons utiliser des notions élémentaires de théorie des catégories [AL91] que nous ne rappellerons pas. On note **Set** la catégorie des ensembles et des fonctions totales. La somme disjointe de deux ensembles X et Y est notée $X + Y$. Si X est un sous-ensemble de Y, on note $i_{X,Y}$ l'inclusion canonique $X \hookrightarrow Y$. L'identité $i_{X,X}$ est notée Id_X.

2.1 Motivation

Syntaxe inductive L'ensemble des types d'un système logique ou d'un système de types est classiquement introduit comme une algèbre de termes inductifs. Par exemple, l'algèbre de types T pour un λ-calcul avec un unique type de base ι peut se définir par la syntaxe suivante :

$$t \quad ::= \quad \iota \mid t \rightarrow t$$

Une telle présentation permet de définir des opérateurs et des relations par induction sur la syntaxe des types. La construction peut être vue d'un point de vue catégorique de la manière suivante. Soit **Set** la catégorie des ensembles et F le foncteur **Set** $\rightarrow$ **Set** qui associe à un ensemble X l'ensemble des termes de la forme ι ou $x \rightarrow y$, avec $x, y \in X$. Une F-algèbre est un couple (X, f) où X est un ensemble et f une fonction $FX \rightarrow X$. L'algèbre des termes T peut bien être vue comme une F-algèbre, avec comme fonction f une bijection souvent laissée implicite. La classe des F-algèbres est naturellement munie d'une structure de catégorie et l'algèbre des types T en est un objet initial. Il serait donc possible de *définir* l'algèbre de type comme la F-algèbre initiale (ce qui la caractérise à isomorphisme près dans la catégorie des F-algèbres).

Le foncteur F représente la *signature* de l'algèbre de types. Pour un foncteur **Set** $\rightarrow$ **Set** arbitraire, on peut essayer de construire formellement une F-algèbre initiale. La technique classique consiste à partir de la fonction $\emptyset \rightarrow F\emptyset$ et à itérer le foncteur, donnant ainsi naissance à un diagramme $\emptyset \rightarrow F\emptyset \rightarrow F^2\emptyset \rightarrow \ldots$, dont on peut considérer la limite inductive. Si le foncteur commute avec cette

limite (cela est garanti si le foncteur est ω-continu), on fait de cette limite une F-algèbre initiale.

Il est tout à fait possible de prendre comme signature un foncteur moins « syntaxique[1] » que l'exemple donné en introduction, comme par exemple le foncteur ω-continu $\mathcal{P}_f : \mathbf{Set} \to \mathbf{Set}$ qui associe à un ensemble l'ensemble de ses parties finies.

Syntaxe récursive Considérons maintenant le cas des types récursifs. Une présentation classique consiste à introduire dans la syntaxe des variables de récursion α et des lieurs :

$$t \quad ::= \quad \iota \mid t \to t \mid \alpha \mid \mu\alpha.t$$

La syntaxe est assez arbitraire. On pourrait en effet en imaginer une autre qui permettrait de partager plus de sous-termes communs :

$$t \quad ::= \quad \iota \mid t \to t \mid \alpha \mid t \text{ where } \{\alpha_1 = t_1; \ldots; \alpha_n = t_n\}$$

De plus, il faut tenir compte :

- d'une restriction syntaxique pour n'autoriser que les termes sans variable libre et sans récursion mal formée (c'est-à-dire qu'une variable doit-être séparée de son lieur par au moins un constructeur), et
- d'une équivalence contextuelle engendrée par α-conversion et par déroulement (*unfolding*) : $\mu\alpha.t = t[\alpha \mapsto \mu\alpha.t]$ (éventuellement interprétée de manière coinductive).

Avoir une présentation syntaxique des types suggère de décrire les algorithmes également sous forme inductive. Par exemple, les algorithmes de sous-typage avec types récursifs sont souvent présentés par un système de règles, avec un environnement de memoization pour en assurer la terminaison. Il faut donc bien être capable de tester efficacement l'égalité de types, et cela peut poser des problèmes d'implémentation.

Ces problèmes sont encore exacerbés si l'on veut introduire des connecteurs dans l'algèbre de type, tel un opérateur d'union commutatif et associatif. En effet, les tests d'équivalence peuvent devenir très coûteux. Une solution simple consiste à n'utiliser aucune notion d'équivalence, et à travailler avec la syntaxe elle-même, en prenant en compte le déroulement et d'autres équivalences éventuelles explicitement dans les règles et les algorithmes. On perd alors la possibilité d'utiliser ces équivalences dans l'étude théorique du système, ainsi que dans l'implémentation (pour augmenter le partage des structures en mémoire).

Ce genre d'approche syntaxique tente de donner une nature inductive (termes) à des objets fondamentalement cycliques. Les algorithmes sur les types récursifs travaillent en général en *décomposant* les types. Même en cas de déroulement infini, on a la garantie de ne tomber que sur un nombre fini de types différents, et c'est cette propriété qui assure la terminaison des algorithmes via des techniques de memoization.

Il nous semble plus naturel de formaliser de manière coinductive les types récursifs et de développer la théorie sur cette formalisation. La syntaxe inductive n'a qu'un rôle superficiel, et elle peut être traitée au niveau du *parser*.

[1] Par foncteur syntaxique, nous entendons un foncteur qui associe à un ensemble X un ensemble de « termes » construits sur X.

Une première idée est de prendre simplement le dual catégorique de la construction par algèbre initiale. Pour un foncteur F donné, une F-coalgèbre est un couple (X, δ) où X est un ensemble et $\delta : X \to FX$ une fonction. Alors qu'une F-algèbre représente la manière de construire des objets, une F-coalgèbre s'intéresse aux destructeurs, c'est-à-dire à la manière de voir un objet de X comme une expression construite sur des atomes dans X. La classe des F-coalgèbres est naturellement munie d'une structure de catégorie, et on peut définir naturellement la notion de F-coalgèbre finale. On la construit par itération à partir d'un objet final $\bullet$ dans **Set** (c'est-à-dire un singleton) et de l'unique fonction $F\bullet \to \bullet$ en prenant une limite projective.

Considérons de nouveau le foncteur $F : \mathbf{Set} \to \mathbf{Set}$ qui associe à un ensemble X les termes de la forme ι et $x \to y$ avec $x, y \in X$. La F-coalgèbre finale peut être décrite comme l'ensemble des arbres binaires potentiellement infinis, avec des feuilles ι et des noeuds internes $\to$. Ces objets ne sont pas représentables finiment. On voudrait restreindre notre attention aux arbres réguliers, c'est-à-dire à ceux qui n'ont qu'un nombre fini de sous-arbres différents, et qui peuvent être vus comme le déroulement infini de types récursifs. Ces arbres réguliers peuvent être représentés concrètement sous forme de graphes, mais cette représentation n'est pas unique : plusieurs graphes représentent le même arbre régulier. Avec l'exemple du foncteur F ci-dessus, il est assez facile de tester l'égalité sur la représentation, mais ce n'est pas le cas avec des foncteurs plus compliqués, qui font intervenir par exemple l'opérateur $\mathcal{P}_f(\_)$.

L'idée est alors de dire qu'il n'est pas nécessaire d'avoir une notion d'égalité si forte. Il n'est pas nécessaire d'identifier tous les graphes qui représentent le même arbre régulier pour assurer la terminaison des algorithmes. On peut se contenter d'une notion plus faible d'égalité ; il suffit en effet d'identifier suffisamment de types, pour que, partant d'un type et en le déconstruisant itérativement, on ne rencontre qu'un nombre fini de types différents pour cette notion d'égalité. Notre approche consiste à axiomatiser cette propriété, ce qui permet de développer la théorie tout en laissant plus de flexibilité du coté de l'implémentation de l'algèbre.

2.2 La catégorie des F-coalgèbres

Definition 2.1 (F-coalgèbre) *Soit $F : \mathbf{Set} \to \mathbf{Set}$ un endofoncteur ensembliste. Une* **F-coalgèbre** *est un couple $\mathbb{T} = (T, \tau)$ où T est un ensemble et τ est une fonction $T \to FT$. On dit que T est le* **support** *de $\mathbb{T}$.*

Pour simplifier, nous allons simplement parler de coalgèbre. Il est entendu que toutes les définitions sont relatives à un foncteur F fixé.

Definition 2.2 (Catégorie des coalgèbres) *Soient $\mathbb{T} = (T, \tau)$ et $\mathbb{S} = (S, \sigma)$ deux coalgèbres. Un* **morphisme** *$\mathbb{T} \to \mathbb{S}$ est donné par une fonction ensembliste $f : T \to S$ telle que $Ff \circ \tau = \sigma \circ f$. On note encore f pour le morphisme de coalgèbres.*

$$\begin{array}{ccc} T & \xrightarrow{f} & S \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle\sigma} \\ FT & \xrightarrow[Ff]{} & FS \end{array}$$

La composition des morphismes de coalgèbre hérite de l'opération correspondante dans **Set**, *et de même pour les identités. La classe* $\mathbf{Alg}_F$ *des coalgèbres est ainsi munie d'une structure de catégorie.*

Remarque 2.3 *Le foncteur* F *induit naturellement un endofoncteur* $\tilde{F}$ *dans* $\mathbf{Alg}_F$ *par* $\tilde{F}\mathbb{T} = (FT, F\tau)$ *si* $\mathbb{T} = (T, \tau)$ *et* $\tilde{F}f = Ff$. *On voit que* τ *est un morphisme de coalgèbres* $\mathbb{T} \to \tilde{F}\mathbb{T}$, *et que le diagramme commutatif dans* **Set** *dans la définition d'un morphisme de coalgèbre peut se lire dans* $\mathbf{Alg}_F$ *également :*

$$\begin{array}{ccc} \mathbb{T} & \xrightarrow{f} & \mathbb{S} \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle\sigma} \\ \tilde{F}\mathbb{T} & \xrightarrow[Ff]{} & \tilde{F}\mathbb{S} \end{array}$$

On écrira encore F *pour ce morphisme* $\tilde{F}$.

Remarque 2.4 *Le foncteur d'oubli* $\mathbf{Alg}_F \to \mathbf{Set}$ *qui associe à une coalgèbre son support est fidèle. Ainsi, pour vérifier qu'un diagramme dans* $\mathbf{Alg}_F$ *est commutatif, on peut simplement considérer sa projection dans* **Set**.

Le lemme suivant est intéressant pour « comprendre » la catégorie $\mathbf{Alg}_F$, en particulier la manière dont elle hérite l'existence de somme amalgamées de **Set**.

Lemme 2.5 *La catégorie* $\mathbf{Alg}_F$ *est cocomplète.*

> *Preuve*: On peut montrer facilement l'existence de sommes arbitraires et de coégalisateurs, ce qui suffit à établir l'existence de colimites arbitraires. Les constructions se déduisent directement de celle de **Set**.
>
> Nous esquissons une autre preuve, qui montre comment les colimites peuvent être construites directement sur des colimites obtenues dans **Set**. Soit I une petite catégorie et D un I-diagramme dans $\mathbf{Alg}_F$, c'est-à-dire un foncteur $I \to \mathbf{Alg}_F$. On écrit $Di = \mathbb{T}_i = (T_i, \tau_i)$. La composition avec le foncteur d'oubli $\mathbf{Alg}_F \to \mathbf{Set}$ donne un I-diagramme D' dans **Set**. Comme **Set** est cocomplète, on peut considérer un D'-cocône limitant $(T, (f_i : T_i \to T)_{i\in I})$. Pour faire de T une coalgèbre, il faut définir une fonction $\tau : T \to FT$. D'après la propriété universelle des colimites, une telle fonction est définie par la donnée d'un D-cocône $(FT, (g_i : T_i \to FT)_{i\in I})$. On prend $g_i = Ff_i \circ \tau_i$. Montrons que c'est bien un cocône. Soit $u : i \to j$. Comme $Du : \mathbb{T}_i \to \mathbb{T}_j$ est un morphisme dans $\mathbf{Alg}_F$, nous avons : $\tau_j \circ Du = F(Du) \circ \tau_i$, et donc $g_j \circ Du = Ff_j \circ \tau_j \circ Du = F(f_j \circ Du) \circ \tau_i = Ff_i \circ Du = g_i$. On déduit de ce cône une fonction $\tau : T \to FT$, qui donne une coalgèbre $\mathbb{T} = (T, \tau)$. Il est alors mécanique de montrer que les f_i sont des morphismes de coalgèbres $\mathbb{T}_i \to \mathbb{T}$ et que le D-cocône $(\mathbb{T}, (f_i : \mathbb{T}_i \to \mathbb{T})_{i\in I})$ est limitant. □

Coalgèbres finies et environnements récursifs Il serait concevable de s'arrêter là, et de travailler avec des coalgèbres finies. Une telle coalgèbre peut être vue comme un système fini d'équations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ où les d_i

sont des éléments de FX avec $X = \{\alpha_1, \ldots, \alpha_n\}$. Si l'on fixe un tel système, on peut appeler « type » les éléments de FX et « variables de récursion » les éléments de X. Le système lui-même est un « environnement récursif » qui donne la définition des variables de récursion. Le formalisme doit alors se développer relativement à un environnement récursif fixé, et il faut indiquer explicitement lorsque l'on a besoin de l'étendre (dans un algorithme ou dans la preuve d'un théorème). D'un point de vue formel, un type n'a de sens que relativement à un environnement donné. Pour définir une algèbre de types, on devrait en fait définir un type comme un couple formé d'un environnement $(X, \sigma : X \to FX)$ et d'un élément de FX. On veut sûrement identifier deux types qui ne diffèrent que par leur environnement, l'un étant une extension de l'autre. Mais on peut vouloir identifier plus de types, par exemple modulo renommage des variables, déroulement des équations, ou modulo une notion d'égalité coinductive. Rendre plus de types égaux permet d'accélérer les algorithmes qui procèdent par saturation, mais détecter les équivalence peut avoir un coût important. Il y a donc un compromis à faire au niveau de l'implémentation. De plus, prendre en compte les équivalences dans le développement théorique peut s'avérer lourd techniquement.

Il est légitime de chercher un formalisme suffisamment abstrait pour permettre un développement théorique propre et qui rend néanmoins compte des différents choix possibles d'implémentation.

2.3 Congruences, quotients

Definition 2.6 *Soit $\mathbb{T} = (T, \tau)$ une coalgèbre et $\equiv$ une relation d'équivalence sur T. On note $T/\equiv$ l'ensemble quotient de cette équivalence et $\pi_\equiv : T \to T/\equiv$ la projection canonique. On dit que $\equiv$ est une* **congruence** *(sur $\mathbb{T}$) s'il existe une fonction $\tau_\equiv$ qui fait commuter le diagramme suivant :*

$$\begin{array}{ccc} T & \xrightarrow{\tau} & FT \\ \downarrow{\scriptstyle \pi_\equiv} & & \downarrow{\scriptstyle F\pi_\equiv} \\ T/\equiv & \xrightarrow[\tau_\equiv]{} & F(T/\equiv) \end{array}$$

Une telle fonction est unique si elle existe. Si $\equiv$ est une congruence, on peut donc parler de la *coalgèbre quotient $\mathbb{T}/\equiv \;=\; (T/\equiv, \tau_\equiv)$. La projection $\pi_\equiv$ est un morphisme de coalgèbres.*

Lemme 2.7 *Soit $\mathbb{T}$ une coalgèbre et $(\equiv_i)_{i\in I}$ une famille de congruences sur $\mathbb{T}$. Alors la relation d'équivalence $\equiv$ engendrée par la réunion des $\equiv_i$ est encore une congruence sur $\mathbb{T}$.*

> *Preuve*: Il s'agit de montrer que si $\alpha, \beta \in T$ et $\alpha \equiv \beta$, alors $F\pi \circ \tau(\alpha) = F\pi \circ \tau(\beta)$. Comme la relation d'équivalence $\equiv$ est engendrée par les $\equiv_i$, on peut supposer que $\alpha \equiv_i \beta$ pour un certain i. Notons π'_i la projection $T/\equiv_i \to T/\equiv$. On a : $F\pi \circ \tau = F(\pi'_i \circ \pi_{\equiv_i}) \circ \tau = F\pi'_i \circ F\pi_{\equiv_i} \circ \tau = F\pi'_i \circ \tau_{\equiv_i} \circ \pi_{\equiv_i}$. Comme $\pi_{\equiv_i}(\alpha) = \pi_{\equiv_i}(\beta)$, on en déduit $F\pi \circ \tau(\alpha) = F\pi \circ \tau(\beta)$. □

Corollaire 2.8 *Pour toute coalgèbre $\mathbb{T}$, il existe une plus grande congruence.*

Exemple 2.9 *Donnons un exemple pour appuyer l'intuition (pour le reste de ce chapitre). Soit Σ un alphabet et F le foncteur défini par $F(X) = \{0,1\} \times X^\Sigma$. On peut interpréter une coalgèbre finie $\mathbb{T} = (T, \tau)$ comme un automate déterministe fini complet sur l'alphabet Σ. Si $q \in T$ et $\tau(q) = (\epsilon, d)$, alors $\epsilon = 1$ signifie que q est un état final et $d : \Sigma \to T$ donne les transitions sortantes pour q. La plus grande congruence correspond à la notion classique d'équivalence de Nérode, et le quotient de $\mathbb{T}$ par cette congruence est l'automate déterministe minimal associé à $\mathbb{T}$.*

2.4 Régularité et récursivité

Plutôt que de définir notre algèbre de types par une propriété universelle (algèbre initiale, coalgèbre finale), nous allons axiomatiser les deux propriétés dont nous aurons besoin pour pouvoir développer la théorie. Cela laissera plus de liberté à l'implémentation. Ces deux propriétés sont la régularité et la récursivité de la coalgèbre. Elles permettent respectivement de garantir :

- la terminaison des algorithmes qui procèdent par décomposition itérative des types et mémoization ;
- l'existence de solutions à des systèmes d'équations dans l'optique de définir des types récursifs.

Pour formaliser ces deux conditions, il est commode de faire l'hypothèse suivante sur le foncteur F.

Definition 2.10 (Signature) *Un foncteur F : **Set** $\to$ **Set** préserve les inclusions ensemblistes si pour, pour des ensembles X, Y tels que $X \subseteq Y$, alors $FX \subseteq FY$ et l'image de l'inclusion canonique $i_{X,Y}$ est l'inclusion canonique $i_{FX,FY}$. On dit également que F est une* **signature**.

Exemple 2.11 *Le foncteur $\mathcal{P}_f(\_)$ préserve les inclusions ensemblistes, ainsi qu'un foncteur qui associe à un ensemble X un ensemble de termes avec des variables libres dans X (et comme cas particulier, les foncteurs constants). La composition de deux foncteurs qui préservent les inclusions ensemblistes possède encore cette propriété, et il en est de même pour la somme et le produit de deux foncteurs (définies par $(F+G)(X) = FX + GX$, et $(F \times G)(X) = FX \times GX$). On peut obtenir avec ces briques de base le foncteur de l'Exemple 2.9.*

On suppose à partir de maintenant que le foncteur F préserve les inclusions ensemblistes.

Lemme 2.12 *Soit $f : X \to Y$ une fonction. Alors : $(Ff)(FX) \subseteq F(f(X))$.*

| *Preuve*: Il suffit d'écrire $f = i_{f(X),Y} \circ f'$. □

Lemme 2.13 *Soit $f : X \to Y$ une fonction et $A \subseteq X$. Alors : $F(f_{|A}) = (Ff)_{|FA}$.*

| *Preuve*: Il suffit d'écrire $f_{|A} = f \circ i_{A,X}$. □

Definition 2.14 (Sous-coalgèbre, extension) *Soient $\mathbb{T} = (T, \tau)$ et $\mathbb{S} = (S, \sigma)$ deux coalgèbres. On dit que $\mathbb{T}$ est une* **sous-coalgèbre** *de $\mathbb{S}$, ou que $\mathbb{S}$ est une*

extension *de* $\mathbb{T}$ *si* $T \subseteq S$ *et si l'inclusion canonique* $T \hookrightarrow S$ *est un morphisme d'algèbre* $\mathbb{T} \to \mathbb{S}$. *On écrit alors* $\mathbb{T} \subseteq \mathbb{S}$. *Une* **retraction** *de* $\mathbb{S}$ *sur* $\mathbb{T}$ *est un morphisme* $f : \mathbb{S} \to \mathbb{T}$ *tel que* $f \circ i_{\mathbb{T},\mathbb{S}} = Id_{\mathbb{T}}$.

Remarque 2.15 *Les sous-coalgèbres de* $\mathbb{S} = (S, \sigma)$ *correspondent exactement aux sous-ensembles* $T \subseteq S$ *tels que* $\sigma(T) \subseteq FT$. *Avec cette correspondance, une réunion quelconque de sous-coalgèbres est encore une sous-coalgèbre.*

Remarque 2.16 *Les extensions de* $\mathbb{T} = (T, \tau)$ *correspondent exactement aux fonctions* $\sigma : X \to F(T + X)$ *où* X *est un ensemble quelconque.*

Lemme 2.17 *Soit* $f : \mathbb{T} \to \mathbb{S}$ *un morphisme de coalgèbres. Alors l'image de* f *est une sous-coalgèbre de* $\mathbb{S}$.

> *Preuve*: Il s'agit de montrer que $\sigma(f(T)) \subseteq F(f(T))$. Or $\sigma(f(T)) = Ff(\tau(T))$ car f est un morphisme de coalgèbres. On écrit ensuite : $Ff(\tau(T)) \subseteq Ff(FT) \subseteq F(f(T))$ par le Lemme 2.12. □

2.4.1 Régularité

Definition 2.18 (Base, régularité) *Soit* $\mathbb{T}$ *une coalgèbre. Une* **base** *de* $\mathbb{T}$ *est une sous-coalgèbre à support fini. Une élément du support de* $\mathbb{T}$ *est dit* **régulier** *s'il appartient à une certaine base. L'algèbre* $\mathbb{T}$ *est dite* **régulière** *si tous les éléments de son support sont réguliers.*

Lemme 2.19 *Une réunion finie de bases est une base.*

Lemme 2.20 *L'image directe d'une base par un morphisme de coalgèbres est une base.*

Lemme 2.21 *Soit* $f : \mathbb{T} \to \mathbb{S}$ *un morphisme de coalgèbres. Si* $\mathbb{T}$ *est régulière, alors son image par* f *l'est aussi. En particulier, tout quotient d'une coalgèbre régulière est régulière.*

Remarque 2.22 *Nous avons vu que les sous-coalgèbres de* $\mathbb{T} = (T, \tau)$ *peuvent être identifiées à leur support, c'est-à-dire à un sous-ensemble de* T. *Nous allons utiliser cette identification pour les bases.*

La condition de régularité peut se comprendre de la manière suivante. Soit $\mathbb{T} = (T, \tau)$ une coalgèbre régulière et t un élément de T. On peut trouver une base $S \subseteq T$ qui le contient. La décomposition $\tau(t)$ est dans FS, c'est-à-dire que c'est une expression construite immédiatement sur d'autres éléments de S. On peut continuer ainsi la décomposition et tous les éléments que l'on obtient sont encore dans S. La finitude de S permet de garantir la terminaison de ce processus de saturation.

Exemple 2.23 *Reprenons le foncteur* F *de l'Exemple 2.9. Considérons une coalgèbre* $\mathbb{T} = (T, \tau)$, *c'est-à-dire un automate (éventuellement infinie). Un état* $q \in T$ *est régulier s'il fait partie d'un sous-automate fini, c'est-a-dire si sa fermeture avant est finie.*

Lemme 2.24 *Soit* $\mathbb{T}$ *une algèbre. La sous-coalgèbre constituée des éléments réguliers est une algèbre régulière.*

2.4.2 Récursivité

Il s'agit maintenant de garantir que tout système d'équations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$, avec les d_i dans $F\{\alpha_1, \ldots, \alpha_n\}$ admet une solution. En fait, on veut aussi pouvoir utiliser dans les membres droits des équations des éléments que l'on connaît déjà dans l'algèbre.

Definition 2.25 (Extension finie, extension régulière) *Soit $\mathbb{T}$ une coalgèbre. Une extension $\mathbb{S}$ de $\mathbb{T}$ est* **finie** *si la différence ensembliste des supports est finie.*

Une extension $\mathbb{S}$ de $\mathbb{T}$ est **régulière** *si pour tout élément de $\mathbb{S}$, il existe une sous-algèbre $\mathbb{T}'$ de $\mathbb{S}$ qui contient cet élément et qui est une extension finie de $\mathbb{T}$ ($\mathbb{T} \subseteq \mathbb{T}' \subseteq \mathbb{S}$).*

Remarque 2.26 *Les extensions régulières de la coalgèbre vide sont précisément les coalgèbres régulières.*

Une manière plus concrète de voir les choses est de dire qu'une extension finie de $\mathbb{T}$ est donnée par un système fini $\{\alpha_1 = d_1; \ldots, \alpha_n = d_n\}$ où les d_i sont des éléments de $F(T+\{\alpha_1, \ldots, \alpha_n\})$, c'est-à-dire des expressions construites soit sur des variables, soit sur des éléments de T. Une retraction est une solution à ce système, c'est-à-dire un moyen de plonger les variables α_i dans T en validant les équations (c'est ce que dit le diagramme commutatif qui exprime le fait que la retraction est un morphisme de coalgèbres).

Definition 2.27 *Une coalgèbre $\mathbb{T}$ est* **récursive** *si toute extension finie admet une retraction.*

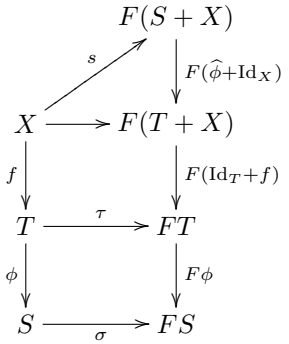
Cette définition n'impose pas d'unicité des solutions. Le même système peut avoir plusieurs solutions différentes, et cela permet de ne pas imposer à l'implémentation de devoir calculer une représentation unique des types.

Exemple 2.28 *Poursuivons l'Exemple 2.9. Si $\mathbb{T}$ est une coalgèbre récursive, alors tout automate fini peut se plonger dedans (pas forcément de manière unique). Mieux, si l'on se donne un automate fini avec des transitions vers des états de $\mathbb{T}$, on peut trouver un plongement de cet automate dans $\mathbb{T}$ qui préserve les états dans $\mathbb{T}$.*

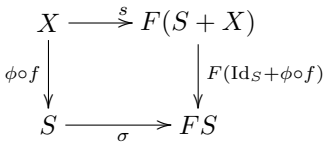
Lemme 2.29 *L'image d'une coalgèbre récursive par un morphisme est une coalgèbre récursive. En particulier, tout quotient d'une coalgèbre récursive est récursive.*

Preuve: Soit $\phi : \mathbb{T} \to \mathbb{S}$ un morphisme, avec $\mathbb{T} = (T, \tau)$ et $\mathbb{S} = (S, \sigma)$. On peut supposer que le morphisme est surjectif et il s'agit de montrer que si $\mathbb{T}$ est récursive, alors $\mathbb{S}$ l'est aussi. Soit $\widehat{\phi}$ un inverse à droite de la fonction ensembliste ϕ, c'est-à-dire une fonction $\widehat{\phi} : S \to T$ telle que $\phi \circ \widehat{\phi} = \mathrm{Id}_S$.

On considère une extension finie de $\mathbb{S}$, disons $\mathbb{S}' = (S+X, \sigma+s)$ avec $s : X \to F(S+X)$. La récursivité de $\mathbb{T}$ donne l'existence d'une fonction $f : X \to T$ qui fait commuter le diagramme :



La fonction $F(S+X) \to FS$ obtenue par composition sur le coté droit du diagramme est $F(\phi \circ \widehat{\phi} + \phi \circ f) = F(\mathrm{Id}_S + \phi \circ f)$, ce qui donne la retraction $\mathbb{S}' \to \mathbb{S}$:



□

Lemme 2.30 *Soit $\mathbb{T} = (T, \tau)$ une coalgèbre récursive. Alors la fonction $\tau : T \to FT$ est surjective.*

Preuve: Soit $t \in FT$. On considère l'extension finie $\mathbb{S} = (T + \{\bullet\}, \sigma)$ de $\mathbb{T}$ définie par $\sigma(\bullet) = t$. Une retraction de $\mathbb{S}$ sur $\mathbb{T}$ donne bien un élément de T dont l'image par τ est t. □

Ce fait montre que la coalgèbre $F\mathbb{T}$ est isomorphe à un quotient de $\mathbb{T}$ si $\mathbb{T}$ est récursive.

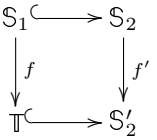
De plus, on peut choisir un inverse à droite de τ, c'est-à-dire une fonction $f : FT \to T$ telle que $\tau \circ f = \mathrm{Id}_{FT}$. Cela permet de voir T non plus comme une F-coalgèbre mais comme une F-algèbre. Cependant, le choix de l'inverse f n'est en général pas unique, donc cette vision n'a rien de canonique.

Lemme 2.31 *Soit $\mathbb{T}$ une coalgèbre récursive. Alors la coalgèbre $F\mathbb{T} = (FT, F\tau)$ est récursive. Si de plus $\mathbb{T}$ est régulière, alors $F\mathbb{T}$ l'est aussi.*

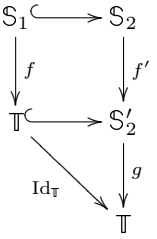
Preuve: Corollaire des Lemmes 2.21, 2.29 et 2.30. □

Lemme 2.32 (Prolongement fini) *Soit $\mathbb{T}$ une coalgèbre récursive, $\mathbb{S}_1$ une coalgèbre quelconque et $\mathbb{S}_2$ une extension finie de $\mathbb{S}_1$. Alors tout morphisme $\mathbb{S}_1 \to \mathbb{T}$ se prolonge à $\mathbb{S}_2$.*

Preuve: Soit $f : \mathbb{S}_1 \to \mathbb{T}$ un morphisme. Considérons la somme amalgamée des deux morphismes $f : \mathbb{S}_1 \to \mathbb{T}$ et $i_{\mathbb{S}_1,\mathbb{S}_2} : \mathbb{S}_1 \to \mathbb{S}_2$ (elle existe d'après le Lemme 2.5). On peut la construire de sorte à avoir une extension finie $\mathbb{S}_2'$ de $\mathbb{T}$:



La régularité de $\mathbb{T}$ donne une retraction $g : \mathbb{S}_2' \to \mathbb{T}$:



On en déduit un morphisme $g \circ f' : \mathbb{S}_2 \to \mathbb{T}$ qui prolonge f. □

Theorème 2.33 (Prolongement régulier) *Soit $\mathbb{T}$ une coalgèbre récursive, $\mathbb{S}_1$ une coalgèbre quelconque et $\mathbb{S}_2$ une extension régulière de $\mathbb{S}_1$. Alors tout morphisme $\mathbb{S}_1 \to \mathbb{T}$ se prolonge à $\mathbb{S}_2$.*

Preuve: Soit $f : \mathbb{S}_1 \to \mathbb{T}$ un morphisme. On considère l'ensemble des couples $(\mathbb{S}', f')$ où $\mathbb{S}_1 \subseteq \mathbb{S}' \subseteq \mathbb{S}_2$ et f' est un morphisme $\mathbb{S}' \to \mathbb{T}$ qui prolonge f. On munit cet ensemble de l'ordre défini par : $(\mathbb{S}'_1, f'_1) \leq (\mathbb{S}'_2, f'_2)$ ssi $\mathbb{S}'_1 \subseteq \mathbb{S}'_2$ et f'_2 prolonge f'_1. Cet ensemble ordonné est inductif, et le lemme de Zorn donne l'existence d'un élément $(\mathbb{S}_m, f_m)$ maximal. Montrons que $\mathbb{S}_m = \mathbb{S}$. Il suffit de montrer que pour toute sous-coalgèbre $\mathbb{S}'$ de $\mathbb{S}_2$ qui est une extension finie de $\mathbb{S}_1$, on a $\mathbb{S}' \subseteq \mathbb{S}_m$. Pour une telle coalgèbre, on considère la sous-coalgèbre $\mathbb{S}'' = \mathbb{S}' \cup \mathbb{S}_m$. C'est une extension finie de $\mathbb{S}_m$, et donc par le lemme précédent, le morphisme f_m se prolonge à $\mathbb{S}''$. La maximalité du couple $(\mathbb{S}_m, f_m)$ donne $\mathbb{S}_m = \mathbb{S}''$. □

Corollaire 2.34 (Retraction régulière) *Soit $\mathbb{T}$ une coalgèbre récursive et $\mathbb{S}$ une extension régulière. Alors il existe une retraction $\mathbb{S} \to \mathbb{T}$.*

Preuve: Il suffit d'appliquer le théorème précédent pour prolonger le morphisme $\mathrm{Id}_{\mathbb{T}}$ à $\mathbb{S}$. □

Ce corollaire sera utilisé dans la section suivante.

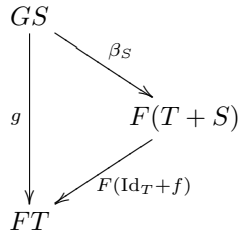
2.4.3 Application : transfert entre signatures

Dans cette section seulement, nous prenons deux signatures différentes, c'est-à-dire deux foncteurs F et G qui préservent les inclusions ensemblistes. On se donne une F-coalgèbre $\mathbb{T} = (T, \tau)$ récursive et une G-coalgèbre $\mathbb{S} = (S, \sigma)$ régulière. Il est naturel de considérer les couples de fonctions $(f : S \to T, g : GS \to FT)$ qui sont compatibles avec les structures de coalgèbres respectives, c'est-à-dire qui font commuter le diagramme suivant :

$$\begin{array}{ccc} S & \xrightarrow{\sigma} & GS \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ T & \xrightarrow[\tau]{} & FT \end{array}$$

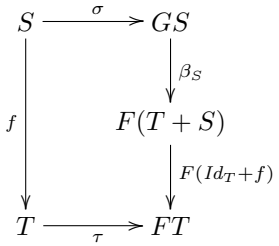
On veut contraindre ces fonctions par un système d'équations qui associe à tout élément de GS un élément de $F(T + S)$. Pour exprimer le fait que ce système ne dépend pas de la construction de $\mathbb{S}$, on suppose que la fonction $GS \to F(T + S)$ est définie comme β_S où β est une transformation naturelle $\beta : G \to F(T + \_)$. Formellement, on demande que le diagramme ci-dessous

commute :



Autrement dit, la fonction g se déduit de f. Le théorème suivant assure l'existence d'une telle fonction f.

Theorème 2.35 *Étant donnés les objets introduits ci-dessus, il existe une fonction $f : S \to T$ tel que le diagramme ci-dessous commute :*



Preuve: On dispose de la F-coalgèbre $\mathbb{T}' = (T + S, \tau + \beta_S \circ \sigma)$ qui est une extension de $\mathbb{T}$. Une retraction de cette F-coalgèbre sur $\mathbb{T}$ donnera bien une fonction $f : S \to T$ solution du système, c'est-à-dire qui vérifie $\tau \circ f = F(\mathrm{Id}_T + f) \circ \beta_S \circ \sigma$. Pour toute base B de $\mathbb{S}$, on dispose de la sous-coalgèbre de $\mathbb{T}'$ de support $T + B$, qui est une extension finie de $\mathbb{T}$. Cela résulte du diagramme de naturalité de β pour l'injection canonique $i_{B,S}$. Le Corollaire 2.34 donne bien l'existence d'une retraction $\mathbb{T}' \to \mathbb{T}$. □

Un exemple Nous donnons maintenant un exemple concret d'application du résultat précédent. Il s'agit d'une version simplifiée du Théorème 6.9. Fixons un alphabet fini Σ. Le foncteur F associe à un ensemble X l'ensemble des termes finis engendrés par la syntaxe suivante :

$$t := \quad (t \vee t) \;\mid\; (x \times x) \;\mid\; a$$

où x désigne un élément générique de X et a désigne un élément générique de Σ. On suppose donnée une F-coalgèbre récursive $\mathbb{T} = (T, \tau)$. On peut la voir comme une algèbre de types (avec types produit et types réunion).

De même, le foncteur G associe à un ensemble Y l'ensemble des termes finis engendrés par la syntaxe suivante :

$$p := \quad (p|p) \;\mid\; (y, y) \;\mid\; t$$

où y désigne un élément générique de Y et t désigne un élément générique de FT. On suppose donnée une F-coalgèbre régulière $\mathbb{S} = (S, \sigma)$. On peut la voir comme une algèbre de motifs.

On veut alors associer à tout motif un type qui représente les valeurs acceptées par p. Formellement, on veut définir deux fonctions $f : S \to T$ et $g : GS \to FT$ reliées par la relation $\tau \circ f = g \circ \sigma$, et qui vérifient de plus les équations suivantes :

$$\begin{cases} g(p_1|p_2) & = & g(p_1) \mathbin{\mathsf{V}} g(p_2) \\ g((y_1, y_2)) & = & f(y_1) \mathbin{\mathsf{\times}} f(y_2) \\ g(t) & = & t \end{cases}$$

On rentre dans le cadre du théorème ci-dessus on définissant la transformation naturelle $\beta : G \to F(T + \_)$ par :

$$\begin{cases} \beta_Y(p_1|p_2) & = & \beta_Y(p_1) \mathbin{\mathsf{V}} \beta_Y(p_2) \\ \beta_Y((y_1, y_2)) & = & y_1 \mathbin{\mathsf{\times}} y_2 \\ \beta_Y(t) & = & t \end{cases}$$

On peut alors appliquer le théorème pour définir la fonction $f : S \to T$ et la fonction g s'en déduit.

2.5 Constructions

Dans cette section, nous montrons comment construire de manière formelle des coalgèbres récursives et régulières. Nous donnons trois constructions. La première modélise une implémentation qui partage de manière optimale les types récursifs (c'est-à-dire que deux types qui possèdent le même déroulement infini seront égaux dans cette coalgèbre). La seconde au contraire modélise une implémentation qui ne cherche à créer aucun partage. La troisième est intermédiaire : elle détecte les types définis par les mêmes équations (modulo renommage des variables de récursion et suppression d'équations inutiles) et les partage.

Les constructions s'appuient sur un ensemble infini dénombrable V de variables, que l'on fixe pour la suite de cette section.

Comme pour la construction classique d'une algèbre initiale, nous aurons besoin d'une hypothèse de continuité sur le foncteur F. Compte tenu du fait que F préserve les inclusions ensemblistes, nous pouvons donner une version concrète de cette hypothèse de continuité.

Definition 2.36 (Continuité) *Une famille d'ensembles $(X_i)_{i \in I}$ est* **dirigée** *si :*

$$\forall i, j \in I.\ \exists k \in I.\ X_i \cup X_j \subseteq X_k$$

Le foncteur F est **continu** *si pour toute famille d'ensembles dirigée $(X_i)_{i \in I}$:*

$$F(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} FX_i$$

Dans toute la suite de ce chapitre, nous supposons le foncteur F continu. Une conséquence de cette hypothèse est que la classe des coalgèbres régulières est stable par extension finie.

Lemme 2.37 (Stabilité des coalgèbres régulières par extension finie) *Soit $\mathbb{T} \subseteq \mathbb{S}$ une extension finie. Si $\mathbb{T}$ est régulière, alors $\mathbb{S}$ est régulière.*

Preuve: Notons $\mathbb{T} = (T, \tau)$ et $\mathbb{S} = (S, \sigma)$. On peut écrire $S = T + X$ où X est un ensemble fini. Par continuité du foncteur F et régularité de $\mathbb{T}$, on obtient : $F(S) = F(T + X) = F(\bigcup B + X) = \bigcup F(B + X)$ où les réunions sont prises sur l'ensemble des bases de $\mathbb{T}$. Il s'agit bien d'une famille dirigée car la réunion de deux bases est encore une base. L'ensemble $\sigma(X)$ étant fini, il est possible de trouver une base $\mathbb{B}$ telle que $\sigma(X) \subseteq F(B + X)$. On a aussi, par définition d'une base : $\sigma(B) \subseteq FB \subseteq F(B + X)$. On obtient une base $\mathbb{B}'$ de $\mathbb{S}$ dont le support est $B + X$. On a montré que tous les éléments de $S \backslash T$ sont dans une base de $\mathbb{S}$. Il en va évidemment de même des éléments de T. □

Remarque 2.38 *Par une preuve similaire, on peut montrer la transitivité des extensions régulières : si $\mathbb{T}_1 \subseteq \mathbb{T}_2$ et $\mathbb{T}_2 \subseteq \mathbb{T}_3$ sont deux extensions régulières, alors $\mathbb{T}_1 \subseteq \mathbb{T}_3$ l'est également.*

2.5.1 Partage optimal

Dans cette section, nous allons construire une coalgèbre régulière et récursive avec partage optimal, c'est-à-dire qui n'a pas de quotients.

L'idée est d'introduire un type par un système fini d'équations $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ en distinguant une de ses variables α_i. On considère les couples $(\mathbb{T}, \alpha)$ où $\mathbb{T} = (T, \tau)$ est une coalgèbre finie à support dans V ($T \in \mathcal{P}_f(V)$) et $\alpha \in T$. Pour introduire le partage optimal, on définit $\simeq_D$ comme la relation d'équivalence engendrée par les équations :

$$(\mathbb{T}, \alpha) \simeq_D (\mathbb{S}, f(\alpha))$$

pour tout morphisme de coalgèbres finies $f : \mathbb{T} \to \mathbb{S}$. On note $[\mathbb{T}, \alpha]$ la classe d'équivalence de $(\mathbb{T}, \alpha)$ et D l'ensemble de ces classes. Nous allons munir D d'une structure de coalgèbre et montrer qu'elle est régulière et récursive.

Remarque 2.39 *Du fait de l'équivalence $\simeq_D$, on s'autorisera à noter $(\mathbb{T}, \alpha)$ pour toute coalgèbre finie $\mathbb{T}$. En effet, l'équivalence $\simeq_D$ permet de raisonner à renommage des variables près, ce qui permet de toujours se ramener au cas où le support de $\mathbb{T}$ est un sous-ensemble de V.*

Tout d'abord, pour toute coalgèbre finie $\mathbb{T} = (T, \tau)$, on définit la fonction $\Phi_{\mathbb{T}} : T \to D$ par :

$$\Phi_{\mathbb{T}}(\alpha) = [\mathbb{T}, \alpha]$$

Lemme 2.40 *Si $f : \mathbb{T} \to \mathbb{S}$ est un morphisme de coalgèbres finies, alors $\Phi_{\mathbb{S}} \circ f = \Phi_{\mathbb{T}}$.*

Preuve: On calcule : $\Phi_{\mathbb{S}}(f(\alpha)) = [\mathbb{S}, f(\alpha)] = [\mathbb{T}, \alpha] = \Phi_{\mathbb{T}}(\alpha)$. □

Cela permet de définir une fonction $\delta : D \to FD$ par :

$$\delta([\mathbb{T}, \alpha]) = F\Phi_{\mathbb{T}}(\tau(\alpha))$$

si $\mathbb{T} = (T, \tau)$. Cette définition est valide car si $f : \mathbb{T} \to \mathbb{S}$, alors, par le fait précédent : $F\Phi_{\mathbb{S}}(\sigma(f(\alpha))) = F\Phi_{\mathbb{S}}(Ff \circ \tau(\alpha)) = F(\Phi_{\mathbb{S}} \circ f)(\tau(\alpha)) = F\Phi_{\mathbb{T}}(\tau(\alpha))$. Soit $\mathbb{D} = (D, \delta)$ la coalgèbre ainsi définie. Nous allons prouver qu'elle est régulière et récursive.

Lemme 2.41 *La fonction $\Phi_{\mathbb{T}} : T \to D$ est un morphisme de coalgèbres $\mathbb{T} \to \mathbb{D}$.*

> *Preuve*: Conséquence immédiate de la définition de $\Phi_{\mathbb{T}}$ et de δ. □

Theorème 2.42 (Régularité) *La coalgèbre $\mathbb{D}$ est régulière.*

> *Preuve*: Soit $\mathbb{T} = (T, \tau)$ une coalgèbre finie quelconque. En utilisant les Lemmes 2.41 et 2.17, on obtient que l'ensemble $\Phi_{\mathbb{T}}(T) = \{[\mathbb{T}, \alpha] \mid \alpha \in T\}$ est une sous-coalgèbre de $\mathbb{D}$. Son caractère fini est évident. C'est donc une base, et tout élément de D est dans un tel ensemble. □

Le lemme suivant permet de relier deux visions des bases de $\mathbb{D}$. En effet, une telle base $\mathbb{B}$ peut être vue soit comme un ensemble fini d'éléments de D, soit comme une coalgèbre finie qui donne naissance à un morphisme $\Phi_{\mathbb{B}} : \mathbb{B} \to \mathbb{D}$.

Lemme 2.43 (Internalisation des bases) *Soit $\mathbb{B}$ une base de $\mathbb{D}$. Alors $\Phi_{\mathbb{B}}$ est l'inclusion canonique $\mathbb{B} \hookrightarrow \mathbb{D}$.*

> *Preuve*: Il s'agit de montrer que pour tout élément $[\mathbb{T}, \alpha]$ de $\mathbb{B}$, on a : $[\mathbb{B}, [\mathbb{T}, \alpha]] = [\mathbb{T}, \alpha]$. Pour cela, introduisons la base $\mathbb{S} = \Phi_{\mathbb{T}}(\mathbb{T}) \cup \mathbb{B}$ de $\mathbb{D}$. On peut voir $\Phi_{\mathbb{T}}$ comme un morphisme de coalgèbres $\mathbb{T} \to \mathbb{S}$ qui envoie α sur $[\mathbb{T}, \alpha]$ et cela donne l'égalité : $[\mathbb{T}, \alpha] = [\mathbb{S}, [\mathbb{T}, \alpha]]$. D'autre part, on peut considérer l'inclusion canonique $\mathbb{B} \to \mathbb{S}$, qui envoie $[\mathbb{T}, \alpha]$ (dans $\mathbb{B}$) sur lui-même (dans $\mathbb{S}$), ce qui donne : $[\mathbb{B}, [\mathbb{T}, \alpha]] = [\mathbb{S}, [\mathbb{T}, \alpha]]$. Par transitivité, on obtient l'égalité recherchée. □

Corollaire 2.44 *Soit $\mathbb{T}$ une coalgèbre finie. Alors le seul morphisme $\mathbb{T} \to \mathbb{D}$ est $\Phi_{\mathbb{T}}$.*

> *Preuve*: Soit $f : \mathbb{T} \to \mathbb{D}$ un tel morphisme. Soit $\mathbb{T}'$ l'image de $\mathbb{T}$ par f ; c'est encore une coalgèbre finie, donc une base de $\mathbb{D}$. On peut écrire $f = i_{\mathbb{T}',\mathbb{D}} \circ f'$ où f' est un morphisme $\mathbb{T} \to \mathbb{T}'$. Le Lemme 2.40 donne $\Phi_{\mathbb{T}} = \Phi_{\mathbb{T}'} \circ f'$ et le lemme précédent donne $\Phi_{\mathbb{T}'} = i_{\mathbb{T}',\mathbb{D}}$. On en déduit $f = \Phi_{\mathbb{T}}$. □

Le corollaire ci-dessous caractérise $\mathbb{D}$ par une propriété universelle dans la sous-algèbre pleine de $\mathbf{Alg}_F$ constituée des coalgèbres régulières.

Corollaire 2.45 (Finalité) *Soit $\mathbb{T}$ une coalgèbre régulière. Alors il existe un unique morphisme $\mathbb{T} \to \mathbb{D}$, que l'on note $\Phi_{\mathbb{T}}$.*

> *Preuve*: La restriction d'un tel morphisme à chaque base $\mathbb{B}$ de $\mathbb{T}$ doit coïncider avec l'unique morphisme $\mathbb{B} \to \mathbb{D}$. Cela donne l'unicité, ainsi que la manière de construire le morphisme $\mathbb{T} \to \mathbb{D}$. □

Corollaire 2.46 *L'unique morphisme $\mathbb{D} \to \mathbb{D}$ est l'identité.*

Corollaire 2.47 *Si $\mathbb{T}$ est une coalgèbre régulière, alors tout morphisme $\mathbb{D} \to \mathbb{T}$ est injectif.*

Preuve: Si f est un morphisme $\mathbb{D} \to \mathbb{T}$, alors $\Phi_{\mathbb{T}} \circ f$ est l'identité d'après le Corollaire précédent. □

Theorème 2.48 (Récursivité) *La coalgèbre $\mathbb{D}$ est récursive.*

Preuve: Soit $\mathbb{T} = (T, \tau)$ une extension finie de $\mathbb{D}$. D'après le Lemme 2.37, c'est une coalgèbre régulière, et on obtient avec le Corollaire 2.45 un morphisme $\Phi_{\mathbb{T}} : \mathbb{T} \to \mathbb{D}$. C'est une retraction d'après le Corollaire 2.46. □

Theorème 2.49 (Partage optimal) *L'unique congruence sur $\mathbb{D}$ est l'identité.*

Preuve: Soit $\equiv$ une congruence sur $\mathbb{D}$. Le Lemme 2.21 montre que le quotient $\mathbb{D}/\equiv$ est une coalgèbre régulière. Le Corollaire 2.47 indique alors que la projection $\pi : \mathbb{D} \to \mathbb{D}/\equiv$ est injective. La congruence est donc l'identité. □

Lemme 2.50 *Le morphisme $\delta : \mathbb{D} \to F\mathbb{D}$ est un isomorphisme.*

Preuve: Il suffit de vérifier que la fonction ensembliste est bijective. La surjectivité est conséquence du Lemme 2.30. D'après le Lemme 2.31, on sait que $F\mathbb{D}$ est régulière. Le Corollaire 2.47 montre alors que la fonction δ est injective. □

Techniques d'implémentation Des algorithmes efficaces pour implémenter le partage optimal ont été développés par Mauborgne [Mau99, Mau00] et Considine [Con00], dans le cas où le foncteur F associe à un ensemble X un ensemble de termes (libres) construits sur X. Ces techniques s'inspirent des méthodes de minimisation des automates déterministes finis [Wat94]. Elles ne s'adaptent pas directement, par exemple, au cas du foncteur $\mathcal{P}_f(\_)$, ou de foncteurs construits à partir de $\mathcal{P}_f(\_)$, c'est-à-dire lorsque l'on ajoute des axiomes tels que la commutativité et l'associativité sur les termes.

2.5.2 Partage minimal

Nous décrivons maintenant une autre construction d'une coalgèbre régulière et récursive qui ne s'appuie pas sur le partage maximal. En particulier, deux types définis par deux systèmes d'équations identiques à renommage des variables près ne seront pas égaux dans la coalgèbre. Au contraire, nous allons construire une coalgèbre récursive et régulière $\mathbb{T}_\infty$ avec partage minimal, c'est-à-dire telle que toute autre coalgèbre régulière et récursive s'obtienne comme un quotient de $\mathbb{T}_\infty$.

L'idée est de travailler avec des termes de la forme « α_1 where $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ » où les d_i sont des expressions construites sur les α_i et d'autres termes de la même forme.

Fixons un ensemble infini de variables V. On définit une suite d'ensembles $(T_n)_{n\in\mathbb{N}}$ par $T_0 = \emptyset$, $T_{n+1} = \{(X, \sigma, \alpha) \mid X \in \mathcal{P}_f(V), \sigma : X \to F(T_n + X), \alpha \in X\}$. On constate que cette suite est croissante (pour l'inclusion). On pose alors : $T_\infty = \bigcup T_n$. Pour tout couple (X, σ) avec $X \in \mathcal{P}_f(V), \sigma : X \to F(T_n + X)$,

on dispose de la fonction $\Phi_\sigma : T_\infty + X \to T_\infty$ définie comme l'identité sur T_∞ et par $\Phi_\sigma(\alpha) = (X, \sigma, \alpha) \in T_{n+1} \subseteq T_\infty$ pour $\alpha \in X$. On peut alors définir une fonction $\tau_\infty : T_\infty \to FT_\infty$ par $\tau_\infty(X, \sigma, \alpha) = F\Phi_\sigma(\sigma(\alpha))$. Soit $\mathbb{T}_\infty = (T_\infty, \tau_\infty)$ la coalgèbre ainsi définie.

Theorème 2.51 *La coalgèbre $\mathbb{T}_\infty$ est récursive.*

Preuve: Soit $\mathbb{S} = (S, \sigma)$ une extension finie de $\mathbb{T}_\infty$. On peut supposer que $S = T_\infty + X$ pour un certain ensemble $X \in \mathcal{P}_f(V)$. On note encore σ pour sa restriction à X. L'image de X par σ est contenue dans $F(T_\infty + X)$, qui, par continuité du foncteur F, est la réunion $\bigcup F(T_n + X)$. Or cet ensemble $\sigma(X)$ est fini, donc, pour n assez grand, on peut voir σ comme une fonction $X \to F(T_n + X)$. On considère alors la fonction $\Phi_\sigma : S \to T_\infty$ définie plus haut. Elle prolonge l'identité sur T_∞. Pour conclure la preuve, il suffit donc de montrer que c'est un morphisme de coalgèbres $\mathbb{S} \to \mathbb{T}_\infty$, c'est-à-dire que :

$$\tau_\infty \circ \Phi_\sigma = F\Phi_\sigma \circ \sigma$$

On vérifie cette égalité séparément sur T_∞ et sur X. Dans la mesure où Φ_σ se comporte comme l'identité sur T_∞, on en déduit que $F\Phi_\sigma$ se comporte comme l'identité sur FT_∞. De plus, σ coïncide avec τ_∞ sur T. Donc pour un élément $t \in T_\infty$, on a bien $\tau_\infty(\Phi_\sigma(t)) = \tau_\infty(t) = \sigma(t) = F\Phi_\sigma(\sigma(t))$.

Maintenant, soit $\alpha \in X$. On a : $\tau_\infty(\Phi_\sigma(\alpha)) = \tau_\infty(X, \sigma, \alpha)$ par définition de Φ_σ et $\tau_\infty(X, \sigma, \alpha) = F\Phi_\sigma(\sigma(\alpha))$ par définition de τ_∞. □

Theorème 2.52 *La coalgèbre T_∞ est régulière.*

Preuve: Tout d'abord, constatons que les T_n sont des sous-coalgèbres de T_∞. Cela découle du fait que si $\sigma : X \to F(T_n + X)$, alors l'image de tout $\alpha \in X$ par Φ_σ est dans T_{n+1} ; ainsi τ envoie T_{n+1} dans FT_{n+1}.

Comme $T_\infty = \bigcup T_n$, il suffit de montrer que toutes les sous-coalgèbres $\mathbb{T}_n$ sont régulières. On procède par récurrence. Le cas de base $n = 0$ est trivial. Supposons donc $\mathbb{T}_n$ régulière et montrons que $\mathbb{T}_{n+1}$ l'est aussi. Soit (X, σ, α) un élément de T_{n+1}. Montrons qu'il est régulier. On définit une coalgèbre $\mathbb{S} = (T_n + X, \tau + \sigma)$. C'est une extension finie de $\mathbb{T}_n$, c'est donc une algèbre régulière, d'après le Lemme 2.37. Comme dans la preuve du Théorème 2.51, on peut alors voir Φ_σ comme un morphisme de coalgèbres $\mathbb{S} \to \mathbb{T}_{n+1}$. D'après le Lemme 2.21, son image est régulière. Or $\Phi_\sigma(\alpha) = (X, \sigma, \alpha)$ et cet élément de T_{n+1} est donc régulier. □

Dans la mesure où $\mathbb{T}_\infty$ est régulière, on sait qu'il existe un morphisme $\mathbb{T}_\infty \to \mathbb{T}$ pour toute coalgèbre récursive $\mathbb{T}$ (Théorème 2.33). Nous allons montrer que si l'on suppose de plus $\mathbb{T}$ régulière et dénombrable, alors on peut construire un morphisme *surjectif* $\mathbb{T}_\infty \to \mathbb{T}$. Cela montre que toute coalgèbre qui vérifie ces conditions est isomorphe à un « quotient » de $\mathbb{T}_\infty$.

Theorème 2.53 *Soit $\mathbb{T} = (T, \tau)$ une coalgèbre récursive, régulière, et de support dénombrable. Alors il existe des morphismes surjectifs $\mathbb{T}_1 \to \mathbb{T}$ et $\mathbb{T}_\infty \to \mathbb{T}$.*

Preuve: Supposons donné, pour tout couple (X, σ) avec $X \in \mathcal{P}_f(V)$ et $\sigma : X \to FX$, un morphisme de coalgèbres $f_\sigma : (X, \sigma) \to \mathbb{T}$. On peut alors définir une fonction $f : T_1 \to T$ par $f(X, \sigma, \alpha) = f_\sigma(\alpha)$. On a alors $f_\sigma = f \circ \Phi_\sigma$. On a également $Ff_\sigma \circ \sigma = \tau \circ f_\sigma$, qui exprime le fait que f_σ est un morphisme. Cela permet d'établir que f est un morphisme de coalgèbres $\mathbb{T}_1 \to \mathbb{T}$; pour un triplet (X, σ, α), on calcule : $Ff \circ \tau_\infty(X, \sigma, \alpha) = Ff \circ F\Phi_\sigma \circ \sigma(\alpha) = F(f \circ \Phi_\sigma) \circ \sigma(\alpha) = Ff_\sigma \circ \sigma(\alpha) = \tau \circ f_\sigma(\alpha) = \tau \circ f(X, \sigma, \alpha)$

L'existence des morphismes f_σ pour tout (X, σ) est une conséquence de la récursivité de $\mathbb{T}$. On montre maintenant qu'un choix judicieux permet de rendre le morphisme $f : \mathbb{T}_1 \to \mathbb{T}$ surjectif.

Les bases de $\mathbb{T}$ sont en nombre dénombrable ; on les note $(B_n)_{n\in\mathbb{N}}$. Pour chacune de ces bases, on choisit un couple (X_n, σ_n) qui lui est isomorphe en tant que coalgèbre. On note $f_n : X_n \to B_n$ ce morphisme. Il est possible de choisir les couples (X_n, σ_n) pour qu'ils soient tous différents. Cela permet de choisir $f_{\sigma_n} = f_n$. Pour les autres f_σ, on fait un choix arbitraire.

Avec ces choix, le morphisme f est alors surjectif. En effet, pour tout entier n, il induit une bijection entre $\Phi_\sigma(X_n)$ et B_n. Le Corollaire 2.33 permet d'étendre f en un morphisme surjectif $\mathbb{T}_\infty \to \mathbb{T}$. □

Lemme 2.54 *Les deux coalgèbres $\mathbb{T}_\infty$ et $\mathbb{D}$ ne sont pas isomorphes (sauf si $FX = \emptyset$ pour tout X).*

Preuve: Si $FX = \emptyset$ pour tout X, il n'y a qu'une seule coalgèbre (de support vide). Supposons que ce n'est pas le cas. Par continuité du foncteur F, il existe un ensemble *fini* X tel que $FX \neq \emptyset$. À renommage près, on peut supposer que $X \in \mathcal{P}_f(V)$ et que $X \neq \emptyset$.

On sait qu'il existe un unique morphisme $f : \mathbb{T}_\infty \to \mathbb{D}$. Il s'agit de montrer que ce n'est pas un isomorphisme. Nous allons montrer qu'il n'est pas injectif. Considérons sa restriction à la sous-coalgèbre $\mathbb{T}_1$. C'est l'unique morphisme $f_1 : \mathbb{T}_1 \to \mathbb{D}$. Or l'on peut définir une fonction $T_1 \to D$ par $f_1(X, \sigma, \alpha) = [(X, \sigma), \alpha]$ et il est aisé de voir que c'est effectivement un morphisme de coalgèbre. C'est donc f_1. Il suffit alors de prouver que f_1 n'est pas injective. On dispose de $X \in \mathcal{P}_f(V)$ tel que $FX \neq \emptyset$. Il existe donc une fonction $\sigma : X \to FX$, et un élément $\alpha \in X$. On en déduit un triplet différent (X', σ', α'), en renommant par exemple les éléments de X (ou simplement en prenant un sur-ensemble strict de X), de sorte qu'il existe un morphisme de coalgèbres finies $(X, \sigma) \to (X', \sigma')$ qui envoie α sur α'. Les deux triplets (X, σ, α) et (X', σ', α') ont la même image par f_1, qui n'est donc pas injective. □

2.5.3 Partage modulo renommage et extension

On donne une troisième construction d'une coalgèbre régulière et récursive, qui n'est isomorphe ni à $\mathbb{T}_\infty$, ni à $\mathbb{D}$. Cela signifie qu'elle peut être obtenue à partir de $\mathbb{T}_\infty$ en quotientant par une congruence plus grosse que l'identité, mais qui n'est pas maximale non plus. Nous en donnons une construction directe.

L'idée est de considérer des systèmes $\{\alpha_1 = d_1; \ldots; \alpha_n = d_n\}$ (où les d_i sont construits sur les α_i) modulo renommage des variables et extension (ajout d'équations inutiles). Formellement, cela revient à considérer les couples $(\mathbb{T}, \alpha)$ où $\mathbb{T} = (T, \tau)$ est une coalgèbre finie dont le support est inclus dans V, et $\alpha \in T$. On considère l'équivalence induite par les relations :

$$(\mathbb{T}, \alpha) \simeq_{D'} (\mathbb{S}, f(\alpha))$$

pour tout morphisme *injectif* $f : \mathbb{T} \to \mathbb{S}$ (qui représente un renommage et un ajout d'équations). On note $< \mathbb{T}, \alpha >$ la classe d'équivalence du couple $(\mathbb{T}, \alpha)$ et D' l'ensemble de ces classes. On procède alors comme pour le cas du partage optimal. Pour toute coalgèbre finie $\mathbb{T} = (T, \tau)$, on définit la fonction $\Phi'_{\mathbb{T}} : T \to D'$ par :

$$\Phi'_{\mathbb{T}}(\alpha) =< \mathbb{T}, \alpha >$$

Lemme 2.55 *Si $f : \mathbb{T} \to \mathbb{S}$ est un morphisme injectif de coalgèbres finies, alors $\Phi'_{\mathbb{S}} \circ f = \Phi'_{\mathbb{T}}$.*

On peut alors définir la fonction $\delta' : D' \to FD'$ par :

$$\delta'(< \mathbb{T}, \alpha >) = F\Phi'_{\mathbb{T}}(\tau(\alpha))$$

si $\mathbb{T} = (T, \tau)$. Soit $\mathbb{D}' = (D', \delta')$ la coalgèbre ainsi définie.

Lemme 2.56 *La fonction $\Phi'_{\mathbb{T}} : T \to D'$ est un morphisme de coalgèbres $\mathbb{T} \to \mathbb{D}'$.*

Theorème 2.57 *La coalgèbre $\mathbb{D}'$ est régulière.*

Preuve: La preuve est identique à celle du Théorème 2.42. □

Theorème 2.58 *La coalgèbre $\mathbb{D}'$ est récursive.*

Preuve: Soit $\mathbb{S} = (S, \sigma)$ une extension finie de $\mathbb{D}'$. On peut supposer $S = D' + X$ avec $X = \{\alpha_1, \ldots, \alpha_n\} \in \mathcal{P}_f(V)$. D'après le Lemme 2.37, on en déduit l'existence d'une base B de $\mathbb{D}'$ qui contient X. On écrit $B = \{< \mathbb{T}_1, \beta_1 >, \ldots, < \mathbb{T}_n, \beta_m >\} + X$. En fait, compte-tenu de l'équivalence modulo renommage et extension et quitte à renommer les β_i, on peut même supposer que tous les $\mathbb{T}_i$ sont égaux : $B = \{< \mathbb{T}, \beta_1 >, \ldots, < \mathbb{T}, \beta_m >\} + X$ avec $\mathbb{T} = (T, \tau)$. Posons alors $T' = T + X$ et définissons la fonction $f : B \to T'$ par $f(< \mathbb{T}, \beta_i >) = \beta_i$ et $f(\alpha_i) = \alpha_i$. On peut alors définir $\tau' : T' \to FT'$, prolongement de $\tau : T \to FT$, par : $\tau'(\alpha_i) = Ff(\sigma(\alpha_i))$ pour $\alpha_i \in X$. En effet, on sait que $\mathbb{B}$ est une base, et donc $\sigma(X) \subseteq B$. On dispose ainsi d'une coalgèbre finie $\mathbb{T}' = (T', \tau')$.

On peut maintenant définir une retraction $g : \mathbb{S} \to \mathbb{D}'$ comme l'identité sur D' et par $g(\alpha_i) = \Phi'_{\mathbb{T}'}(\alpha_i) =< \mathbb{T}', \alpha_i >$. Il faut vérifier que c'est un morphisme de coalgèbres. Cela découle d'un simple calcul, en remarquant que la restriction de g à B est la composée $\Phi'_{\mathbb{T}'} \circ f : \delta' \circ g(\alpha_i) = \delta'(< \mathbb{T}', \alpha_i >) = F\Phi'_{\mathbb{T}'}(\tau'(\alpha_i)) = F(\Phi'_{\mathbb{T}'} \circ f)(\sigma(\alpha_i)) = Fg \circ \sigma(\alpha_i)$. □

Nous allons maintenant montrer que cette construction donne en général une coalgèbre $\mathbb{D}'$ qui n'est isomorphe ni à $\mathbb{D}$ ni à $\mathbb{T}_\infty$. Nous allons prendre comme foncteur F l'identité. Les coalgèbres sont alors les graphes (finis ou infinis) dont le degré sortant de chaque nœud est exactement 1.

On voit facilement que $\mathbb{D}$ est un singleton. On peut aussi décrire très précisément $\mathbb{D}'$. Pour toute coalgèbre $\mathbb{T} = (T, \tau)$, et tout élément $\alpha \in \tau$, on définit $p_\mathbb{T}(\alpha)$ comme le plus petit entier n strictement positif tel quel $\tau^n(\alpha) = \alpha$, ou $+\infty$ si un tel entier n'existe pas (ce qui n'est possible que lorsque $\mathbb{T}$ est infinie). On constate que si $f : \mathbb{T} \to \mathbb{S}$ est un morphisme injectif de coalgèbres, alors $p_\mathbb{T}(\alpha) = p_\mathbb{S}(f(\alpha))$ pour tout α. On peut donc définir une fonction $p : D' \to \mathbb{N}^+$ par $p(< \mathbb{T}, \alpha >) = p_\mathbb{T}(\alpha)$. Nous allons montrer que c'est une bijection. Pour un entier $n > 0$ fixé, on peut considérer la coalgèbre finie $\mathbb{Z}_n = (\mathbb{Z}/n\mathbb{Z}, (x \mapsto x+1))$. Pour tout $x \in \mathbb{Z}$, on a : $p_{\mathbb{Z}_n}(x) = n$. On constate alors que pour toute coalgèbre $\mathbb{T}$ et $\alpha \in T$, si $p_\mathbb{T}(\alpha) = n$, alors il existe un morphisme injectif $\mathbb{Z}_n \to \mathbb{T}$ qui envoie un élément arbitraire de $\mathbb{Z}_n$ sur α. On en déduit le caractère bijectif de la fonction p, ce qui donne un cardinal dénombrable pour $\mathbb{D}'$ et l'on voit immédiatement que $\mathbb{D}$ et $\mathbb{D}'$ ne sont pas isomorphes. Mais on observe aussi que pour tout élément x de $\mathbb{D}'$, on a $p_{\mathbb{D}'}(x) = 1$ (autrement dit : δ' est l'identité). En effet, un tel élément x peut toujours s'écrire $< \mathbb{Z}_n, 0 >$ pour un certain n, et $\delta'(< \mathbb{Z}_n, 0 >) =< \mathbb{Z}_n, 1 >=< \mathbb{Z}_n, 0 >$. Clairement, la coalgèbre $\mathbb{T}_\infty$ n'a pas la même structure. Par exemple, si l'on prend $X = \{\alpha_1, \alpha_2\}$, et $\sigma(\alpha_1) = \alpha_2$, $\sigma(\alpha_2) = \alpha_1$, alors pour l'élément $x = (X, \sigma, \alpha_1)$ de $\mathbb{T}_\infty$, on a : $p_{\mathbb{T}_\infty}(x) = 2$.

Chapitre 3

Algèbre de types

Dans le chapitre précédent, nous avons introduit le formalisme pour construire des (co)algèbres de termes récursifs sur une signature arbitraire, donnée par un endofoncteur ensembliste F. Dans ce chapitre, nous instancions ce cadre pour définir l'algèbre de types d'un calcul, qui servira de cœur sur lequel nous construirons le langage $\mathbb{C}$Duce.

3.1 Combinaisons booléennes

3.1.1 Motivation

Nous voulons introduire des combinaisons booléennes (réunion, intersection, complémentaire) dans l'algèbre de types. Une solution immédiate serait de considérer ces opérations booléennes comme des constructeurs de types, au même titre, disons, que les types flèche ou produit. Dans une telle approche, on définirait le foncteur F par :

$$t \in FX ::= x \rightarrow x \mid x \times x \mid x \vee x \mid x \wedge x \mid \neg x \mid \ldots$$

où la métavariable x représente des éléments quelconques de X.

Cette approche se prête bien à la construction d'une algèbre inductive de types. Mais dans notre formalisme pour les coalgèbres récursives, cela permettrait de définir un type t par une équation comme :

$$t = \neg t$$

Évidemment, il n'est pas possible de donner une sémantique ensembliste à un tel type. De même, une équation comme :

$$t = t \vee t$$

ne donne aucune information sur le contenu du type t. De manière générale, on veut éviter les récursions mal-formées, qui « ne traversent pas un vrai constructeur ». Une solution est de stratifier la construction, en utilisant une construction inductive par les connecteurs booléens. Par exemple, on peut définir le foncteur F par :

$$t \in FX ::= x \rightarrow x \mid x \times x \mid t \vee t \mid t \wedge t \mid \neg t \mid \ldots$$

Pour un ensemble X fixé, l'ensemble FX représente des termes finis construits avec les connecteurs booléens, et dont les atomes sont les « vrais » constructeurs de l'algèbre de types. On évite ainsi de pouvoir définir des types mal-formés.

Cette construction présente cependant l'inconvénient d'interdire d'identifier dans l'implémentation des types équivalents modulo des tautologies booléennes (par exemple t et $t \vee t$). En particulier, même si l'ensemble X est fini, le foncteur donne un ensemble FX infini. Cela peut poser un problème avec des algorithmes (par exemple, un algorithme de sous-typage) qui procèdent en saturant un ensemble de types, et en introduisant dans leurs calculs internes des combinaisons booléennes de types. Pour en assurer la terminaison, il faut trouver une représentation qui autorise à partager des combinaisons booléennes tautologiquement équivalentes. Il n'est pas nécessaire de détecter toutes ces équivalences, car cela peut avoir un coût important, mais identifier plus de combinaisons équivalente permet de réduire le nombre d'itérations nécessaire dans les algorithmes qui procèdent par saturation.

Nous allons étudier un exemple de représentation qui convient. Dans la section 11.3, nous indiquons des variantes possibles.

3.1.2 Combinaisons booléennes finies en forme normale disjonctive

Une combinaisons booléenne finie en forme normale disjonctive sur un ensemble X d'atomes est une réunion finie d'intersections finies de littéraux, un littéral étant soit un atome, soit le complémentaire d'un atome. On peut définir l'ensemble de ces objets par la syntaxe stratifiée :

$$\begin{array}{lcl} C & ::= & \mathbb{0} \mid C \vee C \mid L \\ L & ::= & \mathbb{1} \mid L \wedge L \mid x \mid \neg x \end{array}$$

La « production » L représente les « lignes » d'une forme normale disjonctive. En prenant en compte l'associativité et la commutativité de l'intersection, on peut la définir de manière moins syntaxique en réunissant d'une part l'ensemble des littéraux positifs x et d'autre part l'ensemble des littéraux négatifs $\neg x$. On considère donc l'ensemble $\mathcal{P}_f(X) \times \mathcal{P}_f(X)$. Par exemple, la ligne $x_1 \wedge \neg x_2 \wedge x_3$ peut se représenter par le couple $(\{x_1, x_3\}, \{x_2\})$. L'élément neutre pour l'intersection $\mathbb{1}$ correspond au couple $(\emptyset, \emptyset)$.

On peut faire de même pour la production C, en voyant une combinaison comme un ensemble fini de lignes. Globalement, une combinaison booléenne finie en forme normale disjonctive peut ainsi se représenter par un élément de l'ensemble $\mathcal{P}_f(\mathcal{P}_f(X) \times \mathcal{P}_f(X))$ que l'on note $\mathcal{B}X$ ($\mathcal{B}$ pour combinaison booléennes). L'élément neutre pour la réunion $\mathbb{0}$ correspond à l'ensemble vide dans cette représentation.

Nous pouvons maintenant définir les connecteurs booléens comme des opérateurs sur cet ensemble $\mathcal{B}X$. On commence par noter $@x$ l'élément $\{(\{x\}, \emptyset)\}$ de $\mathcal{B}X$ qui correspond à l'atome x, et de même $@\neg x$ pour le littéral négatif $\{(\emptyset, \{x\})\}$. On définit la réunion et l'intersection par les formules :

$$t_1 \vee t_2 := t_1 \cup t_2$$

$$t_1 \wedge t_2 := \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_1, N_1) \in t_1, (P_2, N_2) \in t_2\}$$

Il est clair que les opérateurs $\vee$ et $\wedge$ sont commutatifs et associatifs, et qu'ils admettent respectivement $\mathbb{0} := \emptyset$ et $\mathbb{1} := \{(\emptyset, \emptyset)\}$ comme éléments neutres. Pour toute famille *finie* $(t_i)_{i\in I}$ d'éléments de $\mathcal{B}X$, on peut donc définir naturellement $\bigvee_{i\in I} t_i$ et $\bigwedge_{i\in I} t_i$ par récurrence sur le cardinal de I. De manière équivalente, on peut les définir directement par les formules :

$$\bigvee_{i\in I} t_i := \bigcup_{i\in I} t_i$$

$$\bigwedge_{i\in I} t_i := \{(\bigcup_{i\in I} P_i, \bigcup_{i\in I} N_i \mid \forall i \in I.\ (P_i, N_i) \in t_i\}$$

qui montrent le caractère associatif des opérateurs $\vee$ et $\wedge$.

Lemme 3.1 *Pour tout élément $t \in \mathcal{B}X$, nous avons :*

$$t = \bigvee_{(P,N)\in t} \left(\bigwedge_{x\in P} @x \wedge \bigwedge_{x\in N} @\neg x \right)$$

Ce fait suggère la définition suivante pour le complémentaire et la différence :

$$\neg t := \bigwedge_{(P,N)\in t} \left(\bigvee_{x\in P} @\neg x \vee \bigvee_{x\in N} @x \right)$$

$$t_1 \setminus t_2 := t_1 \wedge \neg t_2$$

Lemme 3.2 *Pour tout atome $x \in X$, on a $\neg @x = @\neg x$.*

Dans l'optique de définir une interprétation ensembliste des types, il est naturel d'introduire la définition suivante.

Definition 3.3 *Une* **interprétation ensembliste** *de $\mathcal{B}X$ dans un ensemble D est une fonction $[\![\_]\!] : \mathcal{B}X \to \mathcal{P}(D)$ telle que, pour tout $t, t_1, t_2 \in \mathcal{B}X$:*

- $[\![\mathbb{0}]\!] = \emptyset$
- $[\![\mathbb{1}]\!] = D$
- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
- $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$
- $[\![\neg t]\!] = D \setminus [\![t]\!]$

Lemme 3.4 *Une fonction $[\![\_]\!] : \mathcal{B}X \to \mathcal{P}(D)$ est une interprétation ensembliste si et seulement si pour tout $t \in \mathcal{B}X$:*

$$[\![t]\!] = \bigcup_{(P,N)\in t} \left(\bigcap_{x\in P} [\![@x]\!] \setminus \bigcup_{x\in N} [\![@x]\!] \right)$$

(avec la convention qu'une intersection indexée sur l'ensemble vide vaut D)

> *Preuve*: Supposons que $[\![\_]\!]$ est une interprétation ensembliste et considérons un élément $t \in \mathcal{B}X$. En utilisant le Lemme 3.1, on obtient :
>
> $$[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{x\in P} [\![@x]\!] \cap \bigcap_{x\in N} [\![@\neg x]\!]$$

Le Lemme 3.2 donne $[\![@\neg x]\!] = [\![\neg @x]\!] = D \backslash [\![@x]\!]$, ce qui permet de conclure :

$$[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{x\in P} [\![@x]\!] \backslash \bigcup_{x\in N} [\![@x]\!]$$

Réciproquement, si l'on suppose cette formule valide pour tout $t \in \mathcal{B}X$, on vérifie facilement que $[\![\_]\!]$ est une interprétation ensembliste. Étudions par exemple le cas $t_1 \wedge t_2$. On a :

$$\begin{aligned}
[\![t_1]\!] \cap [\![t_2]\!] &= \left(\bigcup_{(P_1,N_1)\in t_1} \bigcap_{x\in P_1} [\![@x]\!] \backslash \bigcup_{x\in N_1} [\![@x]\!] \right) \\
&\quad \cap \left(\bigcup_{(P_2,N_2)\in t_2} \bigcap_{x\in P_2} [\![@x]\!] \backslash \bigcup_{x\in N_2} [\![@x]\!] \right) \\
&= \bigcup_{\substack{(P_1,N_1)\in t_1 \\ (P_2,N_2)\in t_2}} \bigcap_{x\in P_1\cup P_2} [\![@x]\!] \backslash \bigcup_{x\in N_1\cup N_2} [\![@x]\!] \\
&= [\![t_1 \wedge t_2]\!]
\end{aligned}$$

□

Corollaire 3.5 *Pour tout fonction $[\![\_]\!] : X \to \mathcal{P}(D)$, il existe une unique interprétation ensembliste de $\mathcal{B}X$ dans D qui envoie $@x$ sur $[\![x]\!]$ pour tout atome $x \in X$. On la note encore $[\![\_]\!]$.*

Une propriété importante est que l'on peut voir $\mathcal{B} = \mathcal{P}_f(\mathcal{P}_f(\_) \times \mathcal{P}_f(\_))$ comme un foncteur qui préserve les inclusions ensemblistes. Il est légitime de ne pas préciser l'ensemble X lorsqu'on manipule les opérateurs $\vee, \wedge, \neg, \setminus$ et les éléments $\mathbb{0}$ et $\mathbb{1}$. Par exemple, si t_1, t_2 sont simultanéments dans $\mathcal{B}X$ et dans $\mathcal{B}Y$, alors on peut calculer $t_1 \vee t_2$ indifféremment dans $\mathcal{B}X$ et dans $\mathcal{B}Y$, et obtenir le même résultat. On peut interpréter cela de manière catégorique en disant que les connecteurs sont « naturels » par rapport au foncteur $\mathcal{B}$ (par exemple, l'opérateur $\vee$ peut être vu comme une transformation naturelle $\mathcal{B}\_ \times \mathcal{B}\_ \to \mathcal{B}\_$).

Pour tout élément $t \in \mathcal{B}X$, on peut définir l'« ensemble des atomes de t » comme le plus petit ensemble $Y \subseteq X$ tel que $t \in \mathcal{B}Y$, à savoir $\bigcup_{(P,N)\in t} P \cup N$.

3.2 Algèbres avec types récursifs et combinaisons booléennes

Nous avons maintenant tous les éléments en main pour définir une algèbre de types avec les caractéristiques suivantes :

- types récursifs ;
- constructeurs arbitraires ;
- combinaisons booléennes bien formées.

Si l'on note F la signature des constructeurs (c'est-à-dire un foncteur **Set** $\to$ **Set** qui préserve les inclusions ensemblistes), on peut définir une algèbre de types comme étant une $(\mathcal{B} \circ F)$-coalgèbre récursive et régulière $\mathbb{T} = (T, \tau)$. Nous notons $\widehat{T} = \mathcal{B}FT$ et $T_A = FT$.

Definition 3.6 *Les éléments de T (resp. $\widehat{T}$) (resp. T_A) sont appelés* **nœuds de types** *(resp.* **expression de types***, ou simplement* **types***) (resp.* **atomes***).*

La fonction $\tau : T \to \widehat{T}$ associe à un nœud une expression, et une expression est une combinaison booléenne (en forme normale disjonctive) d'atomes. Les atomes sont des objets construits sur des nœuds de types, via la signature F. On notera $\theta, \theta_1, \ldots$ les éléments de T, et $t, t_1, \ldots$ les éléments de $\widehat{T}$.

Pour assurer l'existence d'une algèbre de type, il suffit de supposer que le foncteur F est continu. Si c'est le cas, le foncteur $\mathcal{B} \circ F$ l'est également, et on peut appliquer les construction du chapitre précédent.

Il s'agit maintenant de définir la signature F, c'est-à-dire la manière de construire des atomes à partir de nœuds de types.

3.3 L'algèbre minimale

Nous allons commencer à travailler avec une algèbre minimale qui ne contient que trois sortes de constructeurs : types produit, types flèche, et types de base. Cela sera suffisant pour présenter notre approche. Nous pourrons ensuite étendre l'algèbre avec d'autres constructeurs (enregistrements, éléments XML, ...) en expliquant les modifications à apporter au système minimal.

Nous fixons un ensemble $\mathbb{B}$ de types de base (par exemple `Int`, `Char`) et nous définissons la signature F de la manière suivante :

$$a \in FX ::= x \rightarrow x \;\mid\; x \times x \;\mid\; b$$

où b désigne un élément générique de $\mathbb{B}$ et x un élément générique de X.

Résumons les propriétés de l'algèbre de types ainsi construite. On dispose des fonctions suivantes :

- Les deux constructeurs binaires $\rightarrow$ et $\times : T \times T \to T_A$ et les constructeurs 0-aires $b \in T_A$.
- L'injection des atomes dans les expressions de types $@ : T_A \to \widehat{T}$. Nous allons la rendre implicite, en omettant $@$, ce qui revient à faire comme si $T_A \subseteq \widehat{T}$.
- Les connecteurs booléens sur les expressions types $\vee, \wedge, \setminus : \widehat{T} \times \widehat{T} \to \widehat{T}$ et $\neg : \widehat{T} \to \widehat{T}$, ainsi que les deux expressions de type $\mathbb{0}, \mathbb{1} \in \widehat{T}$.
- La fonction surjective $\tau : T \to \widehat{T}$ qui associe à un nœud de type sa description (une expression).

Il y a trois « univers » d'atomes : les types flèche, les types produit, les types de base. Notons T_u pour les atomes dans l'univers u, où $u \in \{\mathbf{fun}, \mathbf{prod}, \mathbf{basic}\}$:

$$\begin{array}{lcl} T_{\mathbf{basic}} & = & \mathbb{B} \\ T_{\mathbf{fun}} & = & \{\theta_1 \rightarrow \theta_2 \mid (\theta_1, \theta_2) \in T\} \\ T_{\mathbf{prod}} & = & \{\theta_1 \times \theta_2 \mid (\theta_1, \theta_2) \in T\} \end{array}$$

On a :

$$T_A = \bigcup_{u \in \{\mathbf{fun}, \mathbf{prod}, \mathbf{basic}\}} T_u$$

De plus, la $(\mathcal{B} \circ F)$-coalgèbre est régulière et récursive. La recursivité implique que la fonction τ est surjective : pour toute expression de type t, il est possible de trouver (en pratique, de créer) un nœud dont la description est t. Cela permet

de définir des types inductivement. Il existe ainsi des nœuds $\theta_{\mathbb{0}}$ et $\theta_{\mathbb{1}}$ tels que : $\tau(\theta_{\mathbb{0}}) = \mathbb{0}$ et $\tau(\theta_{\mathbb{1}}) = \mathbb{1}$. Pour simplifier les notations, nous les noterons encore $\mathbb{0}$ et $\mathbb{1}$.

On peut aussi former des types récursifs en considérant des systèmes finis :

$$\left\{ \begin{array}{lcl} \alpha_1 & = & d_1 \\ \dots & & \\ \alpha_n & = & d_n \end{array} \right.$$

où les d_i sont des éléments de $\mathcal{B}F(T+\{\alpha_1,\dots,\alpha_n\})$, c'est-à-dire des expressions de types, mais où l'on autorise des variables en plus des nœuds comme éléments constitutifs des atomes. Voici un exemple :

$$\left\{ \begin{array}{lcl} \alpha_1 & = & \alpha_1 \times \alpha_2 \\ \alpha_2 & = & (\alpha_1 \to \alpha_2) \vee (\alpha_2 \to \alpha_1) \end{array} \right.$$

La récursivité assure l'existence de deux nœuds θ_1 et θ_2 tels que :

$$\left\{ \begin{array}{lcl} \tau(\theta_1) & = & \theta_1 \times \theta_2 \\ \tau(\theta_2) & = & (\theta_1 \to \theta_2) \vee (\theta_2 \to \theta_1) \end{array} \right.$$

On peut alors réutiliser ces nœuds dans un nouveau système :

$$\left\{ \begin{array}{lcl} \alpha_3 & = & (\theta_1 \times \alpha_3) \vee (\theta_2 \to \theta_2) \end{array} \right.$$

qui permet de construire un nœud θ_3 tel que :

$$\tau(\theta_3) = (\theta_1 \times \theta_3) \vee (\theta_2 \to \theta_2)$$

Voilà la manière de *construire* des types récursifs. Nous allons maintenant voir comment les déconstruire.

Definition 3.7 *Un* **socle** *est ensemble de types $\beth \subseteq \widehat{T}$ qui vérifie les conditions :*

- *$\beth$ est fini ;*
- *$\beth$ contient $\mathbb{0}$, $\mathbb{1}$ et il est stable par les connecteurs booléens ($\vee$,$\wedge$,$\neg$) ;*
- *pour tout type $t \in \beth$ et tout $(P,N) \in t$:*

$$(\theta_1 \to \theta_2 \in P \cup N) \vee (\theta_1 \times \theta_2 \in P \cup N) \Rightarrow \tau(\theta_1) \in \beth \wedge \tau(\theta_2) \in \beth$$

Theorème 3.8 *Tout ensemble fini de types est inclus dans un socle.*

Preuve: Il suffit de le montrer pour un ensemble de la forme $\tau(X)$ où X est une base. En effet, tout ensemble fini de types est inclus dans un tel $\tau(X)$.

Soit A l'ensemble des atomes qui apparaissent dans un élément de $\tau(X)$:

$$A = \bigcup_{t \in \tau(X)} \bigcup_{(P,N) \in t} P \cup N$$

Il est clair que c'est le plus petit ensemble tel que $\tau(X) \subseteq \mathcal{B}A$. Comme X est une base, on a $\tau(X) \subseteq \mathcal{B}FX$, et donc $A \subseteq FX$.

Nous allons montrer que l'ensemble $\beth = \mathcal{B}A$ est un socle. Comme l'ensemble X est fini, il en est de même de $\tau(X)$, donc

de A, et finalement de $\beth$. La stabilité de $\beth$ par les connecteurs booléens est triviale (comme pour tout ensemble de la forme $\mathcal{B}\_$).

Prenons un atome $\theta_1 \to \theta_2$ qui apparaît dans un des types de $\beth$. C'est un élément de A, et donc de FX. On en déduit que θ_1 et θ_2 sont dans X, et donc $\tau(\theta_i) \in \tau(X) \subseteq \mathcal{B}A = \beth$.

Le cas $\theta_1 \times \theta_2$ est similaire. □

Ce théorème assure que si l'on déroule un type t en considérant la description des nœuds de types qui apparaissent dans ses atomes, et cela inductivement, on n'obtient qu'un nombre fini de types différents. Mieux, on peut saturer l'ensemble obtenu en appliquant les connecteurs booléens, et l'on obtient encore un ensemble fini. Cette propriété de finitude sera utilisée pour assurer la terminaison des algorithmes.

Chapitre 4

Sous-typage sémantique

Dans ce chapitre, nous montrons comment définir une relation de sous-typage sur l'algèbre de types minimale $\mathbb{T}$ introduite dans le chapitre précédent. Nous adoptons une approche originale qui repose sur une interprétation ensembliste des types.

Approches ensemblistes naïves Nous voulons *définir* la relation de sous-typage $\leq$ en utilisant une interprétation ensembliste (au sens de la Définition 3.3), c'est-à-dire une fonction $\widehat{T} \to \mathcal{P}(D)$, pour un certain ensemble D, qui interprète de manière ensembliste les connecteurs booléens.

Definition 4.1 (Sous-typage) *Soit $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ une interprétation ensembliste. Elle définit une relation binaire $\leq_{[\![\_]\!]}$ sur $\widehat{T}$ par :*

$$t_1 \leq_{[\![\_]\!]} t_2 \iff [\![t_1]\!] \subseteq [\![t_2]\!]$$

Cette relation est **la relation de sous-typage induite par $[\![\_]\!]$**.

On a clairement l'équivalence : $t_1 \leq_{[\![\_]\!]} t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. Autrement dit, pour étudier cette relation de sous-typage, il suffit de se concentrer sur le prédicat unaire « $[\![\_]\!] = \emptyset$ ».

Pour définir ainsi une relation de sous-typage, il faut choisir d'une part cet ensemble D, et d'autre part la manière d'interpréter les atomes T_A.

Un choix naturel serait de prendre pour D l'ensemble des valeurs du calcul que nous voulons définir (voir Chapitre 5), d'interpréter un type t comme l'ensemble des valeurs de ce type, et de vérifier que l'on a bien défini une interprétation ensembliste. On poserait : $[\![t]\!] = \{v \mid \vdash v : t\}$. Cela nécessite de disposer du système de types pour le langage, or c'est précisément pour pouvoir le définir que nous avons besoin d'introduire la relation de sous-typage. En fait, nous avons seulement besoin de connaître le type pour les valeurs, et non pour des expressions arbitraires. Il s'agit simplement de savoir, étant donné un atome $a \in T_A$ et une valeur v, si v est de type a ou non. Le problème provient des fonctions. Si v est une λ-abstraction $\lambda x.e$ et a un type flèche $\theta_1 \to \theta_2$, on voudrait dire que v a le type a si et seulement si $x : \tau(\theta_1) \vdash e : \tau(\theta_2)$, mais alors on aurait besoin du système de type pour tout le langage. De plus, une telle définition pose un problème de sûreté par rapport au typage. Dans un calcul avec sous-typage, le type d'une expression devient de plus en plus précis au cours de son

évaluation. Si l'abstraction ci-dessus n'a pas le type a, cela signifie simplement que le système de type n'est pas assez puissant pour prouver que e a le type $\tau(\theta_2)$ sous l'hypothèse $x : \tau(\theta_1)$; dans ce cas, la valeur v doit avoir le type $\neg a$ (pour avoir une interprétation ensembliste des types). Supposons que l'on puisse réduire le corps de v [1], de sorte à obtenir une nouvelle valeur $v' = \lambda x.e'$. Il se peut que le système arrive alors à prouver que e' a bien le type $\tau(\theta_2)$, et donc v' le type a, mais elle doit bien avoir le type $\neg a$, tout comme l'abstraction dont elle est dérivée. Nous avons donc une valeur de type a et de type $\neg a$, ce qui est impossible avec une interprétation ensembliste.

Une autre possibilité serait de dire que l'abstraction $\lambda x.e$ a le type $\theta_1 \rightarrow \theta_2$ si et seulement si toutes les valeurs que l'on peut obtenir en réduisant une expression $e[x \mapsto v]$ avec v de type θ_1 sont de type θ_2, et qu'aucune erreur de type ne peut apparaître au cours de ces réductions. Cela pose également des problèmes. D'un part, on s'appuie encore sur le système de types du calcul (qui n'est pas encore défini, et dont la définition va dépendre de la relation de sous-typage). D'autre part, la sémantique de calcul elle-même va dépendre de la relation de sous-typage (via l'opération de filtrage, ou simplement de test de type dynamique), et donc cette définition est mal fondée. Cela ne fait donc qu'introduire une boucle de circularité supplémentaire (entre typage, sous-typage et sémantique dynamique).

Dans le calcul que nous allons introduire, les abstractions sont explicitement annotées par leur type. Comme nous voulons disposer des fonctions surchargées, une abstraction est annotée par un ensemble fini de types flèche $\theta_1 \rightarrow \theta'_1; \ldots; \theta_n \rightarrow \theta'_n$. Cette abstraction a le type atome $a = \theta \rightarrow \theta'$ si et seulement si a est un super-type de l'intersection $\bigwedge_{i=1..n} \theta_i \rightarrow \theta'_i$ (et donc elle a le type $\neg a$ si ce n'est pas le cas). Ce typage ne dépend donc pas vraiment du corps de l'abstraction (étant entendu par ailleurs que celui-ci est bien typé). Mais nous n'avons fait que déplacer le problème, car l'on définit le prédicat $\vdash v : a$ (lorsque v est une abstraction et a un type flèche) en utilisant la relation de sous-typage que l'on veut définir.

Idée de la solution proposée Les approches naïves présentées plus haut échouent. Nous proposons une solution qui consiste à assouplir le lien qui existe entre l'ensemble D, utilisé pour interpréter les types, et l'ensemble des valeurs du calcul que nous voulons typer. Nous voulons *in fine* pouvoir interpréter les types comme ensembles de valeurs (et le sous-typage comme inclusion des ensembles dénotés). Mais nous avons vu qu'il est problématique de vouloir partir de cette propriété pour définir le sous-typage. En fait, la nature précise des éléments de D n'a pas beaucoup d'importance : cet ensemble n'est utilisé que pour définir la relation $\leq$. Nous sommes donc assez libres pour choisir D, la contrainte étant d'interpréter les atomes de types de sorte à produire une relation de sous-typage qui a du sens pour les valeurs.

Nous allons définir une notion de *modèle*, qui capture cette intuition. Un modèle est une manière d'interpréter de manière ensembliste les types de sorte à ce que la relation de sous-typage induite corresponde à notre intuition des types et du calcul.

Partant d'un modèle, nous pouvons définir une relation de sous-typage, et

[1] Dans le calcul que nous allons introduire, il n'est pas possible de réduire à l'intérieur des abstractions, mais on pourrait le faire sans mettre en cause la sûreté du typage.

l'utiliser pour introduire le système de types du calcul. Il est alors possible de définir *a posteriori* une nouvelle interprétation des types comme ensemble de valeurs, en utilisant ce système de types (on prend $[\![t]\!] = \{v \mid \vdash v : t\}$). Le critère pour valider notre approche consiste alors à vérifier que cette interprétation induit la *même* relation de sous-typage que le modèle dont nous sommes partis. Cela montre que la boucle est bouclée, et que même si nous avons *a priori* brisé le lien entre sous-typage et valeurs (en définissant le sous-typage via un modèle), ce lien est restauré *in fine*.

4.1 Types de base

Pour chaque type de base $b \in \mathbb{B}$, nous supposons donné un ensemble $\mathbb{B}[\![b]\!]$, dont les éléments sont appelés constantes de type b. Ces constantes sont celles du calcul que nous voulons étudier.

Les ensembles $\mathbb{B}[\![b]\!]$ ne sont pas forcément deux-à-deux disjoints : une même constante peut avoir plusieurs types de base différents. Par exemple, dans $\mathbb{C}$Duce, la constante 0 a le type singleton 0, le type `Int`, tout type intervalle qui contient 0, ... Cependant, nous supposons que pour chaque constante c, il existe un type de base singleton $b_c \in \mathbb{B}$, qui vérifie :

$$[\![b_c]\!] = \{c\}$$

Cela implique :

$$\forall b \in \mathbb{B}.\ c \in \mathbb{B}[\![b]\!] \iff \mathbb{B}[\![b_c]\!] \subseteq \mathbb{B}[\![b]\!] \iff \mathbb{B}[\![b_c]\!] \cap \mathbb{B}[\![b]\!] \neq \emptyset$$

On note $\mathcal{C} = \bigcup_{b \in \mathbb{B}} \mathbb{B}[\![b]\!]$ l'ensemble des constantes.

4.2 Modèles

Nous voulons définir une notion de modèle ensembliste de l'algèbre de types. Un modèle doit être une interprétation ensembliste (Définition 3.3), ce qui garantit que les connecteurs booléens sont interprétés de manière naturelle. Il faut aussi contraindre l'interprétation des constructeurs de type. Par exemple, on peut dire que le constructeur $\times$ doit être interprété comme un produit cartésien ensembliste. L'ensemble D dans lequel les types sont interprétés doit alors être tel que $D \times D \subseteq D$. Les choses sont plus compliquées pour le constructeur $\rightarrow$. On pourrait prendre comme interprétation d'un type flèche un ensemble de fonctions ensemblistes. Il faudrait alors avoir $D^D \subseteq D$, ce qui est impossible pour des raisons de cardinalité. De plus, nous voulons pouvoir interpréter *in fine* les types comme des ensembles de valeurs du calcul, donc il faut pouvoir interpréter un type flèche comme un ensemble de λ-abstractions (qui sont des objets syntaxiques, et certainement pas des fonctions ensemblistes).

Ce que nous voulons dire, c'est que dans un modèle, les constructeurs de type se comportent comme s'ils étaient interprétés de manière ensembliste naturelle (même s'il ne sont pas effectivement interprétés ainsi), du point de vue du sous-typage (ou, ce qui est équivalent, du prédicat $[\![t]\!] = \emptyset$, puisque $[\![t]\!] \subseteq [\![s]\!] \iff [\![t \setminus s]\!] = \emptyset$ pour une interprétation ensembliste). La définition formelle ci-dessous formalise cette idée de « comme si ».

Definition 4.2 (Interprétation extensionnelle, modèle) *Soit* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ *une interprétation ensembliste de* $\widehat{T} = \mathcal{B}T_A$ *dans un ensemble* D*. On pose :*

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega)$$

où $D_\Omega = D + \{\Omega\}$*. Si* $X, Y \subseteq D$*, on pose :*

$$X \to Y := \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\ d \in X \Rightarrow d' \in Y\} \subseteq \mathcal{P}(D \times D_\Omega)$$

*On définit alors l'***interprétation extensionnelle** *associée à* $[\![\_]\!]$ *comme l'unique interprétation ensembliste* $\mathbb{E}[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ *telle que :*

$$\begin{array}{lcll} \mathbb{E}[\![b]\!] & = & \mathbb{B}[\![b]\!] & \subseteq \ \mathcal{C} \\ \mathbb{E}[\![\theta_1 \times \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \times [\![\tau(\theta_2)]\!] & \subseteq \ D \times D \\ \mathbb{E}[\![\theta_1 \to \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \to [\![\tau(\theta_2)]\!] & \subseteq \ \mathcal{P}(D \times D_\Omega) \end{array}$$

On dit que $[\![\_]\!]$ *est un* **modèle** *de l'algèbre de types si :*

$$\forall t \in \widehat{T}.\ [\![t]\!] = \emptyset \iff \mathbb{E}[\![t]\!] = \emptyset$$

On note $\mathbb{E}_{\mathbf{basic}}D = \mathcal{C}$*,* $\mathbb{E}_{\mathbf{prod}}D = D \times D$*,* $\mathbb{E}_{\mathbf{fun}}D = \mathcal{P}(D \times D_\Omega)$*.*

L'interprétation extensionnelle associée à une interprétation ensembliste donne une interprétation ensembliste naturelle pour les constructeurs de type : les produits sont interprétés comme des produits cartésiens, les flèches sont interprétées comme des ensembles de fonctions. En fait, plutôt que de prendre des graphes fonctionnels, nous considérons des relations binaires, ou, ce qui revient au même, des fonctions non déterministes. Cela est motivé par le fait que le calcul que nous allons introduire peut effectivement posséder des traits non déterministes (sous la forme d'opérateurs, ou si l'on considère une extension avec des effets de bords). De plus, ce choix simplifie les calculs ensemblistes, en évitant des cas limites liés aux ensembles finis. Donnons un exemple (dans lequel nous identifions types et nœuds de types) : considérons le type flèche $t \to (s_1 \vee s_2)$, avec s_1 et s_2 disjoints, et t non vide. Nous nous demandons s'il s'agit d'un sous-type de $(t \to s_1) \vee (t \to s_2)$, c'est-à-dire si toute fonction de type $t \to (s_1 \vee s_2)$ est nécessairement de type $t \to s_i$ pour un certain i. Si l'on considère des fonctions non déterministes, cela n'est pas le cas (car la fonction peut choisir de répondre dans s_1 ou dans s_2 de manière non déterministe). Mais si l'on considère des fonctions ensemblistes, cela devient vrai dans un cas limite, lorsque t est un singleton (car la valeur de la fonction pour ce point est dans $s_1 \vee s_2$, donc dans l'un des deux types s_i, et alors la fonction elle-même est dans $t \to s_i$). Utiliser des fonctions non déterministes permet d'éviter des cas particuliers, difficile à traiter, dans la relation de sous-typage.

L'élément spécial Ω, dans le codomaine des fonctions, représente une éventuelle erreur de type. Si l'on supprime Ω de la définition d'un modèle, on voit immédiatement que tout type flèche est sous-type de $\mathbb{1} \to \mathbb{1}$, c'est-à-dire que n'importe quelle fonction peut être appliquée à n'importe quel argument. Dans un travail antérieur [Fri01], nous avons étudié un tel système. La sémantique du calcul associé doit effectivement pouvoir appliquer n'importe quelle abstraction à n'importe quel argument, sans provoquer d'erreur de type. Pour cela, il faut tester lors de l'appel de la fonction si l'argument est dans le domaine (déclaré

dans l'interface de l'abstraction), et si ce n'est pas le cas, il faut éviter d'évaluer le corps de la fonction (et il est légal de renvoyer n'importe quelle valeur, par exemple une valeur particulière pour signaler une erreur). Le système que nous présentons ici garantit que lorsque l'on évalue une application bien typée, l'argument est dans le domaine de la fonction. Il s'agit d'une illustration du principe suivant : la définition du modèle doit capturer de manière suffisamment précise la sémantique du calcul à typer. Nous donnerons plus bas (Section 5.9) une autre illustration de ce principe.

La définition d'un modèle contraint l'interprétation « locale » des constructeurs de type. Il manque une contrainte pour garantir que le sous-typage induit par un modèle correspond exactement à celui d'une interprétation des types comme ensemble de valeurs. Considérons un type récursif $\tau(\theta) = \theta \times \theta$. Rien dans la définition d'un modèle n'assure que $[\![\tau(\theta)]\!]$ est l'ensemble vide (en fait, on peut construire des modèles pour lesquels cet ensemble n'est pas vide). Mais une valeur v de ce type doit être un couple (v_1, v_2) où v_1 et v_2 sont encore de ce type ; on voit ainsi que v est nécessairement un arbre infini, mais le calcul que nous allons introduire ne permet pas de créer de telles valeurs infinies. Nous souhaitons donc que l'interprétation de $\tau(\theta)$ soit vide, et nous introduisons une notion de modèle bien fondé pour garantir cette propriété.

Definition 4.3 *Un modèle $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ est* **bien fondé** *s'il existe un ordre bien fondé $\lhd$ sur D tel que :*

$$\forall t \in \widehat{T}. \forall d \in [\![t]\!]. \exists d' \in \mathbb{E}[\![t]\!]. d' = (d_1, d_2) \Rightarrow (d_1 \lhd d) \wedge (d_2 \lhd d)$$

Definition 4.4 *Un modèle $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ est* **structurel** *si D est la solution initiale d'une équation de la forme :*

$$D = \mathcal{C} + D \times D + D_{\mathbf{fun}}$$

pour un certain ensemble $D_{\mathbf{fun}}$, et si :

$$\begin{array}{lcl} [\![b]\!] & = & \mathbb{E}[\![b]\!] \\ [\![\theta_1 \times \theta_2]\!] & = & \mathbb{E}[\![\theta_1 \times \theta_2]\!] \\ [\![\theta_1 \to \theta_2]\!] & \subseteq & D_{\mathbf{fun}} \end{array}$$

Lemme 4.5 *Un modèle structurel est bien fondé.*

> *Preuve*: Il suffit de considérer l'ordre $\lhd$ bien fondé induit par les relations $d_1 \lhd (d_1, d_2)$ et $d_2 \lhd (d_1, d_2)$ pour tout $(d_1, d_2) \in D \times D$. □

4.3 Inclusions ensemblistes

Pour préparer l'étude du sous-typage dans les modèles, nous aurons besoin d'établir quelques propriétés ensemblistes simples. La propriété exprimée dans le lemme ci-dessous a servi de base à l'algorithme de sous-typage descendant introduit dans XDuce [Hos01].

Lemme 4.6 *Soient $(X_i)_{i \in P}, (X_i)_{i \in N}$ (resp. $(Y_i)_{i \in P}, (Y_i)_{i \in N}$) deux familles de parties d'un ensemble D_1 (resp. D_2). Alors :*

$$\left(\bigcap_{i \in P} X_i \times Y_i\right) \setminus \left(\bigcup_{i \in N} X_i \times Y_i\right) = \bigcup_{N' \subseteq N} \left(\bigcap_{i \in P} X_i \setminus \bigcup_{i \in N'} X_i\right) \times \left(\bigcap_{i \in P} Y_i \setminus \bigcup_{i \in N \setminus N'} Y_i\right)$$

(avec les conventions : $\bigcap_{i\in\emptyset} X_i \times Y_i = D_1 \times D_2$ *;* $\bigcap_{i\in\emptyset} X_i = D_1$ *et* $\bigcap_{i\in\emptyset} Y_i = D_2$*)*
En particulier :

$$\bigcap_{i\in P} X_i \times Y_i \subseteq \bigcup_{i\in N} X_i \times Y_i$$
$$\Longleftrightarrow$$
$$\forall N' \subseteq N.\ \left(\bigcap_{i\in P} X_i \subseteq \bigcup_{i\in N'} X_i\right) \vee \left(\bigcap_{i\in P} Y_i \subseteq \bigcup_{i\in N\setminus N'} Y_i\right)$$

Preuve: On commence par remarquer que :

$$\overline{X_i \times Y_i}^{D_1\times D_2} = \left(\overline{X_i}^{D_1} \times D_2\right) \cup \left(D_1 \times \overline{Y_i}^{D_2}\right)$$

On en déduit :

$$\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2} =$$
$$\bigcup_{N'\subseteq N} \left(\bigcap_{i\in N'} \left(\overline{X_i}^{D_1} \times D_2\right) \cap \bigcap_{i\in N\setminus N'} \left(D_1 \times \overline{Y_i}^{D_2}\right)\right)$$
$$\bigcup_{N'\subseteq N} \left(\bigcap_{i\in N'} \overline{X_i}^{D_1} \times \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)$$

Et finalement :

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \cap \left(\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2}\right) =$$
$$\bigcup_{N'\subseteq N} \left(\left(\bigcap_{i\in P} X_i \cap \bigcap_{i\in N'} \overline{X_i}^{D_1}\right) \times \left(\bigcap_{i\in P} Y_i \cap \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)\right)$$

□

Nous allons maintenant établir l'analogue du Lemme 4.6 pour les types flèche. Nous commençons par décomposer l'opérateur ensembliste $\rightarrow$ en des opérateurs plus simples : ensemble des parties, complémentaire, produit cartésien.

Lemme 4.7 *Soient* $X, Y \subseteq D$*. Alors :*

$$X \rightarrow Y = \mathcal{P}\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\right)$$

Preuve: Le résultat provient d'un simple calcul :

$$\begin{aligned} X \rightarrow Y &= \{f \subseteq D \times D_\Omega \mid \forall (x,y) \in f.\ \neg(x \in X \wedge y \notin Y)\} \\ &= \{f \subseteq D \times D_\Omega \mid f \cap X \times \overline{Y}^{D_\Omega} = \emptyset\} \\ &= \{f \subseteq D \times D_\Omega \mid f \subseteq \overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\} \end{aligned}$$

□

Lemme 4.8 *Soit $(X_i)_{i\in P}$ et $(X_i)_{i\in N}$ deux familles de parties d'un ensemble D. Alors :*

$$\bigcap_{i\in P}\mathcal{P}(X_i)\subseteq\bigcup_{i\in N}\mathcal{P}(X_i)\iff\exists i_o\in N.\ \bigcap_{i\in P}X_i\subseteq X_{i_0}$$

Preuve: L'implication $\Leftarrow$ est triviale. Montrons l'implication inverse, et supposons donc que $\bigcap_{i\in P}\mathcal{P}(X_i)\subseteq\bigcup_{i\in N}\mathcal{P}(X_i)$. L'ensemble $\bigcap_{i\in P}X_i$ appartient à tous les $\mathcal{P}(X_i)$ pour $i\in P$, donc il est dans la réunion des $\mathcal{P}(X_i)$ pour $i\in N$. On peut donc trouver un $i_0\in N$ tel que $\bigcap_{i\in P}X_i\in\mathcal{P}(X_{i_0})$, ce qui conclut la preuve. □

Lemme 4.9 *Soient $(X_i)_{i\in P},(X_i)_{i\in N}$, $(Y_i)_{i\in P},(Y_i)_{i\in N}$ quatre familles de parties d'un ensemble D. Alors :*

$$\bigcap_{i\in P}X_i\to Y_i\subseteq\bigcup_{i\in N}X_i\to Y_i$$
$$\iff$$
$$\exists i_0\in N.\ \forall P'\subseteq P.\ \left(X_{i_0}\subseteq\bigcup_{i\in P'}X_i\right)\vee\left\{\begin{array}{l}P\neq P'\\\left(\bigcap_{i\in P\setminus P'}Y_i\subseteq Y_{i_0}\right)\end{array}\right.$$

(avec la convention : $\bigcap_{i\in\emptyset}X_i\to Y_i=\mathcal{P}(D\times D_\Omega)$)

Preuve: Corollaire des Lemmes 4.7, 4.8, 4.6, en remarquant que dans la condition $\bigcap_{i\in P\setminus P'}Y_i\subseteq Y_{i_0}$ qui apparaît, la convention est d'interpréter l'intersection comme valant D_Ω si $P=P'$, ce qui rend impossible l'inclusion. □

Corollaire 4.10 *Soit $[\![\_]\!]:\widehat{T}\to\mathcal{P}(D)$ un modèle, et $(\theta_i)_{i\in P}$, $(\theta_i)_{i\in N}$, $(\theta'_i)_{i\in P}$, $(\theta'_i)_{i\in N}$ quatre familles de nœuds de type. Alors :*

$$\bigwedge_{i\in P}\theta_i\to\theta'_i\leq\bigvee_{i\in N}\theta_i\to\theta'_i$$
$$\iff$$
$$\left\{\begin{array}{l}P\neq\emptyset\\\exists i_0\in N.\ \forall P'\subseteq P.\ \left(\tau(\theta_{i_0})\leq\bigvee_{i\in P'}\tau(\theta_i)\right)\vee\left\{\begin{array}{l}P\neq P'\\\left(\bigwedge_{i\in P\setminus P'}\tau(\theta'_i)\leq\tau(\theta'_{i_0})\right)\end{array}\right.\end{array}\right.$$

4.4 Analyse du sous-typage

Nous avons maintenant tous les éléments nécessaires pour exprimer le prédicat $\mathbb{E}[\![t]\!]=\emptyset$ en fonction du prédicat $[\![t]\!]=\emptyset$, et ainsi obtenir une définition plus effective pour vérifier qu'une interprétation ensembliste est un modèle.

Soit t un type. On a :

$$t\simeq\bigvee_{(P,N)\in t}\bigwedge_{a\in P}a\wedge\bigwedge_{a\in N}\neg a$$

L'interprétation extensionnelle de ce type est vide si et seulement si celle de chacun des termes de la réunion l'est. C'est le cas en particulier pour $(P, N) \in t$ dès que P contient deux atomes de genre différent, puisque les $\mathbb{E}_u D$ sont deux-à-deux disjoints (pour $u \in \{\mathbf{basic}, \mathbf{prod}, \mathbf{fun}\}$). Lorsque tous les atomes sont des produits (resp. des flèches), on peut utiliser le Lemme 4.6 (resp. 4.9) pour décomposer le prédicat $\mathbb{E}[\![\bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg a]\!] = \emptyset$ en une formule booléenne sur des prédicats de la forme $[\![t]\!] = \emptyset$, pour des types t obtenus comme combinaisons booléennes des $\tau(\theta)$, pour les θ qui apparaissent dans les atomes a. Ce raisonnement est formalisé dans la définition ci-dessous et le Théorème 4.13.

Definition 4.11 *Soit $\mathcal{S} \subseteq \widehat{T}$. On définit :*

$$\mathbb{E}\mathcal{S} = \{t \in \widehat{T} \mid \forall (P, N) \in t. \forall u.\ (P \subseteq T_u \Rightarrow C_u)\}$$

où :

$$C_{\mathbf{basic}} \quad ::= \quad \mathcal{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!]$$

$$C_{\mathbf{prod}} \quad ::= \quad \forall N' \subseteq N \cap T_{\mathbf{prod}}. \begin{cases} \left(\bigwedge_{\theta_1\times\theta_2\in P} \tau(\theta_1)\right) \setminus \left(\bigvee_{\theta_1\times\theta_2\in N'} \tau(\theta_1)\right) \in \mathcal{S} \\ \qquad \vee \\ \left(\bigwedge_{\theta_1\times\theta_2\in P} \tau(\theta_2)\right) \setminus \left(\bigvee_{\theta_1\times\theta_2\in N\setminus N'} \tau(\theta_2)\right) \in \mathcal{S} \end{cases}$$

$$C_{\mathbf{fun}} \quad ::= \quad \exists \theta'_1 \to \theta'_2 \in N.\ \forall P' \subseteq P. \begin{cases} \tau(\theta'_1) \setminus \left(\bigvee_{\theta_1\times\theta_2\in P'} \tau(\theta_1)\right) \in \mathcal{S} \\ \qquad \vee \\ \begin{cases} P \neq P' \\ \left(\bigwedge_{\theta_1\times\theta_2\in P\setminus P'} \tau(\theta_2)\right) \setminus \tau(\theta'_2) \in \mathcal{S} \end{cases} \end{cases}$$

On dit que $\mathcal{S}$ est une **simulation** *si :*

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S}$$

Remarque 4.12 *La définition de $\mathbb{E}\mathcal{S}$ ne dépend pas d'un modèle fixé ou d'une interprétation ensembliste. C'est une définition purement axiomatique. Pour un type t et $(P, N) \in t$, il existe au plus un u tel que $P \subseteq T_u$, sauf si $P = \emptyset$, auquel cas il faut vérifier les trois conditions C_u.*

Theorème 4.13 *Soit $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ une interprétation ensembliste. Posons :*

$$\mathcal{S} = \{t \mid [\![t]\!] = \emptyset\}$$

Alors :

$$\mathbb{E}\mathcal{S} = \{t \mid \mathbb{E}[\![t]\!] = \emptyset\}$$

Preuve: Soit $t \in \widehat{T}$. On a :

$$\mathbb{E}[\![t]\!] = \bigcup_{(P,N)\in t} \bigcap_{a\in P} \mathbb{E}[\![a]\!] \setminus \bigcup_{a\in N} \mathbb{E}[\![a]\!]$$

Et donc :

$$\mathbb{E}[\![t]\!] = \emptyset \iff \forall (P,N) \in t. \bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a \in N} \mathbb{E}[\![a]\!]$$

Décomposons sur chaque univers :

$$\bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a \in N} \mathbb{E}[\![a]\!] \iff \forall u.\ \mathbb{E}_u D \cap \bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \mathbb{E}_u D \cap \bigcup_{a \in N} \mathbb{E}[\![a]\!]$$

Notons que pour tout atome a, si $a \in T_u$, alors $\mathbb{E}_u D \cap \mathbb{E}[\![a]\!] = \mathbb{E}[\![a]\!]$, et si $a \notin T_u$, alors $\mathbb{E}_u D \cap \mathbb{E}[\![a]\!] = \emptyset$. Ainsi :

$$\mathbb{E}_u D \cap \bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \mathbb{E}_u D \cap \bigcup_{a \in N} \mathbb{E}[\![a]\!]$$

$$\iff$$

$$\left(P \subseteq T_u \Rightarrow \bigcap_{a \in P} \mathbb{E}[\![a]\!] \subseteq \bigcup_{a \in N \cap T_u} \mathbb{E}[\![a]\!] \right)$$

On conclut alors avec les Lemmes 4.9 et 4.6. □

Corollaire 4.14 *Avec les notations du théorème précédent, l'interprétation* $[\![\_]\!]$ *est un modèle si et seulement si* $\mathcal{S} = \mathbb{E}\mathcal{S}$.

Corollaire 4.15 *Soit* $[\![\_]\!]_1 : \widehat{T} \to D_1$ *un modèle et* $[\![\_]\!]_2 : \widehat{T} \to D_2$ *une interprétation ensembliste. Alors les deux assertions ci-dessous sont équivalentes :*

- $[\![\_]\!]_2$ *est un modèle et il induit la même relation de sous-typage que* $[\![\_]\!]_1$
- *pour tout type* $t \in \widehat{T}$, $[\![t]\!]_1 = \emptyset \iff [\![t]\!]_2 = \emptyset$

Ce corollaire dit que le fait qu'une interprétation ensembliste est un modèle ou non ne dépend que de la relation de sous-typage qu'elle induit. Autrement dit, comme attendu, notre définition de modèle ne fait que contraindre la relation de sous-typage induite par l'interprétation ensembliste, et non la nature des éléments de l'ensemble utilisé pour interpréter les types.

4.5 Modèle universel

Definition 4.16 (Modèle universel) *Un modèle* $[\![\_]\!]^0$ *est* **universel** *si, pour tout modèle* $[\![\_]\!]$*, on a :*

$$\forall t \in \widehat{T}.\ [\![t]\!] = \emptyset \Rightarrow [\![t]\!]^0 = \emptyset$$

Un modèle est universel s'il induit la plus grande relation de sous-typage possible. Intuitivement, disposer d'une relation de sous-typage la plus grosse possible permet de typer plus d'expressions dans le calcul du chapitre suivant, compte-tenu de la règle de subsomption (même si formellement, cette propriété est fausse dans le système de types que nous allons introduire, à cause de la règle de typage des abstractions).

Dans cette section, nous allons montrer comment construire un modèle universel. Évidemment, tous les modèles universels induisent la même relation de sous-typage.

La propriété qui définit la notion de modèle impose une correspondance entre les interprétations ensemblistes $[\![\_]\!]$ et $\mathbb{E}[\![\_]\!]$. Pour construire un modèle, il est naturel de considérer l'équation suivante :

$$D = \mathbb{E}D$$

Pour une simple raison de cardinalité, il n'est pas possible de construire un ensemble D qui vérifie cette équation. Le problème vient du fait que le cardinal de $\mathcal{P}(D \times D_\Omega)$ est strictement plus grand que le cardinal de D. Les lemmes suivants montrent qu'en ne considérant que des graphes finis, nous ne changeons pas les relations d'inclusions ensemblistes, et c'est ce qui est important pour obtenir un modèle.

Lemme 4.17 *Si l'on note $X \to_f Y = (X \to Y) \cap \mathcal{P}_f(D \times D_\Omega)$, alors :*

$$X \to_f Y = \mathcal{P}_f\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D \times D_\Omega}\right)$$

Lemme 4.18 *Soit $(X_i)_{i\in P}$ et $(X_i)_{i\in N}$ deux familles finies de parties d'un ensemble D. Alors :*

$$\bigcap_{i\in P} \mathcal{P}_f(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}_f(X_i) \iff \bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i)$$

Preuve: L'implication $\Leftarrow$ est immédiate. Prouvons l'implication $\Rightarrow$. On suppose donc que toute partie finie de $X = \bigcap_{i\in P} X_i$ est incluse dans un certain $\mathcal{P}(X_{i_0})$ avec $i_0 \in N$. Si ce n'était pas le cas pour X lui-même, on pourrait trouver pour chaque $i_0 \in N$ un certain élément $x_{i_0} \in X \backslash X_{i_0}$, et on obtiendrait une contradiction en considérant l'ensemble de ces x_{i_0} (qui est une partie finie de X). □

Lemme 4.19 *Soient $(X_i)_{i\in P}, (X_i)_{i\in N}$, $(Y_i)_{i\in P}, (Y_i)_{i\in N}$ quatre familles finies de parties d'un ensemble D. Alors :*

$$\bigcap_{i\in P} X_i \to_f Y_i \subseteq \bigcup_{i\in N} X_i \to_f Y_i \quad \iff \quad \bigcap_{i\in P} X_i \to Y_i \subseteq \bigcup_{i\in N} X_i \to Y_i$$

(avec la convention : $\bigcap_{i\in\emptyset} X_i \to_f Y_i = \mathcal{P}_f(D \times D_\Omega)$)

Preuve: Corollaire des Lemmes 4.17 et 4.18. □

Definition 4.20 *Soit $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ une interprétation ensembliste dans un ensemble D. On pose :*

$$\mathbb{E}_f D := \mathcal{C} + D \times D + \mathcal{P}_f(D \times D_\Omega)$$

et on définit alors l'<u>**interprétation extensionnelle finie**</u> *associée à $[\![\_]\!]$ comme l'unique interprétation ensembliste $\mathbb{E}_f[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ telle que :*

$$\begin{array}{lcll} \mathbb{E}_f[\![b]\!] & = & \mathbb{B}[\![b]\!] & \subseteq \ \mathcal{C} \\ \mathbb{E}_f[\![\theta_1 \times \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \times [\![\tau(\theta_2)]\!] & \subseteq \ D \times D \\ \mathbb{E}_f[\![\theta_1 \to \theta_2]\!] & = & [\![\tau(\theta_1)]\!] \to_f [\![\tau(\theta_2)]\!] & \subseteq \ \mathcal{P}_f(D \times D_\Omega) \end{array}$$

Lemme 4.21 *Soit* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ *une interprétation ensembliste. Alors :*

$$\mathbb{E}[\![t]\!] = \emptyset \iff \mathbb{E}_f[\![t]\!] = \emptyset$$

Cela suggère de considérer l'équation $D = \mathbb{E}_f D$, qui, contrairement à l'équation $D = \mathbb{E}D$, possède bien des solutions. Nous allons considérer la solution initiale D^0, constituée des « termes » finis engendrés par les productions :

$$\begin{array}{rcl} d \in D^0 & ::= & c \\ & | & (d_1, d_2) \\ & | & \{(d_1, d'_1), \ldots, (d_n, d'_n)\} \end{array}$$

Dans la dernière production, les d'_i sont des éléments de $D^0_\Omega = D^0 + \{\Omega\}$. On a bien $D = \mathbb{E}_f D$.

Theorème 4.22 *Il existe une unique interprétation ensembliste* $[\![\_]\!]^0 : \widehat{T} \to \mathcal{P}(D^0)$ *qui vérifie* $[\![t]\!]^0 = \mathbb{E}_f[\![t]\!]^0$*. Il s'agit d'un modèle structurel.*

> *Preuve*: Le Lemme 3.4 permet de traduire la condition « $[\![\_]\!]^0$ est une interprétation ensembliste et $[\![\_]\!]^0 = \mathbb{E}_f[\![\_]\!]^0$ » en une définition récursive de la valeur de vérité de l'assertion « $d \in [\![t]\!]^0$ », par induction sur la structure de d et simultanément pour tous les $t \in \widehat{T}$:
>
> $$\begin{array}{l} c \in [\![t]\!]^0 \iff \\ \quad \exists (P, N) \in t.\ (P \subseteq T_{\mathbf{basic}}) \wedge \\ \quad\quad (\forall b \in P.c \in \mathbb{B}[\![b]\!]) \wedge \\ \quad\quad (\forall b \in N.c \notin \mathbb{B}[\![b]\!]) \\ (d_1, d_2) \in [\![t]\!]^0 \iff \\ \quad \exists (P, N) \in t.\ (P \subseteq T_{\mathbf{prod}}) \wedge \\ \quad\quad (\forall \theta_1 \times \theta_2 \in P.d_1 \in [\![\tau(\theta_1)]\!]^0 \wedge d_2 \in [\![\tau(\theta_2)]\!]^0) \wedge \\ \quad\quad (\forall \theta_1 \times \theta_2 \in N.d_1 \notin [\![\tau(\theta_1)]\!]^0 \vee d_2 \notin [\![\tau(\theta_2)]\!]^0) \\ f \in [\![t]\!]^0 \iff \\ \quad \exists (P, N) \in t.\ (P \subseteq T_{\mathbf{fun}}) \wedge \\ \quad\quad (\forall \theta_1 \to \theta_2 \in P. \forall (d, d') \in f.\ d \in [\![\tau(\theta_1)]\!]^0 \Rightarrow d' \in [\![\tau(\theta_2)]\!]^0) \wedge \\ \quad\quad (\forall \theta_1 \to \theta_2 \in N. \exists (d, d') \in f.\ d \in [\![\tau(\theta_1)]\!]^0 \wedge d' \notin [\![\tau(\theta_2)]\!]^0) \end{array}$$
>
> Cela montre l'existence et l'unicité d'une interprétation ensembliste $[\![\_]\!]^0$ qui vérifie la condition $[\![\_]\!]^0 = \mathbb{E}_f[\![\_]\!]^0$. On voit immédiatement qu'il s'agit d'un modèle avec le Corollaire 4.14 et le Lemme 4.21. Il est clairement structurel. □

Lemme 4.23 *Soit* $\mathcal{S} \subseteq \widehat{T}$ *une simulation et* $t \in \mathcal{S}$*. Alors* $[\![t]\!]^0 = \emptyset$*.*

> *Preuve*: Soit $\mathcal{S}$ une simulation. Nous allons montrer par récurrence sur la structure de $d \in D^0$ la propriété ci-dessous :
>
> $$\forall t \in \widehat{T}.\ d \in [\![t]\!]^0 \Rightarrow t \notin \mathcal{S}$$
>
> Supposons donc cette propriété vérifiée pour tous les sous-éléments d'un certain $d \in D^0$. Prenons un type $t \in \widehat{T}$ tel que

$d \in [\![t]\!]^0$. Il s'agit de montrer que $t \notin \mathcal{S}$. Comme $\mathcal{S}$ est une simulation, il suffit donc de montrer que partant de l'hypothèse $t \in \mathbb{E}\mathcal{S}$, nous arrivons à une contradiction.

Soit $u \in \{\mathbf{basic}, \mathbf{fun}, \mathbf{prod}\}$ l'univers de d. On choisir $(P, N) \in t$ de sorte que :

$$d \in \bigcap_{a \in P} [\![a]\!]^0 \setminus \bigcup_{a \in N} [\![a]\!]^0$$

Pour tout $a \in P$, on a $d \in [\![a]\!]^0$, et donc $P \subseteq T_u$.

On obtient alors que la propriété C_u de la Définition 4.11 est vérifiée pour ce couple (P, N). Distinguons alors suivant la valeur de u.

Pour $u = \mathbf{basic}$, on obtient directement une contradiction.

Pour $u = \mathbf{prod}$, posons $d = (d_1, d_2)$. On peut écrire :

$$(d_1, d_2) \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \times [\![\tau(\theta_2)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N} [\![\tau(\theta_1)]\!]^0 \times [\![\tau(\theta_2)]\!]^0$$

D'après le Lemme 4.6, on peut trouver $N' \subseteq N \cap T_{\mathbf{prod}}$ tel que :

$$d_1 \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N'} [\![\tau(\theta_1)]\!]^0$$

et

$$d_2 \in \bigcap_{\theta_1 \times \theta_2 \in P} [\![\tau(\theta_2)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in N \setminus N'} [\![\tau(\theta_2)]\!]^0$$

En appliquant l'hypothèse de récurrence à d_1 et au type $t_1 = \bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_1) \setminus \bigvee_{\theta_1 \times \theta_2 \in N'} \tau(\theta_1)$, on obtient : $t_1 \notin \mathcal{S}$. De même, pour $t_2 = \bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_2) \setminus \bigvee_{\theta_1 \times \theta_2 \in N \setminus N'} \tau(\theta_2)$. On obtient bien une contradiction avec la condition $C_{\mathbf{prod}}$.

Pour $u = \mathbf{fun}$, posons $d = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$. Choisissons un élément $\theta'_1 \to \theta'_2 \in N$ tel que donné par la condition $C_{\mathbf{fun}}$. Comme $d \notin [\![\theta'_1 \to \theta'_2]\!]^0 = [\![\tau(\theta'_1)]\!]^0 \to [\![\tau(\theta'_2)]\!]^0$, on peut trouver i tel que :

$$(d_i, d'_i) \in [\![\tau(\theta'_1)]\!]^0 \times \overline{[\![\tau(\theta'_2)]\!]^0}^{D^0_\Omega}$$

Ce couple (d_i, d'_i) est donc dans l'ensemble :

$$[\![\tau(\theta'_1)]\!]^0 \times \overline{[\![\tau(\theta'_2)]\!]^0}^{D^0_\Omega} \setminus \bigcup_{\theta_1 \to \theta_2 \in P} [\![\tau(\theta_1)]\!]^0 \times \overline{[\![\tau(\theta_2)]\!]^0}^{D^0_\Omega}$$

En raisonnant comme dans le cas des produits, on voit qu'il existe un $P' \subseteq P$ tel que :

$$d_i \in [\![\tau(\theta'_1)]\!]^0 \setminus \bigcup_{\theta_1 \times \theta_2 \in P'} [\![\tau(\theta_1)]\!]^0$$

et

$$d'_i \in D^0_\Omega \cap \left(\bigcap_{\theta_1 \times \theta_2 \in P \setminus P'} [\![\tau(\theta_2)]\!]^0 \right) \setminus [\![\tau(\theta'_2)]\!]^0$$

| En appliquant l'hypothèse de récurrence à d_i et au type $t_1 = \tau(\theta'_1) \setminus \bigvee_{\theta_1 \times \theta_2 \in P'} \tau(\theta_1)$, on obtient $t_1 \notin \mathcal{S}$. De plus, si $P' \neq P$, alors $d'_i \neq \Omega$, et on peut appliquer l'hypothèse de récurrence à d'_i et au type $t_2 = \left(\bigwedge_{\theta_1 \times \theta_2 \in P \setminus P'} \tau(\theta_2)\right) \setminus \tau(\theta'_2)$. On obtient bien une contradiction. □

Theorème 4.24 *Le modèle $[\![\_]\!]^0$ est universel.*

| *Preuve*: Conséquence du Corollaire 4.14 et du lemme précédent. □

En combinant le Corollaire 4.14 et le Lemme 4.23, on obtient :

Lemme 4.25 *Soit $t \in \widehat{T}$. Alors $[\![t]\!]^0 = \emptyset$ si et seulement s'il existe une simulation $\mathcal{S}$ qui contient t.*

Le lemme ci-dessous va permettre d'obtenir une version plus effective de cette propriété.

Lemme 4.26 *L'intersection d'une simulation et d'un socle est encore une simulation.*

| *Preuve*: Conséquence directe des Définitions 4.11 et 3.7. □

Une simulation est un ensemble de types stable par un certain ensemble de règles de décomposition (données par la définition de $\mathbb{E}\mathcal{S}$). Le Lemme 4.26 montre que pour vérifier si un type t appartient à une certaine simulation, nous n'avons qu'à considérer un nombre fini de types (si t est dans un socle, les types introduits par les règles de décomposition sont encore dans ce même socle). En utilisant ce lemme, on obtient une version plus effective du Lemme 4.25.

Lemme 4.27 *Soit $t \in \widehat{T}$ et $\sqsupseteq$ un socle qui le contient. Alors $[\![t]\!]^0 = \emptyset$ si et seulement s'il existe une simulation $\mathcal{S} \subseteq \sqsupseteq$ qui contient t.*

L'existence d'un socle qui contient t est garantie par le Théorème 3.8. Comme un socle est un ensemble fini, on peut énumérer tous ses sous-ensembles et vérifier s'il en existe un qui est une simulation. Cela donne un algorithme naïf pour calculer le sous-typage induit par le modèle universel. Dans le Chapitre 7, nous donnerons des algorithmes plus efficaces.

4.6 Modèles non universels

Dans cette section, nous allons montrer qu'il existe un modèle qui induit une relation de sous-typage différente de celle induite par le modèle $[\![\_]\!]^0$. Mieux, nous allons construire un modèle structurel qui vérifie cette propriété. Construire un modèle non-universel permet de voir que la définition d'un modèle laisse une certaine dose de liberté, car elle n'axiomatise pas complètement la relation de sous-typage.

Considérons un nœud θ tel que :

$$\tau(\theta) = (\mathbb{0} \to \mathbb{0}) \setminus (\theta \to \mathbb{0})$$

(ici $\mathbb{0}$ est utilisé comme un nœud de type, qui représente le type vide - voir la Section 3.3) et notons $t_0 = \tau(\theta)$. Un calcul montre que :

$$[\![t_0]\!]^0 = \{\{(d_1, d'_1), \ldots, (d_n, d'_n)\} \mid \exists i.\ d_i \in [\![t_0]\!]^0\}$$

Comme les termes de D^0 sont finis, on en déduit immédiatement que $[\![t_0]\!]^0 = \emptyset$ et donc $t_0 \leq_{[\![\_]\!]^0} \mathbb{0}$. Nous allons maintenant construire un modèle $[\![\_]\!] : \widehat{T} \to D$ tel que $[\![t_0]\!] \neq \emptyset$. Pour cela, nous allons ajouter à D^0 des « fonctions » dont le graphe est un arbre de profondeur infinie. Considérons l'ensemble D^1 des termes engendrés par les productions :

$$\begin{array}{rcl} d \in D^1 & ::= & c \\ & | & (d_1, d_2) \\ & | & \{(d_1, d'_1), \ldots, (d_n, d'_n)\} \\ & | & n \end{array}$$

où les d_i désignent des éléments de D^1, les d'_i des éléments de $D^1_\Omega = D^1 + \{\Omega\}$ et n un entier relatif ($n \in \mathbb{Z}$). Définissons D comme le quotient de D^1 par les équations $n \simeq \{(n-1, \Omega)\}$ pour tout $n \in \mathbb{Z}$. Il vérifie l'équation ensembliste :

$$D = \mathbb{E}_f D$$

Theorème 4.28 *Il existe un modèle $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ qui vérifie :*

$$\begin{array}{rcl} [\![\_]\!] & = & \mathbb{E}_f[\![\_]\!] \\ [\![t_0]\!] & \neq & \emptyset \end{array}$$

La preuve du théorème utilise le lemme technique ci-dessous.

Lemme 4.29 *Soit X un ensemble, f une fonction $\mathcal{P}(X) \to \mathcal{P}(X)$. On suppose que tout élément $x \in X$ est dans un certain sous-ensemble fini $\mathcal{S}$ de X vérifiant :*

$$\forall Y \subseteq X.\ f(Y) \cap \mathcal{S} = f(Y \cap \mathcal{S})$$

Soit $Z \subseteq X$. Alors il existe une unique suite $(Y_n)_{n\in\mathbb{Z}}$ telle que :
- $\forall n \in \mathbb{Z}.\ Y_{n+1} = f(Y_n)$
- $\exists n_0 \in \mathbb{N}. \forall n \geq n_0.\ Y_n = f^n(Z)$
- *pour tout $x \in X$, la suite des prédicats $(x \in Y_n)_{n\in\mathbb{N}}$ est périodique*

> *Preuve*: Considérons la suite $(Z_n)_{n\in\mathbb{N}}$ définie par $Z_0 = Z$ et $Z_{n+1} = f(Z_n)$.
>
> Traitons d'abord le cas où X est fini. La suite (Z_n) est à valeur dans l'ensemble $\mathcal{P}(X)$ qui est fini ; elle est donc ultimement périodique. Il y a donc une unique suite périodique $(Y_n)_{n\in\mathbb{Z}}$ qui correspond ultimement à $(Z_n)_{n\in\mathbb{N}}$. Cette suite vérifie bien la formule de récurrence $Y_{n+1} = Y_n$ pour tout $n \in \mathbb{Z}$.
>
> Passons au cas où X est infini. Si $\mathcal{S}$ est un ensemble comme dans l'énoncé, on peut appliquer ce qui précède à la suite $(Z_n \cap \mathcal{S})_{n\in\mathbb{N}}$. Elle est également obtenue par itération de la fonction f, et on se ramène au cas fini. Cela permet de construire une suite $(Y_n \cap \mathcal{S})_{n\in\mathbb{Z}}$. Par unicité dans le cas fini, cela définit bien une suite $(Y_n)_{n\in\mathbb{Z}}$. □

Passons à la preuve du théorème.

Preuve: On reprend la preuve du Théorème 4.22. L'équation $[\![\_]\!] = \mathbb{E}_f[\![\_]\!]$ donne une définition récursive de la valeur de vérité de l'assertion « $d \in [\![t]\!]$ ». Cette récursion est mal fondée, à cause des éléments $n \in \mathbb{Z}$. Il suffit donc, pour définir un modèle, de choisir les ensembles $T_n = \{t \in \widehat{T} \mid n \in [\![t]\!]\}$ pour tout $n \in \mathbb{Z}$, de manière compatible avec l'égalité $n = \{(n-1, \Omega)\}$ dans D. Une fois cela fait, les équations de la preuve du Théorème 4.22 définissent bien la valeur de vérité de l'assertion « $d \in [\![t]\!]$ », et l'on obtient un modèle.

Notons que l'on doit avoir :

$$\{(n-1, \Omega)\} \in [\![\theta_1 \rightarrow \theta_2]\!] \iff n - 1 \notin [\![\tau(\theta_1)]\!]$$

et donc :

$$n \in [\![\theta_1 \rightarrow \theta_2]\!] \iff n - 1 \notin [\![\tau(\theta_1)]\!]$$

ce qui donne, pour tout $t \in \widehat{T}$:

$$t \in T_n \iff \exists (P, N) \in t. \begin{cases} P \subseteq T_{\mathbf{fun}} \\ \forall \theta_1 \rightarrow \theta_2 \in P.\ \tau(\theta_1) \notin T_{n-1} \\ \forall \theta_1 \rightarrow \theta_2 \in N.\ \tau(\theta_1) \in T_{n-1} \end{cases}$$

Retenons que la condition sur les T_n s'exprime par :

$$(*) \quad \forall n \in \mathbb{Z}.\ T_n = f(T_{n-1})$$

où f est un certain opérateur $\mathcal{P}(\widehat{T}) \to \mathcal{P}(\widehat{T})$. Pour tout suite $(T_n)_{n \in \mathbb{Z}}$ qui vérifie cette condition $(*)$, on obtient un unique modèle.

Pour construire une telle suite, on applique le lemme. En effet si $\sqsupset$ est un socle, alors $f(T') \cap \sqsupset = f(T' \cap \sqsupset)$, pour tout $T' \subseteq \widehat{T}$. Le lemme donne bien l'existence d'une suite $(T_n)_{n \in \mathbb{Z}}$ qui vérifie la condition $(*)$. De plus, on peut fixer l'ensemble $Z \subseteq \widehat{T}$ arbitrairement.

Notons que, pour tout $Z \subseteq \widehat{T}$:

$$t_0 \in f(Z) \iff (\mathbb{0} \notin Z) \wedge (t_0 \in Z)$$

et

$$\mathbb{0} \notin f(Z)$$

Donc si l'on choisit Z de sorte que $t_0 \in Z$, et $\mathbb{0} \notin Z$, alors, pour tout $n \in \mathbb{N}$, $t_0 \in f^n(Z)$. On en déduit que $t_0 \in T_n$ pour tout n, ce qui signifie que $\mathbb{Z} \subseteq [\![t_0]\!]$ (et donc en particulier $[\![t_0]\!] \neq \emptyset$). □

Remarque 4.30 *Si l'on dispose de plusieurs nœuds θ tels que $\tau(\theta) = (\mathbb{0} \to \mathbb{0}) \backslash (\theta \to \mathbb{0})$, alors la preuve du théorème montre que l'on peut choisir indépendemment pour chacun si $[\![\tau(\theta)]\!] \cap \mathbb{Z} = \emptyset$ ou si $\mathbb{Z} \subseteq [\![\tau(\theta)]\!]$. Autrement dit, plusieurs types définis par la même équation n'ont pas forcément la même interprétation dans un modèle non universel.*

Remarque 4.31 *Une construction similaire à D^1 permet d'obtenir un modèle*

dans lequel le type $\tau(\theta) = \theta \times \theta$ n'est pas vide. Évidemment, un tel modèle n'est pas bien fondé.

4.7 Conventions de notation

Jusque là, nous avons soigneusement veillé à distinguer les expressions de types $\widehat{T}$, notées t, t_1, t_2, s, ..., des nœuds de types $\mathcal{T}$, notés θ, θ_1,θ_2, ...

Pour simplifier les notations, nous adoptons maintenant des conventions qui nous permettrons de mélanger plus facilement les deux notions.

Tout d'abord, nous surchargeons les crochets sémantiques pour qu'ils s'appliquent directement aux nœuds :

$$[\![\theta]\!] := [\![\tau(\theta)]\!]$$

Plus généralement, nous autorisons les nœuds θ à apparaître dans n'importe quel contexte qui attend une expression de type, le nœud étant interprété comme $\tau(\theta)$. En particulier, cela permet d'utiliser un nœud comme argument (gauche ou droit) d'une relation de sous-typage $\leq$.

Réciproquement, nous nous autorisons à utiliser une expression de type t là où un nœud est attendu. Il y a deux cas :

- Il s'agit d'un contexte de « capture ». Par exemple, dans la phrase « si a est un atome produit, posons $a = t_1 \times t_2$ », les méta-variables t_1 et t_2 opèrent comme des lieurs. Formellement, cette phrase signifie « si a est un atome produit $\theta_1 \times \theta_2$, on pose $t_1 = \tau(\theta_1)$ et $t_2 = \tau(\theta_2)$ ».
- Lorsqu'une expression de type t apparaît là où un nœud est attendu, hors contexte de capture, il faut *choisir* un nœud θ tel que $\tau(\theta) = t$. Cela est toujours possible car θ est surjectif. À chaque fois que nous utiliserons cette convention, il sera clair que le choix de θ ne change pas *in fine* l'objet dont nous parlons ou qu'il s'agit de définir. Par exemple, si l'on écrit $t_1 \to s_1 \leq t_2 \to s_2$, la valeur de vérité de cette assertion ne dépend pas du choix des nœuds pour représenter t_1, t_2, s_1, s_2.

4.8 Raisonnements sémantiques

Dans cette section, nous montrons comment l'approche que nous avons choisie pour définir le sous-typage, en passant par une notion de modèle ensembliste, permet d'obtenir facilement des propriétés sur la relation de sous-typage, sans considérer un éventuel algorithme de sous-typage.

Nous fixons un modèle, et l'on note $\leq$ au lieu de $\leq_{[\![\_]\!]}$ la relation de sous-typage qu'il induit. Nous supposons que cette relation de sous-typage est décidable (c'est le cas pour les modèles universels lorsque le sous-typage est décidable pour les types de base). Nous noterons $\simeq$ pour l'équivalence induite par ce sous-typage : $t_1 \simeq t_2 \iff (t_1 \leq t_2) \wedge (t_2 \leq t_1)$

Lemme 4.32 *La relation $\leq$ est un préordre (transitive et reflexive). La relation $\simeq$ est une relation d'équivalence.*

Lemme 4.33 *Les opérateurs $\vee$ et $\wedge$ sont covariants en leurs deux arguments.*

4.8.1 Décompositions

Lemme 4.34 $(t_1 \times t_2) \wedge (t'_1 \times t'_2) \simeq (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$

Preuve: Ce genre de propriété s'observe directement sur l'interprétation extensionnelle, en utilisant que $[\![\_]\!]$ et $\mathbb{E}[\![\_]\!]$ sont toutes deux des interprétations ensemblistes. Détaillons la preuve à titre d'exemple, en posant $t''_i = t_i \wedge t'_i$:

$$\begin{aligned}
\mathbb{E}[\![t''_1 \times t''_2]\!] &= [\![t''_1]\!] \times [\![t''_2]\!] \\
&= ([\![t_1]\!] \cap [\![t'_1]\!] \times ([\![t_2]\!] \cap [\![t'_2]\!]) \\
&= ([\![t_1]\!] \times [\![t_2]\!]) \cap ([\![t'_1]\!] \times [\![t'_2]\!]) \\
&= \mathbb{E}[\![t_1 \times t'_1]\!] \cap \mathbb{E}[\![t_2 \times t'_2]\!] \\
&= \mathbb{E}[\![(t_1 \times t'_1) \wedge (t_2 \times t'_2)]\!]
\end{aligned}$$

□

Voici un autre exemple de propriété qui « se voit » sur l'interprétation extensionnelle :

Lemme 4.35 $(t_1 \times t_2) \setminus (t'_1 \times t'_2) \simeq (t_1 \setminus t'_1) \times t_2 \vee t_1 \times (t_2 \setminus t'_2)$

En combinant ces deux derniers lemmes, on obtient le résultat ci-dessous, qui permet de décomposer tout type constitué t tel que $t \leq \mathbb{1} \times \mathbb{1}$ (c'est-à-dire $\mathbb{E}[\![t]\!] \subseteq D \times D$) sous la forme d'une réunion finie de produits cartésiens.

Theorème 4.36 *Soit t un type tel que $t \leq \mathbb{1} \times \mathbb{1}$. Alors il existe un ensemble fini de couples de types $\pi(t)$ tel que :*

- $t \simeq \bigvee_{(t_1,t_2) \in \pi(t)} t_1 \times t_2$
- $\forall (t_1, t_2) \in \pi(t).\ t_1 \not\simeq \mathbb{0}, t_2 \not\simeq \mathbb{0}$
- *si $\sqsupset$ est un socle qui contient t, alors :* $\forall (t_1, t_2) \in \pi(t).\ t_1 \in \sqsupset, t_2 \in \sqsupset$

On voit π comme une fonction partielle $\widehat{T} \to \mathcal{P}_f(\widehat{T}^2)$, définie sur les types t tels que $t \leq \mathbb{1} \times \mathbb{1}$.

Preuve: On commence par écrire :

$$t \simeq \bigvee_{(P,N) \in t \ \mid \ P \subseteq T_{\mathbf{prod}}} (\mathbb{1} \times \mathbb{1}) \wedge \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N \cap T_{\mathbf{prod}}} \neg a$$

En utilisant alors les Lemmes 4.34 et 4.35, on voit que l'on peut écrire t sous la forme d'une réunion d'atomes de la forme $t_1 \times t_2$ où t_1 (resp. t_2) s'exprime comme une combinaison booléenne des $\tau(\theta_1)$ (resp. $\tau(\theta_2)$) où $a = \theta_1 \times \theta_2 \in P \cup N$ pour un certain $(P, N) \in t$. Donc les t_i sont bien dans $\sqsupset$ si $\sqsupset$ est un socle qui contient t. Il ne reste plus qu'à supprimer les atomes $t_1 \times t_2$ tels que $t_1 \simeq \mathbb{0}$ ou $t_2 \simeq \mathbb{0}$. □

Cette décomposition des sous-types de $\mathbb{1} \times \mathbb{1}$ sous forme d'une réunion fini de types produits sera utilisée à de nombreuses reprises dans la suite de l'exposé. Plusieurs fonctions π répondent aux conditions de l'énoncé, et il n'est pas nécessaire de spécifier celle que l'on choisit. Autrement dit, nous n'utiliserons que les propriétés de l'énoncé lorsque nous aurons à manipuler π. En particulier, aucune propriété ne permet *a priori* de mettre en relation $\pi(t)$ et $\pi(s)$,

lorsque $t \leq s$. Il existe cependant des choix de π qui permettent d'obtenir ce genre de relations ; nous allons décrire informellement comment les obtenir. Soit $t \leq \mathbb{1}\times\mathbb{1}$. Un type produit $t_1\times t_2 \leq t$ est dit maximal s'il est non vide et si $t_1\times t_2 \leq t'_1\times t'_2 \leq t \Rightarrow t'_1\times t'_2 \leq t_1\times t_2$. Le nombre de tels types, modulo équivalence, est fini, et si l'on choisit un représentant dans chaque classe d'équivalence, on obtient une décomposition π définie de manière unique (modulo équivalence). Cette décomposition vérifie la propriété ci-dessous, pour des types t et s tels que $t \leq s$:

$$\forall(t_1, t_2) \in \pi(t).\exists(t'_1, t'_2) \in \pi(s).\ t_1 \leq t'_1 \wedge t_2 \leq t'_2$$

Autrement dit, un type plus petit a une décomposition plus fine. En pratique, on peut calculer cette décomposition en produits maximaux à partir d'une décomposition π arbitraire, en « découpant » et en « fusionnant » ses éléments (via des combinaisons booléennes sur chaque composante).

Le résultat suivant donne une décomposition analogue à π pour les types flèche.

Lemme 4.37 *Soit t un type tel que $t \leq \mathbb{0}\to\mathbb{1}$. Alors on peut calculer un type $Dom(t)$ et un ensemble fini de couples de types $\rho(t)$ tels que :*

$$\forall t_1, t_2.\ (t \leq t_1\to t_2) \iff \begin{cases} t_1 \leq Dom(t) \\ \forall(s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq t_2) \end{cases}$$

et tels que si t est dans un socle $\sqsupset$, alors $Dom(t)$ et les types de $\rho(t)$ sont aussi dans $\sqsupset$.

> *Preuve*: La propriété est vraie pour $t \simeq \mathbb{0}$ en prenant $\mathrm{Dom}(t) = \mathbb{1}$ et $\rho(t) = \emptyset$. Si la propriété est vraie pour deux types t et t', alors elle est vraie pour leur réunion $t\vee t'$ (en prenant $\mathrm{Dom}(t\vee t') = \mathrm{Dom}(t)\wedge\mathrm{Dom}(t')$ et $\rho(t\vee t') = \rho(t) \cup \rho(t')$). Or l'on peut écrire :
>
> $$t \simeq \bigvee_{(P,N)\in t} \bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg a$$
>
> et cela permet donc de se ramener au cas où t est une intersection de types flèche et de négations de types flèche, avec de plus $t \not\simeq \mathbb{0}$. On conclut alors facilement avec le Lemme 4.9. □

Remarque 4.38 *On constate facilement que $Dom(t)$ est une plus petite solution s (au sens du sous-typage) de l'inéquation $t \leq s\to\mathbb{1}$. On en déduit en particulier que cette fonction est contravariante : si $t \leq t' \leq \mathbb{0}\to\mathbb{1}$, alors $Dom(t') \leq Dom(t)$.*

Remarque 4.39 *La finitude de l'ensemble $\rho(t)$ exprime le fait que même si l'on dispose de fonctions arbitrairement complexes dans les modèles, au niveau des types, tout se ramène à des fonctions surchargées avec un nombre* fini *de branches. Cette simplification provient du fait que l'on ne considère que des intersections finies.*

4.8.2 Construction de modèles équivalents

Considérons un certain opérateur sur les types $F : \widehat{T} \to \widehat{T}$ dont on veut montrer la monotonie ($t \leq s \Rightarrow F(t) \leq F(s)$), ou d'autres propriétés, comme

par exemple $F(t_1 \vee t_2) \simeq F(t_1) \vee F(t_2)$. Dans la mesure où la relation $\leq$ n'est pas axiomatisée et où les « règles » qui la définissent sont particulièrement complexes, l'approche la plus naturelle consiste à trouver une contre-partie ensembliste à l'opérateur F. En effet, si l'on peut écrire $[\![F(t)]\!] = \widehat{F}([\![t]\!])$ pour un certain opérateur $\widehat{F} : \mathcal{P}(D) \to \mathcal{P}(D)$, on se ramène à prouver la monotonie de $\widehat{F}$ par rapport à l'inclusion ensembliste, ce qui est souvent plus facile. Par exemple, si $\widehat{F}(X)$ est défini comme $\{y \mid xRy\}$ pour une certaine relation binaire R, la propriété est triviale.

Cependant, nous ne savons rien de la nature des éléments du modèle D, et il n'est pas donc pas facile de définir un tel opérateur $\widehat{F}$. On voudrait disposer de modèles qui induisent la même relation de sous-typage, mais dont on contrôle la nature des éléments, au moins superficiellement. Nous allons voir comment procéder.

D'après le Corollaire 4.15, si $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ est un modèle, alors $\mathbb{E}[\![\_]\!] : \widehat{T} \to \mathcal{P}(\mathbb{E}D)$ en est autre, et il est équivalent à $[\![\_]\!]$. Le théorème suivant donne une généralisation de ce résultat.

Theorème 4.40 *Soient* $[\![\_]\!]_1 : \widehat{T} \to \mathcal{P}(D_1)$ *et* $[\![\_]\!]_2 : \widehat{T} \to \mathcal{P}(D_2)$ *deux modèles équivalents à* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$. *Définissons l'unique interprétation ensembliste* $[\![\_]\!]' : \widehat{T} \to \mathcal{P}(D')$ *avec*

$$D' = \mathcal{C} + D_1 \times D_2 + \mathcal{P}(D \times D_\Omega)$$

et :

$$\begin{array}{lcll} [\![t_1 \times t_2]\!]' & = & [\![t_1]\!]_1 \times [\![t_2]\!]_2 & \\ [\![a]\!]' & = & \mathbb{E}[\![a]\!] & \text{pour } a \in T_{\mathbf{basic}} \cup T_{\mathbf{fun}} \end{array}$$

Alors $[\![\_]\!]'$ *est un modèle équivalent à* $[\![\_]\!]$.

La preuve est facile ; elle consiste simplement à reprendre la preuve du Théorème 4.13. Cette construction permet de mener des raisonnements « comme si » l'on avait $[\![\_]\!] = \mathbb{E}[\![\_]\!]$. Nous allons montrer un exemple de ce genre de raisonnement, qui nous sera utile par la suite.

Typage de l'application fonctionnelle Dans le chapitre suivant, nous allons introduire un λ-calcul avec couples. L'application fonctionnelle sera en fait codée par un opérateur **app** qui prend en argument un couple (f, x) et applique la fonction f à l'argument x. Pour typer cet opérateur, nous avons besoin de définir un jugement $\mathbf{app} : t \mathbf{\to} s$, qui exprime, intuitivement, le fait que si l'opérateur **app** est appliqué à une valeur de type t, alors le résultat est nécessairement une valeur de type s.

Commençons par une approche syntaxique assez naturelle. Elle consiste à axiomatiser la relation $\mathbf{app} : \_ \to \_$ par le système inductif de la Figure 4.1.

Ces règles sont « sûres », pour une interprétation intuitive des types flèche dans un langage strict. La règle (**app→**) correspond à une règle de typage classique dans un λ-calcul. Les autres sont génériques : elles sont sémantiquement correctes pour tout opérateur. Par exemple, nous pouvons justifier la règle (**app∨**) en disant que si la valeur à laquelle on applique l'opérateur est de type $t_1 \vee t_2$, alors elle est de type t_i pour un certain i. L'hypothèse correspondante donne alors que le résultat est de type s_i, et donc *a fortiori* de type $\leq s_1 \vee s_2$.

$$\frac{}{\mathbf{app} : ((t\to s)\times t)\to s}\ (\mathbf{app}\to)$$

$$\frac{\mathbf{app} : t_1\to s_1 \quad \mathbf{app} : t_2\to s_2}{\mathbf{app} : (t_1\wedge t_2)\to(s_1\wedge s_2)}\ (\mathbf{app}\wedge)$$

$$\frac{\mathbf{app} : t_1\to s_1 \quad \mathbf{app} : t_2\to s_2}{\mathbf{app} : (t_1\vee t_2)\to(s_1\vee s_2)}\ (\mathbf{app}\vee)$$

$$\frac{\mathbf{app} : t'\to s' \quad t\leq t' \quad s'\leq s}{\mathbf{app} : t\to s}\ (\mathbf{app}\leq)$$

FIG. 4.1 – Axiomatisation du typage de l'application fonctionnelle

Il serait néanmoins agréable de disposer d'une interprétation sémantique de l'application pour donner un sens formel à cette affirmation de sûreté. Cela permettra également de considérer la question de la complétude du système de règles. De plus, nous aurons besoin, pour un type t donné, de calculer le plus petit type s tel que $\mathbf{app} : t\to s$, et la définition axiomatique ne permet pas de le faire facilement.

Nous allons donc proposer une définition sémantique de la relation binaire $\mathbf{app} : \_ \to \_$. Pour ce faire, nous allons « mimer » l'application fonctionnelle dans un modèle. Cela ne peut pas se faire dans le modèle $[\![\_]\!]$ car nous ne connaissons rien sur la nature des éléments de D. Le modèle $\mathbb{E}[\![\_]\!]$ ne convient pas non plus : certains des éléments de $\mathbb{E}D$ sont bien des couples, mais la première composante est juste un élément de D, alors que nous voudrions avoir un élément de $\mathbb{E}D$, et plus précisément de $\mathbb{E}_{\mathbf{fun}}D$, pour pouvoir définir de manière extensionnelle l'application fonctionnelle.

Le Théorème 4.40 permet de construire un modèle $[\![\_]\!]' : \widehat{T} \to \mathcal{P}(D')$, avec :

$$D' = \mathcal{C} + (\mathbb{E}D)\times D + \mathcal{P}(D\times D_\Omega)$$

et

$$[\![t_1\times t_2]\!]' = \mathbb{E}[\![t_1]\!]\times[\![t_2]\!]$$

Ce modèle est équivalent au modèle $[\![\_]\!]$, et il est donc légitime de travailler dedans. Nous pouvons mimer dans D' le comportement attendu de l'application fonctionnelle. Nous définissons une relation binaire $\triangleright$ entre D' et D_Ω par : $d\triangleright\Omega$ si d n'est pas dans $\mathcal{P}(D\times D_\Omega)\times D$, et $(f,x)\triangleright y$ si $f\in\mathcal{P}(D\times D_\Omega)$ et $(x,y)\in f$. Pour un sous-ensemble X de D', notons $\mathbf{app}(X) = \{y \mid x\triangleright y, x\in X\}\subseteq D_\Omega$.

Lemme 4.41 *Soient t_1,t_2 et s trois types. Alors :*

$$\mathbf{app}([\![t_1\times t_2]\!]')\subseteq[\![s]\!] \iff (t_1\leq t_2\to s)\vee(t_2\simeq\mathbb{0})$$

Preuve: Prouvons l'implication $\Rightarrow$. Supposons $\mathbf{app}([\![t_1\times t_2]\!]')\subseteq[\![s]\!]$ et $t_2\not\simeq\mathbb{0}$, et prouvons que $\mathbb{E}[\![t_1]\!]\subseteq[\![t_2]\!]\to[\![s]\!]$. Soit donc $f\in\mathbb{E}[\![t_1]\!]$. On voit tout d'abord que $f\in\mathcal{P}(D\times D_\Omega)$. En effet, pour tout élément $x\in[\![t_2]\!]$, on a $(f,x)\in\mathbb{E}[\![t_1]\!]\times[\![t_2]\!] = [\![t_1\times t_2]\!]'$. Si f n'était pas dans $\mathcal{P}(D\times D_\Omega)$, on aurait $(f,x)\triangleright\Omega$, et donc $\Omega\in\mathbf{app}([\![t_1\times t_2]\!]')$,

et ce qui est contradictoire avec l'hypothèse. Montrons maintenant que $f \in [\![t_2]\!] \to [\![s]\!]$. Prenons donc $x \in [\![t_2]\!]$. Il s'agit de voir que si $(x,y) \in f$, alors $y \in [\![s]\!]$. Mais si $(x,y) \in f$, alors $(f,x) \triangleright y$, et donc $y \in \mathbf{app}([\![t_1 \times t_2]\!]') \subseteq [\![s]\!]$.

Prouvons l'implication $\Leftarrow$. Le cas $t_2 \simeq \mathbb{0}$ est trivial, car alors $t_1 \times t_2 \simeq \mathbb{0}$, et donc $\mathbf{app}([\![t_1 \times t_2]\!]') = \mathbf{app}(\emptyset) = \emptyset$. Supposons que $t_1 \leq t_2 \to s$. Prenons un élément dans $[\![t_1 \times t_2]\!]' = \mathbb{E}[\![t_1]\!] \times [\![t_2]\!]$. C'est donc un couple (f,x), avec $f \in \mathbb{E}[\![t_1]\!]$ et $x \in [\![t_2]\!]$. Puisque $t_1 \leq t_2 \to s$, on a $f \in [\![t_2]\!] \to [\![s]\!]$. Si $(f,x) \triangleright y$, on a $(x,y) \in f$, ce qui implique bien $y \in [\![s]\!]$. □

Theorème 4.42 *Soient t et s deux types. Les assertions suivantes sont équivalentes :*

(i) $\mathbf{app} : t \to s$

(ii) $\mathbf{app}([\![t]\!]') \subseteq [\![s]\!]$

(iii) $(t \leq \mathbb{1} \times \mathbb{1}) \wedge \forall (t_1,t_2) \in \pi(t).\ t_1 \leq t_2 \to s$

Preuve: L'implication *(i)* $\Rightarrow$ *(ii)* se démontre par induction sur la dérivation du jugement $\mathbf{app} : t \to s$. Les règles ($\mathbf{app} \leq$), ($\mathbf{app} \vee$), ($\mathbf{app} \wedge$) se traitent avec des raisonnements ensemblistes élémentaires. Le cas de la règle ($\mathbf{app} \to$) est une conséquence du lemme précédent.

Prouvons l'implication *(ii)* $\Rightarrow$ *(iii)*. Puisque $\Omega \notin \mathbf{app}([\![t]\!]')$, on a $[\![t]\!]' \subseteq D \times D$, et donc $t \leq \mathbb{1} \times \mathbb{1}$. Pour $(t_1,t_2) \in \pi(t)$, on a $t_1 \times t_2 \leq t$, et donc $\mathbf{app}([\![t_1 \times t_2]\!]') \subseteq [\![s]\!]$, et le lemme précédent donne bien $t_1 \leq t_2 \to s$ (car $t_2 \not\simeq \mathbb{0}$).

Prouvons enfin l'implication *(iii)* $\Rightarrow$ *(i)*. On a :

$$t \simeq \bigvee_{(t_1,t_2) \in \pi(t)} t_1 \times t_2$$

Si $(t_1,t_2) \in \pi(t)$, on a $t_1 \times t_2 \leq (t_2 \to s) \times t_2$. La règle ($\mathbf{app} \to$) donne $\mathbf{app} : (t_2 \to s) \times t_2 \to s$. En appliquant la règle ($\mathbf{app} \leq$), on obtient $\mathbf{app} : t_1 \times t_2 \to s$. Les règles ($\mathbf{app} \vee$) et ($\mathbf{app} \leq$) donnent finalement $\mathbf{app} : t \to s$. Nous avons utilisé une généralisation de la règle ($\mathbf{app} \vee$) pour une réunion finie arbitraire. Le cas d'une réunion non vide s'obtient immédiatement. Pour une réunion vide, on constate que $\mathbf{app} : \mathbb{0} \to \mathbb{0}$ grâce à la règle ($\mathbf{app} \to$) (qui donne $\mathbf{app} : (\mathbb{0} \to \mathbb{0}) \times \mathbb{0} \to \mathbb{0}$), puis à la règle ($\mathbf{app} \leq$). □

Le théorème permet de mieux appréhender la relation binaire $\mathbf{app} : \_ \to \_$ entre types. La propriété *(ii)* donne le point de vue sémantique (application extensionnelle). La propriété *(iii)* donne directement un algorithme pour calculer la relation (c'est-à-dire, pour décider si $\mathbf{app} : t \to s$, les types t et s étant donnés). Mieux : en utilisant le Lemme 4.37, on obtient un algorithme pour calculer un plus petit type s tel que $\mathbf{app} : t \to s$, pour t donné, lorsqu'un tel type existe.

Remarque 4.43 *Dans la preuve de (iii) $\Rightarrow$ (i), nous n'avons pas eu besoin de la règle ($\mathbf{app} \wedge$). Cela prouve que cette règle est admissible dans le système constitué des trois autres règles ($\mathbf{app} \to$), ($\mathbf{app} \vee$), ($\mathbf{app} \leq$). Une preuve directe de ce fait, sans passer par l'intermédiaire sémantique, est facile, quoique fastidieuse.*

Remarque 4.44 *La preuve* $(i) \Rightarrow (iii)$ *passe par l'intermédiaire du point de vue sémantique. Le lecteur pourra se convaincre qu'une preuve directe (sans introduire le modèle* $[\![\_]\!]'$*) est difficile à mener.*

Corollaire 4.45 *Soient* t_1*,* t_2 *et* s *trois types. Alors :*

$$\mathbf{app} : t_1 \times t_2 \to s \iff (t_1 \leq t_2 \to s) \vee (t_2 \simeq \mathbb{0})$$

Lemme 4.46 *Soit* $t_f = \bigwedge_{i\in I} t_i \to s_i$ *où* I *est un ensemble non vide, et* t_x *un type tel que* $t_x \leq \bigvee_{i\in I} t_i$*. Posons :*

$$s = \bigvee_{I' \subseteq I \ \mid \ (*)} \bigwedge_{i \in I \setminus I'} s_i$$

où $(*)$ *est la condition* $t_x \not\leq \bigvee_{i\in I'} t_i$*. Alors on a :*

$$\mathbf{app} : t_f \times t_x \to s$$

Preuve: Le Corollaire 4.10 permet de décomposer la relation $t_f \leq t_x \to s$ en :

$$\forall I' \subseteq I.\ \left(t_x \leq \bigvee_{i\in I'} t_i\right) \vee \left\{ \begin{array}{l} I \neq I' \\ \displaystyle\bigwedge_{i\in I\setminus I'} s_i \leq s \end{array} \right.$$

et l'on constate que cette propriété est vraie, avec la définition de s donnée dans l'énoncé. On conclut avec le Corollaire 4.45.

À titre d'illustration, donnons une preuve qui utilise l'interprétation sémantique de la relation **app**. Il s'agit de voir que $\mathbf{app}([\![t_f \times t_x]\!]') \subseteq [\![s]\!]$. Prenons un élément dans $[\![t_f \times t_x]\!]'$. C'est un couple (f, x), avec $f \in \mathbb{E}[\![t_f]\!]$ et $x \in [\![t_x]\!]$. Puisque $I \neq \emptyset$, on a $\mathbb{E}[\![t_f]\!] \subseteq \mathbb{E}_{\mathbf{fun}} D$, et donc f est un ensemble de couples. Il reste à montrer que si y est tel que $(x, y) \in f$, alors $y \in [\![s]\!]$. Pour un tel y, posons $I' = \{i \mid x \notin [\![t_i]\!]\}$. Puisque $t_f \leq t_i \to s_i$, on obtient : $x \in [\![t_i]\!] \Rightarrow y \in [\![s_i]\!]$, et donc $y \in [\![s']\!]$ où $s' = \bigwedge_{i\in I\setminus I'} s_i$. Or $t_x \not\leq \bigvee_{i\in I'} t_i$ (car x est dans l'interprétation du membre gauche et pas dans celle du membre droit), ce qui donne $s' \leq s$. □

4.9 Systèmes d'équations ∨×

Dans cette section, nous considérons un modèle *structurel* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$. Nous allons étudier une forme de systèmes d'équations sur les types. Cet outil nous permettra de définir l'algorithme de typage du filtrage (Théorème 6.12).

Nous considérons des systèmes d'équations de la forme :

$$\left\{ \begin{array}{lcl} \alpha_1 & = & \ldots \\ \ldots & & \\ \alpha_n & = & \ldots \end{array} \right.$$

où les membres droit sont eux-mêmes des variables α_i, des types t, des produits de variables $\alpha_i \times \alpha_j$, ou une réunion finie de termes de cette forme. Ainsi, en notant V l'ensemble des α_i, on peut voir chaque membre droit comme un ensemble fini d'éléments de l'ensemble $V + V \times V + \widehat{T}$.

Definition 4.47 *Un* **système d'équations ∨×** *est un couple* (V, ϕ) *où* V *est un ensemble fini et* ϕ *est une fonction* $V \to \mathcal{P}_f(V + V \times V + \widehat{T})$*. On dit que le système est* **bien fondé** *si* $\phi(\alpha) \subseteq V \times V + \widehat{T}$ *pour tout* $\alpha \in V$*.*

Pour définir une notion de solution pour un tel système, nous allons lui associer un transformateur ensembliste et dire qu'une solution est un point fixe de ce transformateur.

Definition 4.48 *À un système* (V, ϕ)*, on associe l'opérateur* $\widehat{\phi} : \mathcal{P}(D)^V \to \mathcal{P}(D)^V$ *défini par :*

$$\widehat{\phi}(\rho)(\alpha) = \bigcup_{\beta \in \phi(\alpha)} \rho(\beta) \cup \bigcup_{(\beta_1,\beta_2) \in \phi(\alpha)} \rho(\beta_1) \times \rho(\beta_2) \cup \bigcup_{t \in \phi(\alpha)} [\![t]\!]$$

Cet opérateur est croissant pour l'ordre sur $\mathcal{P}(D)^V$ *défini par* $\rho_1 \leq \rho_2 \iff \forall \alpha \in V.\ \rho_1(\alpha) \subseteq \rho_2(\alpha)$*. Son plus petit point fixe est noté* $lfp(\widehat{\phi})$*.*

Nous allons maintenant voir comment calculer le plus petit point fixe, donc l'existence est garantie par le théorème de Kleene.

Theorème 4.49 *Soit* (V, ϕ) *est système d'équations* ∨×*. Alors on peut construire une famille de types* $(t_\alpha)_{\alpha \in V}$ *telle que :*

$$\forall \alpha \in V.\ lfp(\widehat{\phi})(\alpha) = [\![t_\alpha]\!]$$

De plus, cette famille ne dépend pas du modèle (structurel) choisi.

Le théorème est une conséquence des deux lemmes suivant.

Lemme 4.50 *Pour tout système* (V, ϕ)*, on peut calculer un système bien fondé* (V, ϕ') *tel que* $lfp(\widehat{\phi}) = lfp(\widehat{\phi'})$*.*

Preuve: Notons $\overset{*}{\rightsquigarrow}$ la clôture transitive et reflexive de la relation définie par $\alpha \rightsquigarrow \beta \iff \beta \in \phi(\alpha)$. On définit le système bien fondé (V, ϕ') par :

$$\phi'(\alpha) = \bigcup_{\alpha \overset{*}{\rightsquigarrow} \beta} \phi(\beta) \backslash V$$

Notons $\rho_1 = \mathrm{lfp}(\widehat{\phi})$ et $\rho_2 = \mathrm{lfp}(\widehat{\phi'})$.

On constate tout d'abord que pour tout ρ :

$$\widehat{\phi'}(\rho)(\alpha) \subseteq \bigcup_{\alpha \overset{*}{\rightsquigarrow} \beta} \widehat{\phi}(\rho)(\beta)$$

De plus, si $\alpha \rightsquigarrow \beta$, alors $\rho(\beta) \subseteq \widehat{\phi}(\rho)(\alpha)$, et donc $\rho_1(\beta) \subseteq \rho_1(\alpha)$, et cela se généralise au cas $\alpha \overset{*}{\rightsquigarrow} \beta$. On obtient ainsi $\widehat{\phi'}(\rho_1)(\alpha) \subseteq \rho_1(\alpha)$, ce qui donne $\rho_2 \leq \rho_1$.

Prouvons maintenant $\rho_1 \leq \rho_2$. Pour tout ρ, on a :

$$\widehat{\phi}(\rho)(\alpha) \subseteq \widehat{\phi'}(\rho)(\alpha) \cup \bigcup_{\alpha \rightsquigarrow \beta} \rho(\beta)$$

En donc en particulier :

$$\widehat{\phi}(\rho_2)(\alpha) \subseteq \rho_2(\alpha) \cup \bigcup_{\alpha \rightsquigarrow \beta} \rho_2(\beta)$$

Mais si $\alpha \rightsquigarrow \beta$, alors $\phi'(\beta) \subseteq \phi'(\alpha)$, et donc $\rho_2(\beta) = \widehat{\phi'}(\rho_2)(\beta) \subseteq \widehat{\phi'}(\rho_2)(\alpha) = \rho_2(\alpha)$. On obtient ainsi $\widehat{\phi}(\rho_2)(\alpha) \subseteq \rho_2(\alpha)$, ce qui donne bien $\rho_1 \leq \rho_2$. □

Lemme 4.51 *Soit (V, ϕ) un système bien fondé. Alors l'opérateur $\widehat{\phi}$ a un unique point fixe ρ, et on peut calculer une famille de types $(t_\alpha)_{\alpha \in V}$, indépendante du modèle, telle que :*

$$\forall \alpha \in V.\ \rho(\alpha) = [\![t_\alpha]\!]$$

Preuve: Commençons par prouver l'unicité du point fixe. Soient ρ_1 et ρ_2 deux points fixes de $\widehat{\phi}$, et prouvons que $\rho_1 = \rho_2$. Pour cela, il suffit d'établir la propriété ci-dessous pour tout $d \in D$:

$$\forall \alpha \in V.\ d \in \rho_1(\alpha) \iff \rho_2(\alpha)$$

On procède par induction sur d, en remarquant que l'assertion $d \in \rho_i(\alpha)$ s'écrit (puisque $\rho_i = \widehat{\phi}(\rho_i)$) :

$$\begin{cases} \exists t \in \phi(\alpha).\ d \in [\![t]\!] \\ \vee \\ \exists (\beta_1, \beta_2) \in \phi(\alpha), d_1 \in \rho_i(\beta_1), d_2 \in \rho_i(\beta_2).\ d = (d_1, d_2) \end{cases}$$

Passons au deuxième point de l'énoncé. La propriété de régularité de la $\mathcal{B} \circ F$-coalgèbre $\mathbb{T}$ permet de construire une famille de types $(t_\alpha)_{\alpha \in V}$ telle que :

$$\forall \alpha \in V.\ t_\alpha = \bigvee_{(\beta_1, \beta_2) \in \phi(\alpha)} t_{\beta_1} \times t_{\beta_2} \vee \bigvee_{t \in \phi(\alpha)} t$$

On voit immédiatement que si l'on pose $\rho(\alpha) = [\![t_\alpha]\!]$, alors ρ est un point fixe de $\widehat{\phi}$. □

Chapitre 5

Calcul

Dans ce chapitre, nous introduisons un mini-langage (ou calcul) typé. Nous utilisons l'algèbre de types introduite au Chapitre 3.

5.1 Syntaxe

Les expressions sont définies par la grammaire de la Figure 5.1. Nous notons $\mathcal{E}$ l'ensemble des expressions.

e	$::=$	c	$c \in \mathcal{C}$	constante
	$\mid$	(e_1, e_2)		couple
	$\mid$	$\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$	$t_i, s_i \in \widehat{T},\ n \geq 1$	abstraction
	$\mid$	x		variable
	$\mid$	$o(e)$	$o \in \mathcal{O}$	opérateur
	$\mid$	$(x = e \in t\ ?\ e_1 \mid e_2)$	$t \in \widehat{T}$	test de type

FIG. 5.1 – Expressions (termes) du calcul

Les lettres x et f désignent des *variables*. Nous supposons qu'il existe une infinité de variables distinctes, et, de manière classique, nous considérons les expressions modulo alpha-conversion. Les problèmes de collision sont évités par renommage implicite. L'ensemble des variables libres d'une expression e est noté $\mathrm{fv}(e)$, et il est défini de la manière suivante :

$$
\begin{array}{lcl}
\mathrm{fv}(c) & = & \emptyset \\
\mathrm{fv}((e_1, e_2)) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\mu f(\ldots).\lambda x.e) & = & \mathrm{fv}(e) \backslash \{f, x\} \\
\mathrm{fv}(x) & = & x \\
\mathrm{fv}(o(e)) & = & \mathrm{fv}(e) \\
\mathrm{fv}((x = e \in t\ ?\ e_1 \mid e_2)) & = & \mathrm{fv}(e) \cup ((\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)) \backslash \{x\})
\end{array}
$$

On dit que l'expression e est close si $\mathrm{fv}(e) = \emptyset$.

Les productions pour les constantes, les couples, et les variables ne méritent pas de commentaires. La production pour les abstractions est un peu particulière : elle permet d'une part de définir des fonctions récursives (la variable f est liée à la fonction elle-même dans son corps) et d'autre part de spécifier un nombre fini de types $(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n)$ pour le *prototype* (ou l'interface) de ces fonctions. La portée des variables f et x est limitée au corps e de l'abstraction. Si P est un ensemble fini et non vide de types flèche, soit $P = \{t_1 \rightarrow s_1, \ldots, t_n \rightarrow s_n\}$, on s'autorise à noter $\mu f(P).\lambda x.e$ au lieu de $\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$ (cela demande de choisir un ordre d'énumération des éléments de P).

On désigne par $\mathcal{O}$ un ensemble d'opérateurs ; c'est un paramètre du calcul. Nous supposons qu'il existe toujours un opérateur « application fonctionnelle » $\mathbf{app} \in \mathcal{O}$. Nous noterons $e_1 e_2$ au lieu de $\mathbf{app}((e_1, e_2))$.

La dernière production permet d'effectuer un test de type dynamique. Si le résultat de l'expression e est de type t, alors le calcul continue avec l'expression e_1, sinon avec l'expression e_2. La variable x est liée dans e_1 et dans e_2 au résultat de e.

Ces tests de type font que la sémantique du langage est dirigée par les types. Nous devons donc introduire un système de types avant de considérer la sémantique du langage.

5.2 Système de types

Le système de types est présenté de manière classique par un système d'induction (Figure 5.2). Le jugement de typage est de la forme $\Gamma \vdash e : t$ où Γ désigne un environnement de typage, c'est-à-dire une fonction partielle à domaine fini des variables vers les types.

Nous avons besoin d'une relation de sous-typage. Dans le chapitre précédent, nous avons vu comment en définir à partir d'un modèle quelconque. Nous prenons un modèle *bien fondé* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$, et nous notons simplement $\leq$, au lieu de $\leq_{[\![\_]\!]}$ la relation de sous-typage qu'il induit. Nous supposons que cette relation de sous-typage est décidable (c'est le cas pour les modèles universels lorsque le sous-typage est décidable pour les types de base). Nous notons $\simeq$ l'équivalence induite par ce sous-typage : $t_1 \simeq t_2 \iff (t_1 \leq t_2) \wedge (t_2 \leq t_1)$

Pour chaque opérateur $o \in \mathcal{O}$, on suppose donnée une relation binaire entre les types, notée $o : \_ \rightarrow \_$. On suppose que si l'on a $o : t_1 \rightarrow s_1$ et $o : t_2 \rightarrow s_2$, alors on a aussi $o : (t_1 \wedge t_2) \rightarrow (s_1 \wedge s_2)$. Cette condition permettra d'obtenir le Lemme 5.3, qui permet à son tour de voir que l'ensemble des types affectés à une expression est un filtre (ensemble de types clos par subsomption et par intersection). Il est toujours possible de compléter $o : \_ \rightarrow \_$ pour satisfaire cette condition. Pour l'opérateur **app**, on prend la relation binaire $\mathbf{app} : \_ \rightarrow \_$ définie au Chapitre 4.

Outre les règles (*abstr*) et (*case*), le système (Figure 5.2) est assez classique. Commentons donc ces deux règles.

La règle (*abstr*) permet de donner un type aux abstractions. Dans le calcul, les fonctions sont surchargées : leur prototype peut spécifier plusieurs types flèche. Le type donné à l'abstraction correspond à l'intersection de tous ces types flèche. C'est le cas $m = 0$. On vérifie que chacun de ces types flèche est valide : pour cela, il faut vérifier n contraintes sur le type du corps avec des hypothèses différentes sur le type de l'argument.

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}\ (subsum)$$

$$\frac{}{\Gamma \vdash c : b_c}\ (const)$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

$$\frac{\begin{array}{l} t = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j) \\ \forall i = 1..n. (f : t), (x : t_i), \Gamma \vdash e : s_i \\ t \not\simeq \mathbb{0} \end{array}}{\Gamma \vdash \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e : t}\ (abstr)$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Gamma \vdash e : t \quad o : t \to s}{\Gamma \vdash o(e) : s}\ (op)$$

$$\frac{\Gamma \vdash e : t_0 \quad \left\{ \begin{array}{l} t_1 = t_0 \wedge t \\ t_2 = t_0 \setminus t \end{array} \right. \quad \forall i = 1..2. \left\{ \begin{array}{ll} s_i = \mathbb{0} & \text{si } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{si } t_i \not\simeq \mathbb{0} \end{array} \right.}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1 \vee s_2}\ (case)$$

FIG. 5.2 – Système de type

Mais la règle permet aussi d'ajouter à cette intersection un nombre fini arbitraire de négations de types flèche ($m > 0$), pourvu que l'intersection ne devienne pas vide. La raison pour autoriser ces types négatifs est assez simple : nous voulons pouvoir munir l'ensemble des valeurs du langage d'une structure de modèle. Cela se fera en interprétant un type comme l'ensemble des valeurs de ce type. Comme une abstraction close et bien typée est une valeur, elle doit avoir un des deux types $t \to s$ ou $\neg(t \to s)$ pour tout choix de t et s ; en effet, dans un modèle, tout élément doit être soit dans l'interprétation d'un type, soit dans celle de son complémentaire. Cela explique que l'on autorise à ajouter des négations de types flèche dans la règle de typage, sinon on ne pourrait jamais déduire pour les abstractions des négations de type flèche. Le lemme suivant permet d'écrire la condition $t \not\simeq \mathbb{0}$ différemment.

Lemme 5.1 *Soit* $t = \bigwedge_{i=1..n}(t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$ *un type. Alors*

$$t \simeq \mathbb{0} \iff \exists j = 1..m. \bigwedge_{i=1..n} (t_i \to s_i) \leq t'_j \to s'_j$$

| *Preuve*: Conséquence du Lemme 4.9. □

Autrement dit, la condition $t \not\simeq \mathbb{0}$ signifie simplement qu'aucun des types flèche ajoutés (en négatif) ne pourrait être obtenu (en positif) grâce à la règle de subsomption à partir du type $\bigwedge_{i=1..n}(t_i \to s_i)$.

La règle (*case*) est plus simple à décrire. L'idée est de raffiner le type t_0. Si la première branche est sélectionnée, cela signifie que l'expression e s'est évaluée en une valeur de type t ; on sait donc que x aura le type t et le type t_0, et donc le type $t \wedge t_0$. De même, si la deuxième branche est sélectionnée, la valeur n'est pas de type t, et donc de type $\neg t$ car l'ensemble des valeurs doit être un modèle. Il faut tenir compte des deux cas possibles pour calculer le type du résultat. En général, c'est la réunion du type des résultats de chaque branche. Mais on peut raffiner : en effet, si l'on est sûr qu'une branche ne sera pas utilisée, il n'est pas nécessaire de la prendre en compte dans cette réunion. Ce raffinement est nécessaire typiquement lorsque l'on type le corps d'une fonction surchargée (plusieurs types flèche dans son prototype). Il faut alors vérifier plusieurs fois le type du corps, avec des hypothèses différentes, et dans certains cas, il peut être nécessaire de ne pas prendre en compte une certaine branche pour pouvoir vérifier la contrainte. On peut le voir sur l'exemple suivant :

$$\mu f(\texttt{int}\to\texttt{int};\texttt{bool}\to\texttt{bool}).\lambda x.(y = x \in \texttt{int}\ ?\ 1 \mid \texttt{true})$$

Dans cet exemple, nous utilisons deux types de base disjoints `int` et `bool`, et deux constantes 1 (de type `int`) et `true` (de type `bool`). Lorsque l'on doit vérifier la contrainte donnée par $\texttt{int}\to\texttt{int}$, il faut bien ignorer la deuxième branche, sans quoi le type du corps serait $\texttt{int} \vee \texttt{bool}$ qui n'est pas un sous-type de `int`.

5.3 Propriétés syntaxiques du typage

Lemme 5.2 (Renforcement des hypothèses) *Soient Γ_1 et Γ_2 deux environnements de typage tel que pour tout x dans le domaine de Γ_1, on a $\Gamma_2(x) \leq \Gamma_1(x)$. Si $\Gamma_1 \vdash e : t$, alors $\Gamma_2 \vdash e : t$.*

| *Preuve*: Induction sur la dérivation de $\Gamma_1 \vdash e : t$. □

Lemme 5.3 *Si $\Gamma \vdash e : t_1$ et $\Gamma \vdash e : t_2$, alors $\Gamma \vdash e : t_1 \wedge t_2$.*

| *Preuve*: Par induction sur la structure des deux dérivations de typage pour $\Gamma \vdash e : t_1$ et $\Gamma \vdash e : t_2$.
|
| Considérons d'abord le cas où la dernière règle appliquée dans l'une des deux dérivations est (*subsum*), disons :
|
| $$\dfrac{\dfrac{\ldots}{\Gamma \vdash e : s_1} \quad s_1 \leq t_1}{\Gamma \vdash e : t_1} \quad \dfrac{\ldots}{\Gamma \vdash e : t_2}$$
|
| L'hypothèse d'induction permet d'obtenir $\Gamma \vdash e : s_1 \wedge t_2$. On conclut ce cas en constatant $s_1 \wedge t_2 \leq t_1 \wedge t_2$ puisque $s_1 \leq t_1$.
|
| Si aucun des deux dernières règles appliquées n'est (*subsum*), alors il s'agit de deux instances de la même règle.

Règles (*const*), (*var*) : ces règles ne permettent d'obtenir qu'un seul type t pour l'expression, et $t \wedge t \simeq t$.
Règle (*op*) : conséquence directe de la propriété de clôture de o : $\_ \rightarrow \_$ par intersection.
Règle (*pair*) : considérons la situation ci-dessous.

$$\frac{\frac{\ldots}{\Gamma \vdash e_1 : t_1} \quad \frac{\ldots}{\Gamma \vdash e_2 : t_2}}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \frac{\frac{\ldots}{\Gamma \vdash e_1 : t'_1} \quad \frac{\ldots}{\Gamma \vdash e_2 : t'_2}}{\Gamma \vdash (e_1, e_2) : t'_1 \times t'_2}$$

Soit $t''_1 = t_1 \wedge t'_1$ et $t''_2 = t_2 \wedge t'_2$. En appliquant deux fois l'hypothèse d'induction, on obtient $\Gamma \vdash e_1 : t''_1$ et $\Gamma \vdash e_2 : t''_2$. La règle (*pair*) donne alors $\Gamma \vdash (e_1, e_2) : t''_1 \times t''_2$. Pour conclure, il suffit de constater que $t''_1 \times t''_2 \simeq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$ d'après le Lemme 4.34.
Règle (*case*) : considérons la situation ci-dessous.

$$\frac{\frac{\ldots}{\Gamma \vdash e : t_0} \quad \frac{\ldots}{(x : t_i), \Gamma \vdash e_i : s_i}}{\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s_1 \vee s_2} \qquad \frac{\frac{\ldots}{\Gamma \vdash e : t'_0} \quad \frac{\ldots}{(x : t'_i), \Gamma \vdash e_i : s'_i}}{\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s'_1 \vee s'_2}$$

avec $t_1 = t_0 \wedge t$, $t_2 = t_0 \setminus t$, $t'_1 = t'_0 \wedge t$, $t'_2 = t'_0 \setminus t$. L'hypothèse d'induction donne tout d'abord : $\Gamma \vdash e : t''_0$ avec $t''_0 = t_0 \wedge t'_0$. On pose $t''_1 = t''_0 \wedge t$ et $t''_2 = t''_0 \setminus t$. Soit $i = 1..2$. On constate que $t''_i \leq t_i$, et donc le Lemme 5.2 donne $(x : t''_i), \Gamma \vdash e_i : s_i$. De même, l'on obtient $(x : t''_i), \Gamma \vdash e_i : s'_i$, et donc, en appliquant l'hypothèse d'induction $(x : t''_i), \Gamma \vdash e_i : s''_i$ où $s''_i = s_i \wedge s'_i$. Avec la règle (*case*), on prouve alors $\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s''_1 \vee s''_2$, et on conclut en constatant que $s''_1 \vee s''_2 \leq (s_1 \vee s_2) \wedge (s'_1 \vee s'_2)$.

Les cas particuliers (lorsque $t_i \simeq \mathbb{0}$ ou $t'_i \simeq \mathbb{0}$) se traitent sans problème.
Règle (*abstr*) : considérons deux applications de la règle (*abstr*) à la même abstraction $\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$, avec les types ci-dessous :

$$\begin{aligned} t &= \bigwedge_{i=1..n} (t_i \rightarrow s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \rightarrow s'_j) \\ t' &= \bigwedge_{i=1..n} (t_i \rightarrow s_i) \wedge \bigwedge_{j=m+1..m'} \neg(t'_j \rightarrow s'_j) \end{aligned}$$

où $t \not\simeq \mathbb{0}$ et $t' \not\simeq \mathbb{0}$. Posons :

$$t'' = \bigwedge_{i=1..n} (t_i \rightarrow s_i) \wedge \bigwedge_{j=1..m'} \neg(t'_j \rightarrow s'_j)$$

On a $t'' \simeq t \wedge t'$. Vérifions que la règle (*abstr*) permet de donner le type t'' à l'abstraction considérée. Pour $i = 1..n$, on a par hypothèse $(f : t), (x : t_i), \Gamma \vdash e : s_i$, et donc, par le Lemme 5.2 : $(f : t''), (x : t_i), \Gamma \vdash e : s_i$. Il ne reste donc plus qu'à vérifier que $t'' \not\simeq \mathbb{0}$, ce qui résulte immédiatement du Lemme 5.1. Notons que dans ce cas, nous n'avons pas utilisé l'hypothèse d'induction. □

Corollaire 5.4 *Soit* Γ *un environnement et* e *une expression bien typée sous*

Γ. *Alors l'ensemble* $\{t \in \widehat{T} \mid (\Gamma \vdash e : t) \vee (\Gamma \vdash e : \neg t)\}$ *contient* $\mathbb{0}$ *et est stable par les connecteurs booléens.*

> *Preuve*: Soit E l'ensemble introduit dans l'énoncé. Il est clairement stable par l'opérateur $\neg$ et invariant par l'équivalence $\simeq$. On a $\Gamma \vdash e : \mathbb{1}$ par la règle (*subsum*), donc $\mathbb{1} \in E$. On obtient $\mathbb{0} \in E$ en écrivant $\mathbb{0} \simeq \neg\mathbb{1}$. Il ne reste plus qu'à montrer que E est stable par $\vee$, car il sera alors aussi stable par $\wedge$ compte-tenu de $t_1 \wedge t_2 \simeq \neg((\neg t_1) \vee (\neg t_2))$. Prenons t_1 et t_2 dans E, et montrons que $t_1 \vee t_2$ y est aussi. Supposons $\Gamma \not\vdash e : t_1 \vee t_2$. Alors, par la règle (*subsum*), on obtient $\Gamma \not\vdash e : t_1$ et $\Gamma \not\vdash e : t_2$. On a donc $\Gamma \vdash e : \neg t_1$ et $\Gamma \vdash e : \neg t_2$. Le Lemme 5.3 donne $\Gamma \vdash e : \neg t_1 \wedge \neg t_2$. Or $\neg t_1 \wedge \neg t_2 \simeq \neg(t_1 \vee t_2)$. □

Lemme 5.5 (Substitution) *Soient* $e, e_1, \ldots, e_n$ *des expressions,* $x_1, \ldots, x_n$ *des variables distinctes,* $t, t_1, \ldots, t_n$ *des types, et* Γ *un environnement de typage. On note* $e[x_1 := e_1; \ldots; x_n := e_n]$ *l'expression obtenue à partir de* e *en remplaçant (sans capture) les occurrences libres de* x_i *par* e_i*. On a alors :*

$$\begin{cases} (x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t \\ \forall i = 1..n.\ \Gamma \vdash e_i : t_i \end{cases} \Rightarrow \Gamma \vdash e[x_1 := e_1; \ldots; x_n := e_n] : t$$

> *Preuve*: Par induction sur la dérivation de typage $(x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t$. □

5.4 Valeurs

Parmi l'ensemble des expressions, on isole celles qui sont *closes*, *bien typées*, et qui sont produites par la grammaire :

$$\begin{array}{rcl} v & ::= & c \\ & | & (v_1, v_2) \\ & | & \mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e \end{array}$$

Lorsque l'on parle d'induction sur la structure d'une valeur, on considère le cas des abstractions comme un cas de base.

On note $\mathcal{V}$ l'ensemble des valeurs, et l'on pose :

$$[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash v : t\}$$

D'après la règle de subsomption, cette définition est compatible avec la sémantique des types : si $t_1 \leq t_2$, alors $[\![t_1]\!]_{\mathcal{V}} \subseteq [\![t_2]\!]_{\mathcal{V}}$, et en particulier, si $t_1 \simeq t_2$, alors $[\![t_1]\!]_{\mathcal{V}} = [\![t_2]\!]_{\mathcal{V}}$.

Theorème 5.6 *La fonction* $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \rightarrow \mathcal{P}(\mathcal{V})$ *est un modèle structurel. La relation de sous-typage qu'il induit est* $\leq$.

Ce théorème affirme que quelque soit le modèle $[\![\_]\!]$ bien fondé que nous avons choisi pour définir la relation de sous-typage $\leq$ utilisée dans le système de types, nous obtenons une nouvelle interprétation de la relation de sous-typage $t \leq s$ comme l'inclusion de l'ensemble des valeurs de type t dans l'ensemble des

valeurs de type s. Autrement dit, nous pouvons « oublier » le modèle de départ pour interpréter la relation de sous-typage.

Des choix différents pour le modèle de départ peuvent donner naissance à des relations de sous-typage différentes, et donc à des systèmes de types différents pour le même calcul. Il est intéressant de noter que, dans la mesure où la sémantique du calcul est dirigée par les types, celle-ci est également affectée par ces choix.

Corollaire 5.7 *Pour tout modèle bien fondé, on peut construire un modèle structurel qui induit la même relation de sous-typage.*

Les résultats qui suivent permettent d'établir le Théorème 5.6.

Lemme 5.8 *Aucune valeur n'a le type* $\mathbb{0}$.

> *Preuve*: On montre par une induction facile sur la dérivation de typage $\vdash v : t$ que t ne peut pas être $\mathbb{0}$. □

Lemme 5.9 *Pour tout type* $t \in \widehat{T}$, $[\![t]\!]_{\mathcal{V}} \cup [\![\neg t]\!]_{\mathcal{V}} = \mathcal{V}$.

> *Preuve*: Il s'agit de montrer que pour toute valeur v et tout type t, on a $\vdash v : t$ ou $\vdash v : \neg t$. On procède par récurrence sur la structure de v. Le Corollaire 5.4 permet de se ramener au cas où t est un atome a.
>
> Si v est une constante c, alors $\vdash c : b_c$. Si a n'est pas un type de base, alors $b_c \leq \neg a$ et donc $\vdash c : \neg a$. Si a est un type de base, alors on a soit $b_c \leq a$, soit $b_c \leq \neg a$ (car b_c est un type singleton, voir Section 4.1), ce qui permet de conclure pour ce cas.
>
> Considérons le cas où v est une abstraction de prototype $(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n)$. Notons $t_0 = \bigwedge_{i=1..n} t_i \rightarrow s_i$. On a $\vdash v : t_0$. Si a n'est pas un type flèche, alors $t_0 \leq \neg a$, et donc $\vdash v : \neg a$. Si a est un type flèche tel que $t_0 \leq a$, alors $\vdash v : a$ par la règle (*subsum*). Sinon, on peut appliquer la règle (*abstr*) avec a comme type flèche négatif, et l'on obtient $\vdash v : t_0 \wedge \neg a$, et donc $\vdash v : \neg a$.
>
> Enfin, considérons le cas $v = (v_1, v_2)$. Si a n'est pas un type produit, on obtient facilement $\vdash v : \neg a$. Supposons maintenant $a = t_1 \times t_2$. Si $\vdash v_1 : t_1$ et $\vdash v_2 : t_2$, alors la règle (*pair*) donne tout de suite $\vdash v : a$. Si ce n'est pas le cas, disons $\nvdash v_1 : t_1$, alors l'hypothèse d'induction donne $\vdash v_1 : \neg t_1$. Posons $t'_1 = \neg t_1$. On a $\vdash v_1 : t'_1$, et aussi $\vdash v_2 : \mathbb{1}$ puisque v_2 est bien typée. On obtient $\vdash v : t'_1 \times \mathbb{1} \leq \neg(t_1 \times t_2)$. □

Lemme 5.10 *La fonction* $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \to \mathcal{P}(\mathcal{V})$ *est une interprétation ensembliste.*

> *Preuve*: La condition $[\![\mathbb{0}]\!]_{\mathcal{V}} = \mathbb{0}$ est exprimée par le Lemme 5.8. On obtient $[\![\mathbb{1}]\!]_{\mathcal{V}} = \mathcal{V}$ en constatant que par définition, toute valeur est bien typée, et on peut donc lui donner le type $\mathbb{1}$ avec la règle (*subsum*).
>
> La condition $[\![t_1 \wedge t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cap [\![t_2]\!]_{\mathcal{V}}$ s'obtient avec le Lemme 5.3. Elle donne en particulier, pour tout type t : $[\![t]\!]_{\mathcal{V}} \cap$

$[\![\neg t]\!]_{\mathcal{V}} = [\![t \wedge \neg t]\!]_{\mathcal{V}} = [\![\mathbb{0}]\!]_{\mathcal{V}} = \emptyset$. En combinant cela avec le Lemme 5.9, on obtient $[\![\neg t]\!]_{\mathcal{V}} = \mathcal{V} \backslash [\![t]\!]_{\mathcal{V}}$.

Finalement, la condition pour la réunion $[\![t_1 \vee t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cup [\![t_2]\!]_{\mathcal{V}}$ s'obtient en écrivant $t_1 \vee t_2 \simeq \neg(\neg t_1 \wedge \neg t_2)$ et en utilisant les propriétés établies pour la négation et l'intersection. □

Pour conclure que $[\![\_]\!]_{\mathcal{V}}$ est un modèle qui définit la même relation de sous-typage que $[\![\_]\!]$, il suffit donc, d'après le Corollaire 4.15, d'établir le lemme ci-dessous.

Lemme 5.11 *Pour tout type t :*

$$[\![t]\!] = \emptyset \iff [\![t]\!]_{\mathcal{V}} = \emptyset$$

Preuve: L'implication $\Rightarrow$ provient du Lemme 5.8. Pour établir l'implication $\Leftarrow$, nous allons utiliser le fait que $[\![\_]\!]$ est un modèle bien fondé (Définition 4.3). Nous disposons d'un ordre bien fondé $\lhd$ sur D. Raisonnons par induction sur cet ordre pour prouver que pour tout type t, si $d \in [\![t]\!]$, alors il existe $v \in [\![t]\!]_{\mathcal{V}}$. Supposons donc $d \in [\![t]\!]$. La définition d'un modèle bien fondé donne un $d' \in \mathbb{E}[\![t]\!]$ tel que, si $d' = (d_1, d_2)$, alors $d_1 \lhd d$ et $d_2 \lhd d$. Distinguons suivant la forme de d'.

Si $d' = (d_1, d_2)$, alors $d_1 \lhd d$ et $d_2 \lhd d$. Cet élément d' est dans l'ensemble $\mathbb{E}[\![t]\!]$, et donc dans :

$$\bigcup_{(P,N)\in t \ \mid \ P\subseteq T_{\mathbf{prod}}} \left(\bigcap_{t_1 \times t_2 \in P} [\![t_1]\!] \times [\![t_2]\!] \right) \setminus \left(\bigcup_{t_1 \times t_2 \in N} [\![t_1]\!] \times [\![t_2]\!] \right)$$

En utilisant le Lemme 4.6, on trouve $(P, N) \in t$, et $N' \subseteq N$ tels que $P \subseteq T_{\mathbf{prod}}$, $d_1 \in [\![s_1]\!]$ et $d_2 \in [\![s_2]\!]$ où :

$$\begin{cases} s_1 = \bigwedge_{t_1 \times t_2 \in P} t_1 \setminus \bigvee_{t_1 \times t_2 \in N'} t_1 \\ s_2 = \bigwedge_{t_1 \times t_2 \in P} t_2 \setminus \bigvee_{t_1 \times t_2 \in N \setminus N'} t_2 \end{cases}$$

En appliquant l'hypothèse de récurrence à d_1 et d_2, on trouve deux valeurs v_1 et v_2 telles que $\vdash v_1 : s_1$ et $\vdash v_2 : s_2$. On prend $v = (v_1, v_2)$. La règle (*pair*) donne $\vdash v : s_1 \times s_2$. Il ne reste plus qu'à constater que $s_1 \times s_2 \leq t$, ce qui provient encore une fois du Lemme 4.6. On a bien $v \in [\![t]\!]_{\mathcal{V}}$.

Si d' est une constante c, alors on trouve $(P, N) \in t$ tel que $P \subseteq T_{\mathbf{basic}}$, $\forall b \in P.\ c \in \mathbb{B}[\![b]\!]$ et $\forall b \in N.\ c \notin \mathbb{B}[\![b]\!]$. En en déduit $b_c \leq \bigwedge_{b\in P} b \setminus \bigvee_{b\in N} b \leq t$, ce qui donne $\vdash c : t$.

Finalement, considérons le cas où $d' \in \mathcal{P}(D \times D_\Omega)$. On trouve $(P, N) \in t$ tel que $P \subseteq T_{\mathbf{fun}}$ et $d' \in \mathbb{E}[\![t']\!]$ où

$$t' = \bigwedge_{t_1 \to t_2 \in P} t_1 \to t_2 \wedge \bigwedge_{t_1 \to t_2 \in N} \neg(t_1 \to t_2)$$

On a $t' \leq t$ et $t' \not\simeq \mathbb{0}$ (puisque $\mathbb{E}[\![t']\!] \neq \emptyset$). On conclut en considérant une abstraction close et bien typée, dont le prototype est donné par P, par exemple $\mu f(P).\lambda x.\ f\ x$. La règle (*abstr*) donne à cette valeur le type t'. □

Pour établir la sûreté de la sémantique, nous aurons besoin du résultat ci-dessous.

Lemme 5.12 (Inversion)

$$\begin{array}{lcl} [\![t_1 \times t_2]\!]_{\mathcal{V}} & = & \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\} \\ [\![b]\!]_{\mathcal{V}} & = & \{c \mid b_c \leq b\} \\ [\![t \to s]\!]_{\mathcal{V}} & = & \{(\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e) \in \mathcal{V}. \mid \bigwedge_{i=1..n} t_i \to s_i \leq t \to s\} \end{array}$$

Preuve: Pour chacune des égalités, l'inclusion $\supseteq$ est immédiate. Montrons les trois inclusions $\subseteq$ successivement. Commençons par remarquer que si v est une valeur, alors on a $\vdash v : a$ pour un certain atome a de même genre que v : si v est une constante (resp. couple)(resp. fonction), a est un type de base (resp. type produit)(resp. type flèche). Si l'on a de plus $\vdash v : t_0$, alors on obtient par le Lemme 5.3 $\vdash v : t_0 \wedge a$. D'après le Lemme 5.8, on a donc $t_0 \wedge a \not\simeq \mathbb{0}$, et donc si t_0 est lui aussi un atome, il est du même genre que v (car deux atomes de genre différent ont une intersection vide).

Commençons par le cas $t_0 = t_1 \times t_2$. Si l'on a $\vdash v : t_1 \times t_2$, alors d'après la remarque ci-dessus, v doit être un couple : $v = (v_1, v_2)$. Le jugement $\vdash v : t_1 \times t_2$ ne peut être obtenu qu'en appliquant un certain nombre de fois la règle (*subsum*) après une instance de la règle (*pair*). Cela donne deux types t'_1 et t'_2 tels que $\vdash v_1 : t'_1$, $\vdash v_2 : t'_2$, et $t'_1 \times t'_2 \leq t_1 \times t_2$. Or $t'_1 \not\leq \mathbb{0}$ et $t'_2 \not\leq \mathbb{0}$ d'après le Lemme 5.8, et l'on obtient donc $t'_1 \leq t_1$ et $t'_2 \leq t_2$. On a bien : $v \in \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\}$.

Considérons maintenant le cas $t_0 = b$. La valeur v est donc nécessairement une constante c, et toute dérivation de typage pour c consiste en un nombre fini d'applications de la règle (*subsum*) après la règle (*const*). On obtient ainsi $b_c \leq b$.

Enfin, regardons le cas $t_0 = t \to s$. La valeur v est une abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$. On voit, avec un raisonnement comme ci-dessus, qu'il existe un nombre fini de types flèche $(t'_j \to s'_j)_{j=1..m}$ tels que $s_0 \leq t \to s$ et $s_0 \not\simeq \mathbb{0}$ où :

$$s_0 = \bigwedge_{i=1..n} t_i \to s_i \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$$

Le Lemme 5.1 donne donc :

$$\bigwedge_{i=1..n} t_i \to s_i \leq t \to s$$

comme attendu. □

5.5 Sémantique

Nous pouvons maintenant définir la sémantique du calcul. Nous allons en fait définir une sémantique opérationnelle à petits pas, c'est-à-dire une relation de réécriture sur les expressions *closes*.

Nous matérialisons l'erreur de type par le symbole Ω, et nous posons $\mathcal{E}_\Omega = \mathcal{E} + \{\Omega\}$.

Pour chaque opérateur $o \in \mathcal{O}$, nous supposons donnée une relation binaire, notée $v \overset{o}{\rightsquigarrow} e$, où $v \in \mathcal{V}$ et $e \in \mathcal{E}_\Omega$. Pour l'opérateur **app**, nous prenons la β-réduction $(v_1, v_2) \overset{\mathbf{app}}{\rightsquigarrow} e[f := v_1; x := v_2]$ lorsque $v_1 = \mu f(\ldots).\lambda x.e$ et $v \overset{\mathbf{app}}{\rightsquigarrow} \Omega$ lorsque v n'est pas de la forme $(\mu f(\ldots).\lambda x.e, v_2)$.

Les contextes (immédiats) d'évaluation sont définis par les productions :

$$\begin{array}{rcl} C[] & ::= & ([], e) \\ & | & (e, []) \\ & | & o([]) \\ & | & (x = [] \in t \ ?\ e_1 | e_2) \end{array}$$

Nous définissons la relation de réécriture $e \rightsquigarrow e'$ (où $e \in \mathcal{E}$ et $e' \in \mathcal{E}_\Omega$) par le système inductif ci-dessous.

$$\frac{e \rightsquigarrow \Omega}{C[e] \rightsquigarrow \Omega} \quad \frac{e \rightsquigarrow e' \quad e' \neq \Omega}{C[e] \rightsquigarrow C[e']}$$

$$\frac{v \overset{o}{\rightsquigarrow} e}{o(v) \rightsquigarrow e}$$

$$\frac{i \in \{1,2\} \quad (i = 1 \iff \vdash v : t)}{(x = v \in t \ ?\ e_1 | e_2) \rightsquigarrow e_i[x := v]}$$

Notons qu'avec cette sémantique, une valeur ne se réduit pas.

5.6 Sûreté du typage

Definition 5.13 *Soient o un opérateur, et t,s deux types. On note $(o : t \Longrightarrow s)$ si :*

$$\forall v \in \mathcal{V}, \forall e \in \mathcal{E}_\Omega.\ (\vdash v : t) \mathbf{\wedge} (v \overset{o}{\rightsquigarrow} e) \quad \Rightarrow \ \vdash e : s$$

Definition 5.14 *On dit qu'un opérateur o est* **bien typé** *si :*

$$\forall t, s.\ (o : t \mathbf{\rightarrow} s) \Rightarrow (o : t \Longrightarrow s)$$

Definition 5.15 *Soit o un opérateur bien typé. On dit que le typage de o est* **exact** *si, pour tout type t tel que $(o : t \Longrightarrow \mathbb{1})$, il existe un type s tel que $(o : t \mathbf{\rightarrow} s)$ et :*

$$\forall v' \in [\![s]\!]_{\mathcal{V}}. \exists v \in [\![t]\!]_{\mathcal{V}}. \exists e \in \mathcal{E}.\ (v \overset{o}{\rightsquigarrow} e) \wedge (e \overset{*}{\rightsquigarrow} v')$$

Remarque 5.16 *Cette condition d'exactitude est une forme de réciproque à l'implication dans la Définition 5.14. Elle affirme en plus une propriété très forte, à savoir l'existence d'un type qui capture exactement l'ensemble des résultats que l'on peut obtenir en appliquant l'opérateur sur une valeur arbitraire d'un type donné. Le typage d'un opérateur exact n'introduit aucune approximation par rapport à sa sémantique et à l'approximation déjà effectuée sur le type de l'argument.*

Lemme 5.17 *Soient o un opérateur et t_1, t_2, s_1, s_2 quatre types. Alors :*

- $(o : t_1 \Longrightarrow s_1) \wedge (o : t_2 \Longrightarrow s_2) \Rightarrow (o : (t_1 \mathbf{\wedge} t_2) \Longrightarrow (s_1 \mathbf{\wedge} s_2))$,

– $(o : t_1 \Longrightarrow s_1) \wedge (o : t_2 \Longrightarrow s_2) \Rightarrow (o : (t_1 \vee t_2) \Longrightarrow (s_1 \vee s_2))$,
– $(o : t_1 \Longrightarrow s_1) \wedge (t_2 \leq t_1) \wedge (s_1 \leq s_2) \Rightarrow (o : t_2 \Longrightarrow s_2)$,

> *Preuve*: Supposons que $(o : t_1 \Longrightarrow s_1)$ et $(o : t_2 \Longrightarrow s_2)$. Prouvons tout d'abord que $(o : (t_1 \wedge t_2) \Longrightarrow (s_1 \wedge s_2))$. Considérons une valeur v de type $t_1 \wedge t_2$ et une réduction $v \overset{o}{\rightsquigarrow} e$. La valeur v est *a fortiori* de type t_1, et donc par hypothèse $\vdash e : s_1$. De même, $\vdash e : s_2$, et l'on en déduit $\vdash e : s_1 \wedge s_2$ en utilisant le Lemme 5.3.
>
> Prouvons maintenant que $(o : (t_1 \vee t_2) \Longrightarrow (s_1 \vee s_2))$. Considérons une valeur v de type $t_1 \vee t_2$ et une réduction $v \overset{o}{\rightsquigarrow} e$. La valeur v est de type t_1 ou de type t_2. Par exemple, si v est de type t_1, on a, par hypothèse $\vdash e : s_1$ et donc, *a fortiori*, $\vdash e : s_1 \vee s_2$.
>
> Le dernier point se prouve trivialement en utilisant la règle de subsomption. □

Lemme 5.18 *L'opérateur* **app** *est bien typé.*

> *Preuve*: Compte-tenu du lemme ci-dessus et de la définition axiomatique de la relation $\mathbf{app} : \_ \rightarrow \_$ (Figure 4.1), il suffit de vérifier que $(\mathbf{app} : ((t \rightarrow s) \times t) \Longrightarrow s)$ pour des types t et s arbitraires. Soit v une valeur de type $(t \rightarrow s) \times t$. D'après le Lemme 5.12, on voit qu'il s'agit d'un couple $v = (v_f, v_x)$ avec $\vdash v_x : t$ et $\vdash v_f : t \rightarrow s$. La valeur v_f est donc une abstraction $\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$, avec $\bigwedge_{i=1..n} t_i \rightarrow s_i \leq t \rightarrow s$.
>
> Si $v \overset{\mathbf{app}}{\rightsquigarrow} e'$, on a donc $e' = e[f := v_f; x := v_x]$. Il s'agit de montrer que $\vdash e' : s$.
>
> Soit $I = \{1, \ldots, n\}$ et $I' = \{i \in I \mid \vdash v_x : t_i\}$. On a $t \not\leq \bigvee_{i \in I \setminus I'} t_i$. En effet, dans le cas contraire, on aurait $v_x \in [\![\bigvee_{i \in I \setminus I'} t_i]\!]$, et donc il y aurait un $i \notin I'$ avec $\vdash v_x : t_i$, ce qui contredirait la définition de I'.
>
> Puisque $\bigwedge_{i=1..n} t_i \rightarrow s_i \leq t \rightarrow s$ et $t \not\leq \bigvee_{i \in I \setminus I'} t_i$, on obtient avec le Lemme 4.9 que $I' \neq \emptyset$ et :
>
> $$\bigwedge_{i \in I'} s_i \leq s$$
>
> Pour établir $\vdash e' : s$, il suffit donc, compte-tenu du Lemme 5.3, de prouver $\vdash e' : s_i$ pour tout $i \in I'$.
>
> Prenons donc i tel que $\vdash v_x : t_i$. La valeur v_f est bien typée ; en considérant une instance de la règle (*abstr*) qui donne un type t' à v_f, on voit que $(f : t'), (x : t_i) \vdash e : s_i$. Puisque $\vdash v_f : t'$ et $\vdash v_x : t_i$, le Lemme 5.5 donne bien $\vdash e' : s_i$. □

Nous pouvons maintenant établir un résultat classique de préservation du typage par réduction (*subject reduction*). Dans la mesure où nous avons rendu explicite l'erreur Ω dans la sémantique du calcul, ce théorème garantit l'absence d'erreur de type lorsqu'on évalue une expression bien typée.

Theorème 5.19 (Préservation du typage par réduction) *Si tous les opérateurs sont bien typés, alors le typage est préservé par réduction : si* $e \rightsquigarrow e'$ *et* $\vdash e : t$, *alors* $\vdash e' : t$. *En particulier,* $e' \neq \Omega$.

Preuve: Par induction sur la dérivation du jugement $\vdash e : t$. Si la dernière règle appliquée est (*subsum*), on a en fait $\vdash e : s$ avec $s \leq t$. Par induction, on voit que $\vdash e' : s$ et la règle (*subsum*) donne alors $\vdash e' : t$.

On suppose maintenant que la dérivation de $\vdash e : t$ ne se termine pas par une application de la règle (*subsum*). La forme syntaxique de e donne alors la dernière règle appliquée. Les cas constante, variable et abstraction sont impossibles.

Les règles de réduction en contexte se traitent trivialement. Le cas de la réduction d'un opérateur vient directement de la définition d'un opérateur bien typé.

Considérons le cas d'une expression $e = (x = v \in t \ ?\ e_1|e_2)$. La règle de typage utilisée est :

$$\frac{\Gamma \vdash v : t_0 \quad \begin{cases} t_1 = t_0 \wedge t \\ t_2 = t_0 \setminus t \end{cases} \quad \begin{cases} s_i = \mathbb{0} & \text{si } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{si } t_i \not\simeq \mathbb{0} \end{cases}}{\Gamma \vdash (x = v \in t \ ?\ e_1|e_2) : s_1 \vee s_2}$$

Il y a deux cas (mutuellement exclusifs) : $\vdash v : t$ ou $\vdash v : \neg t$. Considérons par exemple le premier cas. La réduction est alors :

$$e \rightsquigarrow e_1[x := v]$$

On a $\vdash v : t_0 \wedge t = t_1$, ce qui donne déjà $t_1 \not\simeq \mathbb{0}$ et donc $(x : t_1), \Gamma \vdash e_1 : s_1$. En appliquant le Lemme 5.5 à ce jugement, on obtient $\vdash e_1[x := v] : s_1 \leq s$. Le cas $\vdash v : \neg t$ se traite de la même manière. □

Le théorème ci-dessous donne une justification supplémentaire de notre démarche : même si l'on a définit le typage de l'application fonctionnelle **app** de manière purement ensembliste, il correspond très exactement à la sémantique du calcul.

Theorème 5.20 *Le typage de l'opérateur* **app** *est exact.*

Preuve: Soit t un type tel que $(\mathbf{app} : t \Longrightarrow \mathbb{1})$. On en déduit déjà $t \leq \mathbb{1} \times \mathbb{1}$, car sinon on pourrait trouver une valeur v telle que $\vdash v : t$ et $v \overset{\mathbf{app}}{\rightsquigarrow} \Omega$.

Si l'on prouve la propriété d'exactitude pour deux types t_1 et t_2, on en déduit la propriété pour le type $t_1 \vee t_2$, en utilisant la règle (**app**$\vee$). En écrivant (Théorème 4.36)

$$t \simeq \bigvee_{(t_f, t_x) \in \pi(t)} t_f \times t_x$$

et en décomposant chaque t_f sous la forme

$$t_f \simeq \bigvee_{(P,N) \in t_f} \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$$

on se ramène au cas où $t = t_f \times t_x$, avec $t_f = \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$, pour deux ensembles finis P et N de types flèche atomiques, avec $t_f \not\simeq \mathbb{0}$. Posons $P = \{t_i \to s_i \mid i \in I\}$ où $I = \{1, \ldots, n\}$.

Soit t_0 le type $\bigvee_{i\in I} t_i$. Si $t_x \not\leq t_0$, alors on choisit une valeur v_x dans $t_x \backslash t_0$, et l'on pose :

$$v_f = \mu f(P).\lambda x.(y = x \in t_0 \ ?\ \mathbf{app}((f,x))|\mathbf{app}(f))$$

Cette expression a le type t_f, car la deuxième branche dans le corps est ignorée lors du typage. La valeur (v_f, v_x) a donc le type t mais $(v_f, v_x) \overset{\mathbf{app}}{\rightsquigarrow} e$ où e est une expression mal typée (car elle se réduit sur $\mathbf{app}(v_f)$ qui est mal typée), ce qui contredit l'hypothèse $(\mathbf{app} : t \Longrightarrow \mathbb{1})$.

On peut donc supposer que $t_x \leq t_0$. Posons :

$$s = \bigvee_{I'\subseteq I \ \mid \ (*)} \bigwedge_{i\in I\setminus I'} s_i$$

où $(*)$ est la condition $t_x \not\leq \bigvee_{i\in I'} t_i$. Le Lemme 4.46 donne $\mathbf{app} : t \to s$. Soit v' une valeur dans s. On peut trouver I' qui vérifie $(*)$ et tel que $v' \in \bigwedge_{i\in I\setminus I'} s_i$. Soit v_x une valeur dans $t_x \backslash \bigvee_{i\in I'} t_i$ et v_f la valeur :

$$v_f = \mu f(P).\lambda x.(y = x \in \bigvee_{i\in I'} t_i \ ?\ \mathbf{app}((f,x))|v')$$

On constate que (v_f, v_x) a bien le type t, et que $(v_f, v_x) \overset{\mathbf{app}}{\rightsquigarrow} e$ avec $e \overset{*}{\rightsquigarrow} v'$. □

5.7 Inférence de types

Dans cette section, nous étudions le problème de l'inférence dans le système de types introduit à la Section 5.2. La difficulté provient du fait que le système n'a pas de *types principaux* : étant donné un environnement de typage Γ et une expression e bien typée sous Γ, il n'existe pas forcément un type t tel que :

$$\forall s.\ (\Gamma \vdash e : s) \iff t \leq s$$

Cela provient essentiellement de la règle de typage (*abstr*) qui permet toujours d'ajouter des types flèche négatifs. La règle (*op*) peut aussi empêcher l'existence de types principaux.

5.7.1 Filtres, schémas

L'idée est d'introduire des objets syntaxiques, appelés schémas, chacun représentant un ensemble de types, de sorte à pouvoir capturer tous les types d'une certaine expression e dans un environnement Γ par un unique schéma. La grammaire pour les schémas est :

$$\begin{array}{lcll} \mathbb{t} & ::= & t & t \in \widehat{T} \\ & \mid & [t_1 \to s_1; \ldots; t_n \to s_n] & n \geq 1; t_i, s_i \in \widehat{T} \\ & \mid & \mathbb{t}_1 \otimes \mathbb{t}_2 & \\ & \mid & \mathbb{t}_1 \oslash \mathbb{t}_2 & \\ & \mid & \Omega & \end{array}$$

Nous écrirons également $[t_i{\rightarrow}s_i]_{i=1..n}$ pour $[t_1{\rightarrow}s_1;\ldots;t_n{\rightarrow}s_n]$.

À chaque schéma $\mathbb{t}$, nous voulons associer un ensemble de types $\{\!|\mathbb{t}|\!\}$. L'idée étant de représenter l'ensemble des types d'une expression par un schéma, nous serons guidés par la définition et le lemme suivant.

Definition 5.21 *Un* **filtre** *est un ensemble de types $\mathcal{F} \subseteq \widehat{T}$ clos par intersection et subsomption :*
- $\forall t_1, t_2.\ t_1 \in \mathcal{F} \wedge t_2 \in \mathcal{F} \Rightarrow t_1 \mathbf{\wedge} t_2 \in \mathcal{F}$
- $\forall t_1, t_2.\ t_1 \in \mathcal{F} \wedge t_1 \leq t_2 \Rightarrow t_2 \in \mathcal{F}$

Lemme 5.22 *Soit Γ un environnement de typage et e une expression, alors l'ensemble $\{t \mid \Gamma \vdash e : t\}$ est un filtre.*

> *Preuve*: Conséquence de la règle (*subsum*) et du Lemme 5.3. □

Definition 5.23 *Nous définissons la fonction $\{\!|\_|\!\}$ qui associe à un schéma un ensemble de types par :*

$$
\begin{array}{lcl}
\{\!|t|\!\} & = & \{s \mid t \leq s\} \\
\{\!|[t_i{\rightarrow}s_i]_{i=1..n}|\!\} & = & \{s \mid \exists s_0 = \bigwedge_{i=1..n} (t_i{\rightarrow}s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j{\rightarrow}s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\} \\
\{\!|\mathbb{t}_1 \otimes \mathbb{t}_2|\!\} & = & \{s \mid \exists t_1 \in \{\!|\mathbb{t}_1|\!\}, t_2 \in \{\!|\mathbb{t}_2|\!\}.\ t_1 \mathbf{\times} t_2 \leq s\} \\
\{\!|\mathbb{t}_1 \oslash \mathbb{t}_2|\!\} & = & \{s \mid \exists t_1 \in \{\!|\mathbb{t}_1|\!\}, t_2 \in \{\!|\mathbb{t}_2|\!\}.\ t_1 \mathbf{\vee} t_2 \leq s\} \\
\{\!|\Omega|\!\} & = & \emptyset
\end{array}
$$

Lemme 5.24 *Pour tout schéma $\mathbb{t}$, l'ensemble $\{\!|\mathbb{t}|\!\}$ est un filtre. On peut décider s'il est vide.*

> *Preuve*: Induction facile sur la syntaxe des filtres. Le cas $[t_i{\rightarrow}s_i]_{i=1..n}$ se traite avec le Lemme 5.1.
>
> Si l'on voit un schéma $\mathbb{t}$ comme un arbre binaire dont les nœuds sont $\otimes$ et $\oslash$, alors le filtre $\{\!|\mathbb{t}|\!\}$ est vide si et seulement si l'une des feuilles est Ω. □

Remarque 5.25 *Il est facile de vérifier l'égalité :*

$$\{\!|\mathbb{t}_1 \oslash \mathbb{t}_2|\!\} = \{\!|\mathbb{t}_1|\!\} \cap \{\!|\mathbb{t}_2|\!\}$$

Lemme 5.26 *Soit t_0 un type et $\mathbb{t}$ un schéma. Alors on peut calculer un schéma, noté $t_0 \oslash \mathbb{t}$, tel que :*

$$\{\!|t_0 \oslash \mathbb{t}|\!\} = \{s \mid \exists t \in \{\!|\mathbb{t}|\!\}.t \mathbf{\wedge} t_0 \leq s\}$$

et de plus $\#(t_0 \oslash \mathbb{t}) \leq \#\mathbb{t}$, où $\#\mathbb{t}$ désigne le nombre maximal de constructeurs $\otimes$ imbriqués dans $\mathbb{t}$.

> *Preuve*: La preuve donne une construction pour $t_0 \oslash \mathbb{t}$. La vérification de la propriété attendue est aisée. Nous procédons par induction sur la syntaxe de $\mathbb{t}$. Si $\mathbb{t}$ est un type t, on prend :
>
> $$t_0 \oslash t = t_0 \mathbf{\wedge} t$$

Si $\mathbb{t}$ est une réunion $\mathbb{t}_1 \vee \mathbb{t}_2$, on distribue :

$$t_0 \wedge (\mathbb{t}_1 \vee \mathbb{t}_2) = (t_0 \wedge \mathbb{t}_1) \vee (t_0 \wedge \mathbb{t}_2)$$

Si $\mathbb{t}$ est Ω, on prend $t_0 \wedge \mathbb{t} = \Omega$.

Pour les deux cas $[t_i \to s_i]_{i=1..n}$ et $\mathbb{t}_1 \otimes \mathbb{t}_2$, nous utilisons le Lemme 3.1, qui nous permet de voir t_0 comme une combinaison booléenne d'atomes. Les deux équations suivantes permettent de se ramener au cas où t_0 est un atome ou une négation d'atome :

$$(t_1 \vee t_2) \wedge \mathbb{t} = (t_1 \wedge \mathbb{t}) \vee (t_2 \wedge \mathbb{t})$$

$$(t_1 \wedge t_2) \wedge \mathbb{t} = t_1 \wedge (t_2 \wedge \mathbb{t})$$

Pour le cas $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, on prend :

$$(t_1 \times t_2) \wedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = (t_1 \wedge \mathbb{t}_1) \otimes (t_2 \wedge \mathbb{t}_2)$$

$$\neg(t_1 \times t_2) \wedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = ((\neg t_1 \wedge \mathbb{t}_1) \otimes \mathbb{t}_2) \vee (\mathbb{t}_1 \otimes (\neg t_2 \wedge \mathbb{t}_2))$$

et si a est un atome qui n'est pas de la forme $t_1 \times t_2$:

$$a \wedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = \mathbb{0}$$

$$\neg a \wedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = (\mathbb{t}_1 \otimes \mathbb{t}_2)$$

Pour le cas $\mathbb{t} = [t_i \to s_i]_{i=1..n}$, on prend :

$$(t \to s) \wedge [t_i \to s_i]_{i=1..n} = \begin{cases} [t_i \to s_i]_{i=1..n} & \text{si } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ \mathbb{0} & \text{si } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

$$\neg(t \to s) \wedge [t_i \to s_i]_{i=1..n} = \begin{cases} \mathbb{0} & \text{si } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ [t_i \to s_i]_{i=1..n} & \text{si } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

et si a est un atome qui n'est pas de la forme $t \to s$:

$$a \wedge [t_i \to s_i]_{i=1..n} = \mathbb{0}$$

$$\neg a \wedge [t_i \to s_i]_{i=1..n} = [t_i \to s_i]_{i=1..n}$$

□

Lemme 5.27 *Soit $\mathbb{t}$ un schéma et t un type. Alors on peut décider si l'assertion $t \in \{\mathbb{t}\}$ est vraie ou fausse. On note $\mathbb{t} \leq t$ lorsqu'elle est vraie.*

Preuve: Commençons par observer l'équivalence :

$$t \in \{\mathbb{t}\} \iff \mathbb{0} \in \{(\neg t) \wedge \mathbb{t}\}$$

On constate en effet que : $\mathbb{0} \in \{(\neg t) \wedge \mathbb{t}\} \iff \exists s \in \{\mathbb{t}\}.\ (\neg t) \wedge s \leq \mathbb{0} \iff \exists s \in \{\mathbb{t}\}.\ s \leq t \iff t \in \{\mathbb{t}\}$.

On se ramène ainsi au cas $t = \mathbb{0}$, et on conclut par induction sur la syntaxe de $\mathbb{t}$, en utilisant :

$$\begin{array}{lcl}
\mathbb{0} \in \{t\} & \iff & t \simeq \mathbb{0} \\
\mathbb{0} \notin \{[t_i \to s_i]_{i=1..n}\} & & \\
\mathbb{0} \in \{\mathbb{t}_1 \otimes \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \vee (\mathbb{0} \in \{\mathbb{t}_2\}) \\
\mathbb{0} \in \{\mathbb{t}_1 \vee \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \wedge (\mathbb{0} \in \{\mathbb{t}_2\}) \\
\mathbb{0} \notin \{\Omega\} & &
\end{array}$$

□

5.7.2 Typage des opérateurs

Nous avons presque tous les ingrédients nécessaires pour reformuler le système de types avec les schémas. Il ne reste plus qu'à supposer que les opérateurs se comportent bien vis-à-vis des schémas. Pour un opérateur o et un schéma $\mathbb{t}$, considérons l'ensemble :

$$\{s \mid \exists t' \geq \mathbb{t}. \exists s' \leq s.\ o : t' \to s'\}$$

Il est aisé de voir qu'il s'agit d'un filtre.

Definition 5.28 *Une* **fonction de typage** *pour l'opérateur o est une fonction des schémas dans les schémas, notée $o[\_]$, telle que, pour tout schéma $\mathbb{t}$:*

$$\{o[\mathbb{t}]\} = \{s \mid \exists t' \geq \mathbb{t}. \exists s' \leq s.\ o : t' \to s'\}$$

Lemme 5.29 *Il existe une fonction de typage calculable pour l'opérateur* **app**.

Preuve: La preuve repose sur le Théorème 4.42 et le Lemme 4.37. Pour un filtre $\mathcal{F}$, notons :

$$\begin{array}{rcl}
X_{\mathcal{F}} & = & \{s \mid \exists t' \in \mathcal{F}. \exists s' \leq s.\ \mathbf{app} : t' \to s'\} \\
& = & \{s \mid \exists t' \in \mathcal{F}.\ \mathbf{app} : t' \to s\}
\end{array}$$

En utilisant le Théorème 4.42, on voit que si $\mathbb{t} \not\leq (\mathbb{0} \to \mathbb{1}) \times \mathbb{1}$, alors $X_{\{\mathbb{t}\}} = \emptyset$, et dans ce cas, nous prenons $\mathbf{app}[\mathbb{t}] = \Omega$. Cela permet de traiter en particulier les cas $\mathbb{t} = \Omega$, $\mathbb{t} = [t_i \to s_i]_{i=1..n}$ et $\mathbb{t} = (\mathbb{t}_1 \otimes \mathbb{t}_2) \otimes \mathbb{t}_3$. Si $\mathbb{t} \leq \mathbb{0}$, on prend $\mathbf{app}[\mathbb{t}] = \mathbb{0}$. Dorénavant, nous supposons que $\mathbb{t} \leq (\mathbb{0} \to \mathbb{1}) \times \mathbb{1}$ et $\mathbb{t} \not\leq \mathbb{0}$.

En utilisant la règle (**appV**), on établit immédiatement l'égalité :

$$X_{\mathcal{F}_1 \cap \mathcal{F}_2} = X_{\mathcal{F}_1} \cap X_{\mathcal{F}_2}$$

Cela permet de poser :

$$\begin{array}{lcl}
\mathbf{app}[\mathbb{t}_1 \vee \mathbb{t}_2] & := & \mathbf{app}[\mathbb{t}_1] \vee \mathbf{app}[\mathbb{t}_2] \\
\mathbf{app}[(\mathbb{t}_1 \vee \mathbb{t}'_1) \otimes \mathbb{t}_2] & := & \mathbf{app}[\mathbb{t}_1 \otimes \mathbb{t}_2] \vee \mathbf{app}[\mathbb{t}'_1 \otimes \mathbb{t}_2] \\
\mathbf{app}[t] & := & \bigvee\limits_{(t_1,t_2) \in \pi(t)} \mathbf{app}[t_1 \otimes t_2]
\end{array}$$

En utilisant ces définitions, on se ramène au cas $\mathbb{t}_1 \otimes \mathbb{t}_2$ où $\mathbb{t}_1$ est soit un type t_1 (avec $t_1 \leq \mathbb{0} \to \mathbb{1}$ et $t_1 \not\leq \mathbb{0}$) soit de la forme $[t'_i \to s'_i]_{i=1..n}$. Le Corollaire 4.45 donne :

$$X_{\{\mathbb{t}_1 \otimes \mathbb{t}_2\}} = \{s \mid \exists t_2 \geq \mathbb{t}_2.\ \mathbb{t}_1 \leq t_2 \to s\}$$

Dans la mesure où :

$$[t'_i \to s'_i]_{i=1..n} \leq t_2 \to s \iff \bigwedge_{i=1..n} t'_i \to s'_i \leq t_2 \to s$$

on se ramène à l'unique cas où $\mathbb{t}_1$ est un type t_1, et l'on a :

$$X_{\{t_1 \otimes \mathbb{t}_2\}} = \{s \mid \exists t_2 \geq \mathbb{t}_2.\ t_1 \leq t_2 \to s\}$$

Le Lemme 4.37 donne :

$$t_1 \leq t_2 \to s \iff \begin{cases} t_2 \leq \mathrm{Dom}(t_1) \\ \forall (s_1, s_2) \in \rho(t_1).\ (t_2 \leq s_1) \vee (s_2 \leq s) \end{cases}$$

On obtient facilement les équivalences suivantes :

$$\begin{array}{ll} & \exists t_2 \geq \mathbb{t}_2.\ t_1 \leq t_2 \to s \\ \iff & \exists \mathbb{t}_2 \leq t_2 \leq \mathrm{Dom}(t_1).\ \forall (s_1, s_2) \in \rho(t_1).\ (t_2 \leq s_1) \vee (s_2 \leq s) \\ \overset{*}{\iff} & \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \wedge \forall (s_1, s_2) \in \rho(t_1).\ (\mathbb{t}_2 \leq s_1) \vee (s_2 \leq s) \\ \iff & \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \wedge s \geq \bigvee_{(s_1,s_2) \in \rho(t_1) \mid \mathbb{t}_2 \not\leq s_1} s_2 \end{array}$$

Seul le sens $\Leftarrow$ l'équivalence marqué d'une étoile mérite un commentaire. Supposons que pour chaque couple $(s_1, s_2) \in \rho(t_1)$, on a $\mathbb{t}_2 \leq s_1$ ou $s_2 \leq s$. On définit alors t_2 comme l'intersection de $\mathrm{Dom}(t_1)$ et des types s_1 tels que $(s_1, s_2) \in \rho(t_1)$ et $\mathbb{t}_2 \leq s_1$. On a bien $\mathbb{t}_2 \leq t_2$.

On en déduit finalement :

$$\begin{array}{lcll} \mathbf{app}[t_1 \otimes \mathbb{t}_2] & := & \Omega & \text{si } \mathbb{t}_2 \not\leq \mathrm{Dom}(t_1) \\ & := & \bigvee_{(s_1,s_2) \in \rho(t_1) \mid \mathbb{t}_2 \not\leq s_1} s_2 & \text{si } \mathbb{t}_2 \leq \mathrm{Dom}(t_1) \end{array}$$

□

5.7.3 Algorithme de typage

Pour le reste de cette section, nous supposons que tous les opérateurs possèdent une fonction de typage calculable. Nous pouvons alors définir un algorithme de typage. On appelle « environnement de typage par schémas » une fonction partielle à domaine finie $\mathbb{\Gamma}$ des variables dans les schémas, telle que $\{\mathbb{\Gamma}(x)\} \neq \emptyset$ pour tout x dans le domaine de $\mathbb{\Gamma}$. L'algorithme est présenté dans la Figure 5.3 sous la forme d'une fonction qui associe à environnement $\mathbb{\Gamma}$ et à une expression e un schéma, noté $\mathbb{\Gamma}[e]$. Cette fonction est définie par induction sur la syntaxe de e.

Si Γ (resp. $\mathbb{\Gamma}$) est un environnement de typage (resp. par schémas), alors on note $\mathbb{\Gamma} \leq \Gamma$ lorsque $\mathbb{\Gamma}$ et Γ ont le même domaine, et pour tout x : $\mathbb{\Gamma}(x) \leq \Gamma(x)$.

Lemme 5.30 *Si $\mathbb{\Gamma} \leq \Gamma_1$ et $\mathbb{\Gamma} \leq \Gamma_2$, alors il existe un environnement Γ tel que : $\mathbb{\Gamma} \leq \Gamma$, $\Gamma \leq \Gamma_1$, $\Gamma \leq \Gamma_2$.*

Preuve: Les trois environnements $\mathbb{\Gamma}$, Γ_1 et Γ_2 ont le même domaine. Si x est dans ce domaine, on pose $\Gamma(x) = \Gamma_1(x) \wedge \Gamma_2(x)$, et sinon $\Gamma(x)$ n'est pas défini. □

$$
\left\{\begin{array}{l}
\mathbb{\Gamma}[c] := b_c \\
\mathbb{\Gamma}[(e_1,e_2)] := \mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e] := \left\{\begin{array}{ll} \mathbb{t} & \text{si } \forall i = 1..n.\ \mathbb{s}_i \leq s_i \\ \Omega & \text{sinon} \end{array}\right. \\
\qquad\qquad \text{où } \left\{\begin{array}{l} \mathbb{t} = [t_i \rightarrow s_i]_{i=1..n} \\ \mathbb{s}_i = ((f:\mathbb{t}),(x:t_i),\mathbb{\Gamma})[e] \quad (i = 1..n) \end{array}\right. \\
\mathbb{\Gamma}[x] := \left\{\begin{array}{lll} \mathbb{\Gamma}(x) & \text{si} & \mathbb{\Gamma}(x) \text{ est défini} \\ \Omega & \text{sinon} & \end{array}\right. \\
\mathbb{\Gamma}[o(e)] := o[\mathbb{\Gamma}[e]] \\
\mathbb{\Gamma}[(x = e \in t\ ?\ e_1 | e_2)] := \mathbb{s}_1 \between \mathbb{s}_2 \\
\text{où } \left\{\begin{array}{l}
\mathbb{t}_0 := \mathbb{\Gamma}[e] \\
\mathbb{t}_1 := t \between \mathbb{t}_0 \\
\mathbb{t}_2 := (\neg t) \between \mathbb{t}_0 \\
\mathbb{s}_i := \left\{\begin{array}{lll} ((x:\mathbb{t}_i),\mathbb{\Gamma})[e_i] & \text{si } \mathbb{t}_i \not\leq \mathbb{0}, \{\mathbb{t}_i\} \neq \emptyset & \\ \mathbb{0} & \text{si } \mathbb{t}_i \leq \mathbb{0} & (i = 1..2) \\ \Omega & \text{si } \{\mathbb{t}_i\} = \emptyset & \end{array}\right.
\end{array}\right.
\end{array}\right.
$$

FIG. 5.3 – Algorithme de typage

Lemme 5.31 (Correction) *Si* $\mathbb{\Gamma}[e] \leq t$*, alors il existe* $\Gamma \geq \mathbb{\Gamma}$ *tel que* $\Gamma \vdash e : t$.

Preuve: Induction sur la structure de l'expression e. Nous traitons les deux cas les plus intéressants.
Cas $e' = (x = e \in t\ ?\ e_1|e_2)$. On prend $\mathbb{t}_0, \mathbb{t}_1, \mathbb{t}_2, \mathbb{s}_1, \mathbb{s}_2$ comme dans la définition de $\mathbb{\Gamma}[(x = e \in t\ ?\ e_1|e_2)]$. Soit s_1, s_2 tels que $\mathbb{s}_1 \leq s_1$ et $\mathbb{s}_2 \leq s_2$. Il s'agit de voir qu'il existe $\Gamma \geq \mathbb{\Gamma}$ tel que $\Gamma \vdash e' : s_1 \vee s_2$. Prenons $i = 1..2$. Puisque $\mathbb{s}_i \leq s_i$, on a $\mathbb{s}_i \neq \Omega$. Il y a deux cas. Si $\mathbb{t}_i \not\leq \mathbb{0}$, alors $\mathbb{s}_i = ((x:\mathbb{t}_i),\mathbb{\Gamma})[e_i] \leq s_i$. D'après l'hypothèse d'induction, il existe $\Gamma_i \geq \mathbb{\Gamma}$ et $t_i \geq \mathbb{t}_i$ tels que $(x:t_i), \Gamma_i \vdash e_i : s_i$. Si $\mathbb{t}_i \leq \mathbb{0}$, alors on pose $t_i = \mathbb{0}$.

Posons $t_0 = (t_1 \wedge t) \vee (t_2 \setminus t)$. Prouvons que $t_0 \geq \mathbb{t}_0$. On a $t_1 \geq \mathbb{t}_1 = t \between \mathbb{t}_0$, et donc il existe $t'_1 \geq \mathbb{t}_0$ tel que $t \wedge t'_1 \leq t_1$. De même, il existe $t'_2 \geq \mathbb{t}_0$ tel que $(\neg t) \wedge t'_2 \leq t_2$. On en déduit : $t_0 \geq (t \wedge t'_1 \wedge t'_2) \vee ((\neg t) \wedge t'_1 \wedge t'_2) \simeq t'_1 \wedge t'_2 \geq \mathbb{t}_0$. Posons $t''_1 = t_0 \wedge t \leq t_1$ et $t''_2 = t_0 \setminus t \leq t_2$.

On peut donc appliquer encore une fois l'hypothèse d'induction, qui donne l'existence d'un $\Gamma_0 \geq \mathbb{\Gamma}$ tel que $\Gamma_0 \vdash e : t_0$. En utilisant les Lemmes 5.2 et 5.30, on voit qu'il existe un environnement $\Gamma \geq \mathbb{\Gamma}$ tel que $\Gamma \vdash e : t_0$, et $(x : t''_i), \Gamma \vdash e_i : s_i$ pour $i = 1..2$ (lorsque $\mathbb{t}_i \not\leq \mathbb{0}$). On conclut en utilisant la règle (*case*) et éventuellement (*subsum*) que $\Gamma \vdash e' : s_1 \vee s_2$.
Cas $e' = \mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e$. On prend $\mathbb{t}$ et $\mathbb{s}_i$ comme dans la définition de $\mathbb{\Gamma}[e']$. Puisque $\mathbb{\Gamma}[e'] \neq \Omega$ (car $\mathbb{\Gamma}[e'] \leq t$), on a $\mathbb{\Gamma}[e'] = \mathbb{t}$ et $\mathbb{s}_i \leq s_i$ pour tout $i = 1..n$. L'hypothèse d'induction montre

que pour chaque i, on peut trouver un environnement $\Gamma = ((f : t^i), (x : t''_i), \Gamma_i) \geq ((f : \mathbb{t}), (x : t_i), \mathbb{\Gamma})$ tel que $\Gamma \vdash e : s_i$. Posons $t' = \bigwedge_{i=1..n} t^i \wedge t$.

On a $t' \geq \mathbb{t}$, et donc on peut trouver une famille $(t'_j \to s'_j)_{j=1..m}$ telle que $t' \geq t'' \not\simeq \mathbb{0}$ où $t'' = \bigwedge_{i=1..n}(t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$.

En utilisant les Lemme 5.2 et 5.30, on obtient un $\Gamma \geq \mathbb{\Gamma}$ tel que, pour tout $i : (f : t''), (x : t_i), \Gamma \vdash e : s_i$. On peut appliquer la règle (*abstr*) pour obtenir $\Gamma \vdash e' : t''$, ce qui permet de conclure, puisque $t'' \leq t$. □

Lemme 5.32 (Complétude) *Si* $\mathbb{\Gamma} \leq \Gamma$ *et* $\Gamma \vdash e : t$, *alors* $\mathbb{\Gamma}[e] \leq t$.

Preuve: Induction sur la dérivation de typage $\Gamma \vdash e : t$. La preuve est mécanique. Considérons par exemple le cas où la dérivation se termine par une instance de la règle (*case*) :

$$\frac{\Gamma \vdash e : t_0 \quad \begin{cases} t_1 = t_0 \wedge t \\ t_2 = t_0 \setminus t \end{cases} \quad \begin{cases} s_i = \mathbb{0} & \text{si } t_i \simeq \mathbb{0} \\ (x : t_i), \Gamma \vdash e_i : s_i & \text{si } t_i \not\simeq \mathbb{0} \end{cases}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s_1 \vee s_2}$$

On prend $\mathbb{t}_0, \mathbb{t}_1, \mathbb{t}_2, \mathbb{s}_1, \mathbb{s}_2$ comme dans la définition de $\mathbb{\Gamma}[(x = e \in t \ ?\ e_1|e_2)]$. En appliquant l'hypothèse d'induction, on obtient $\mathbb{t}_0 = \mathbb{\Gamma}[e] \leq t_0$, et on en déduit $\mathbb{t}_1 \leq t_1$ et $\mathbb{t}_2 \leq t_2$. Prenons $i = 1..2$. Si $\mathbb{t}_i \leq \mathbb{0}$, alors $\mathbb{s}_i = \mathbb{0}$ et donc $\mathbb{s}_i \leq s_i$. Dans le cas contraire, puisque $\{\mathbb{t}_i\} \neq \emptyset$, on a $\mathbb{s}_i = ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i]$. Mais $t_i \not\leq \mathbb{0}$ (car sinon $\mathbb{t}_i \leq \mathbb{0}$), et donc $(x : t_i), \Gamma \vdash e_i : s_i$. L'hypothèse d'induction donne encore $\mathbb{s}_i \leq s_i$ dans ce cas. Finalement, on a bien $\mathbb{s}_1 \vee \mathbb{s}_2 \leq s_1 \vee s_2$. □

En combinant les deux lemmes précédents, on obtient le théorème ci-dessous.

Theorème 5.33 *Soit* $\mathbb{\Gamma}$ *un environnement de typage par schémas et e une expression. Alors :*

$$\{\mathbb{\Gamma}[e]\} = \{t \mid \exists \Gamma.\ (\mathbb{\Gamma} \leq \Gamma) \wedge (\Gamma \vdash e : t)\}$$

Le corollaire suivant donne un algorithme pour décider si une expression est bien typée dans un environnement de typage.

Corollaire 5.34 *Soit* Γ *un environnement de typage. On peut aussi le voir comme un environnement de typage par schémas. Alors, pour toute expression e et tout type t, on a :*

$$\Gamma \vdash e : t \iff \Gamma[e] \leq t$$

En particulier, l'expression e est bien typée sous l'environnement Γ *si et seulement si* $\{\Gamma[e]\} \neq \emptyset$.

5.7.4 Interprétation ensembliste des schémas

Nous avons introduit les schémas comme des représentants syntaxiques d'ensembles de types assignés à une valeur par le système de types. Une vision duale

consiste à dire qu'un schéma *non vide* représente un ensemble de valeurs, défini par :

$$[\![\mathbb{t}]\!]_{\mathcal{V}} := \bigcap_{t \in \{\mathbb{t}\}} [\![t]\!]_{\mathcal{V}}$$

Cette définition est cohérente dans le cas où $\mathbb{t}$ est un type. Nous allons voir que les différents opérateurs que nous avons introduit sur les schémas se comportent comme on s'y attend d'un point de vue ensembliste. Les propriétés suivantes sont faciles à établir :

$$[\![\mathbb{t}_1 \otimes \mathbb{t}_2]\!]_{\mathcal{V}} = [\![\mathbb{t}_1]\!]_{\mathcal{V}} \times [\![\mathbb{t}_2]\!]_{\mathcal{V}}$$

$$[\![\mathbb{t}_1 \oslash \mathbb{t}_2]\!]_{\mathcal{V}} = [\![\mathbb{t}_1]\!]_{\mathcal{V}} \cup [\![\mathbb{t}_2]\!]_{\mathcal{V}}$$

$$[\![t \oslash \mathbb{t}]\!]_{\mathcal{V}} = [\![t]\!]_{\mathcal{V}} \cap [\![\mathbb{t}]\!]_{\mathcal{V}}$$

L'ensemble $[\![[t_i \to s_i]_{i=1..n}]\!]_{\mathcal{V}}$ contient toutes les abstractions dont l'intersection des types de l'interface est équivalente à $\bigwedge_i (t_i \to s_i)$; en particulier, il n'est jamais vide. On voit alors, par induction sur $\mathbb{t}$, que $[\![\mathbb{t}]\!]_{\mathcal{V}} = \emptyset \iff \mathbb{t} \leq \mathbb{0}$, et l'on en déduit $\mathbb{t} \leq t \iff [\![\mathbb{t}]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}$ (car $\mathbb{t} \leq t \iff (\neg t) \oslash \mathbb{t} \leq \mathbb{0}$).

Il est intéressant de constater que ces propriétés deviennent fausses en général si l'on interprète les schémas comme des ensembles d'éléments d'un modèle. Par exemple, dans le modèle universel de la Section 4.5, l'ensemble

$$\bigcap_{t \in \{\mathbb{t}\}} [\![t]\!]^0$$

est toujours vide pour $\mathbb{t} = [\mathbb{0} \to \mathbb{0}]$. Un élément de cet ensemble serait en effet un graphe *fini* $f = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$ avec $d'_i \in D^0_\Omega$. Mais $\mathbb{t} \leq \neg(t \to \mathbb{1})$ dès que $t \not\simeq \mathbb{0}$, donc pour n'importe quel type t non vide, il doit y avoir un couple (d_i, Ω) dans f, avec d_i dans $[\![t]\!]^0$. Comme on peut construire une infinité de types t deux à deux disjoints, cela est en contradiction avec la finitude de f.

Cela peut sembler contradictoire avec le fait que le modèle des valeurs est équivalent au modèle d'amorce, c'est-à-dire que l'interprétation d'un type dans l'un et dans l'autre sont simultanément vides ou non vides. En fait, nous venons simplement de mettre en évidence que cette propriété ne se transpose pas aux schémas, qui sont, intuitivement, des intersections *infinies* de types. C'est la raison pour laquelle nous ne mettons pas en avant l'interprétation ensembliste des schémas dans le formalisme introduit.

5.8 Opérateurs

Le calcul que nous avons introduit est paramétré par un ensemble d'opérateurs $\mathcal{O}$. Pour chaque opérateur $o \in \mathcal{O}$, il faut définir :

- une relation binaire $\_ \overset{o}{\rightsquigarrow} \_$ où le premier argument est une valeur et le deuxième est une expression ou Ω ;
- une relation binaire entre types $o : \_ \to \_$, close par intersection (si $o : t_1 \to s_1$ et $o : t_2 \to s_2$, alors $o : (t_1 \wedge t_2) \to (s_1 \wedge s_2)$) et telle que o soit bien typé (Définition 5.14) ;
- une fonction de typage (Définition 5.28).

Évidemment, le choix des opérateurs dépend des types de base choisis. Par exemple, si l'on dispose d'un type de base `int` qui représente des entiers, nous aurons probablement un opérateur + avec + : `int`×`int`→`int`.

Nous pouvons cependant définir des opérateurs génériques, qui ne dépendent pas des types de base choisis. Nous avons déjà étudié l'opérateur **app**. Nous allons maintenant introduire l'opérateur de première projection π_1 (l'opérateur π_2 se traite de même, évidemment). Sa sémantique est donnée par : $(v_1, v_2) \overset{\pi_1}{\rightsquigarrow} v_1$ et $v \overset{\pi_1}{\rightsquigarrow} \Omega$ lorsque v n'est pas de la forme (v_1, v_2). La relation $\pi_1 : \_\!\rightarrow\!\_$ est définie par le système inductif de la Figure 5.4.

$$\frac{}{\pi_1 : (t_1 \times t_2) \rightarrow t_1}\ (\pi_1\times)$$

$$\frac{\pi_1 : t_1 \rightarrow s_1 \quad \pi_1 : t_2 \rightarrow s_2}{\pi_1 : (t_1 \wedge t_2) \rightarrow (s_1 \wedge s_2)}\ (\pi_1\wedge)$$

$$\frac{\pi_1 : t_1 \rightarrow s_1 \quad \pi_1 : t_2 \rightarrow s_2}{\pi_1 : (t_1 \vee t_2) \rightarrow (s_1 \vee s_2)}\ (\pi_1\vee)$$

$$\frac{\pi_1 : t' \rightarrow s' \quad t \leq t' \quad s' \leq s}{\pi_1 : t \rightarrow s}\ (\pi_1 \leq)$$

FIG. 5.4 – Axiomatisation du typage de la première projection

Lemme 5.35 *L'opérateur π_1 est bien typé.*

> *Preuve*: D'après le Lemme 5.17, il suffit de considérer la règle $(\pi_1\times)$. Si $\vdash v : t_1 \times t_2$ et $v \overset{\pi_1}{\rightsquigarrow} e$, il s'agit de montrer que $e \neq \Omega$ et $\vdash e : t_1$. Or d'après le Lemme 5.12, si $\vdash v : t_1 \times t_2$, on a $v = (v_1, v_2)$ avec $\vdash v_1 : t_1$, et la seule réduction possible pour v est $v \overset{\pi_1}{\rightsquigarrow} v_1$. □

Étudions maintenant le typage de l'opérateur π_1. Nous commençons par établir l'analogue du Théorème 4.42. La preuve est plus directe car on peut interpréter la première projection directement dans le modèle $\mathbb{E}D$.

Lemme 5.36 *Soient t et s deux types. Les assertions suivantes sont équivalentes :*

(i) $\pi_1 : t \rightarrow s$
(ii) $t \leq s \times \mathbb{1}$ *(i.e. :* $\mathbb{E}[\![t]\!] \subseteq [\![s]\!] \times D$*)*
(iii) $(t \leq \mathbb{1} \times \mathbb{1}) \wedge \forall (t_1, t_2) \in \pi(t).\ t_1 \leq s$

> *Preuve*: La preuve de $(i) \Rightarrow (ii)$ est une induction facile sur la dérivation de $\pi_1 : t \rightarrow s$. Prouvons $(ii) \Rightarrow (iii)$. Si $(t_1, t_2) \in \pi(t)$, on a $t_1 \times t_2 \leq t$, ce qui donne $t_1 \times t_2 \leq s \times \mathbb{1}$, et puisque $t_2 \not\simeq \mathbb{0}$, on obtient bien $t_1 \leq s$. La preuve de $(iii) \Rightarrow (i)$ est analogue à celle du Théorème 4.42. □

Corollaire 5.37 *Soient t_1, t_2 et s trois types. Alors :*

$$\pi_1 : t_1 \times t_2 \rightarrow s \iff (t_1 \leq s) \vee (t_2 \simeq \mathbb{0})$$

On voit alors facilement que la fonction $\pi_1[\_]$ définie ci-dessous est une fonction de typage pour π_1 :

$$\begin{array}{lcl}
\pi_1[t] & = & \left\{ \begin{array}{ll} \bigvee_{(t_1,t_2)\in\pi(t)} t_1 & \text{si } t \leq \mathbb{1}\times\mathbb{1} \\ \Omega & \text{sinon} \end{array} \right. \\
\pi_1[t_i{\rightarrow}s_i]_{i=1..n} & = & \Omega \\
\pi_1[\mathbb{t}_1 \otimes \mathbb{t}_2] & = & \left\{ \begin{array}{ll} \mathbb{0} & \text{si } \mathbb{t}_2 \leq \mathbb{0} \\ \mathbb{t}_1 & \text{sinon} \end{array} \right. \\
\pi_1[\mathbb{t}_1 \vee \mathbb{t}_2] & = & \pi_1[\mathbb{t}_1] \vee \pi_1[\mathbb{t}_2] \\
\pi_1[\Omega] & = & \Omega
\end{array}$$

Lemme 5.38 *Le typage de l'opérateur π_1 est exact.*

> *Preuve*: Soit t tel que $(\pi_1 : t \Longrightarrow \mathbb{1})$. On voit que cela signifie $t \leq \mathbb{1}{\times}\mathbb{1}$. Posons alors $s = \pi_1[t]$. On voit facilement que $\pi_1 : t{\rightarrow}s$. Soit v' une valeur dans s. On peut trouver un couple $(t_1, t_2) \in \pi(t)$ tel que $v' \in [\![t_1]\!]$. On pose $v = (v', v_2)$ où v_2 est une valeur arbitraire de type t_2 (il en existe car t_2 n'est pas vide). On a bien $v \overset{\pi_1}{\rightsquigarrow} v'$. □

5.9 Variante du calcul sans fonctions surchargées

Les λ-abstractions du calcul introduit dans ce chapitre représentent des fonctions surchargées : leur interface peut spécifier plusieurs types flèche simultanément. Considérons la restriction syntaxique qui consiste à n'autoriser qu'une seule flèche. La syntaxe d'une abstraction est alors :

$$\mu f(t{\rightarrow}s).\lambda x.e$$

Cette restriction syntaxique est stable par la sémantique. En particulier, l'opérateur d'application n'introduit pas de fonctions surchargée dans un programme qui n'en possède pas. On en déduit immédiatement que le résultat de préservation du typage par réduction est toujours valide dans le calcul restreint.

Cependant, la relation de sous-typage, induite par le modèle choisi initialement, ne correspond plus à l'inclusion des ensembles de valeurs. Autrement dit, le Théorème 5.6 est faux. Voyons pourquoi. Soient t_1, t_2, s_1, s_2 quatre types. Considérons le type $(t_1{\rightarrow}s_1){\wedge}(t_2{\rightarrow}s_2)$. Les valeurs de ce type sont les abstractions bien typées de la forme $\mu f(t{\rightarrow}s).\lambda x.e$, et qui ont simultanément les deux types $t_1{\rightarrow}s_1$ et $t_2{\rightarrow}s_2$, c'est-à-dire telles que $t{\rightarrow}s \leq t_i{\rightarrow}s_i$ pour $i = 1..2$. Mais $t{\rightarrow}s \leq t_i{\rightarrow}s_i$ se décompose en $(t \simeq \mathbb{0}) \vee (t_i \leq t \wedge s \leq s_i)$. La condition est donc équivalente à $(t \simeq \mathbb{0}) \vee (t_1{\vee}t_2 \leq t \wedge s \leq s_1{\wedge}s_2)$, ou encore à $t{\rightarrow}s \leq (t_1{\vee}t_2){\rightarrow}(s_1{\wedge}s_2)$. Ce raisonnement montre que dans le calcul restreint, nous avons la propriété :

$$[\![(t_1{\rightarrow}s_1){\wedge}(t_2{\rightarrow}s_2)]\!]_{\mathcal{V}} = [\![(t_1{\vee}t_2){\rightarrow}(s_1{\wedge}s_2)]\!]_{\mathcal{V}}$$

Or l'on constate facilement que le sous-typage

$$(t_1{\rightarrow}s_1){\wedge}(t_2{\rightarrow}s_2) \leq (t_1{\vee}t_2){\rightarrow}(s_1{\wedge}s_2)$$

est faux en général.

Nous allons montrer dans le reste de cette section comment récupérer le Théorème 5.6. L'interprétation extensionnelle des fonctions comme des graphes (relations binaires non déterministes) dans la définition d'un modèle est cohérente avec la possibilité de spécifier plusieurs types flèche dans l'interface des abstractions. Pour retrouver le Théorème 5.6 dans le calcul restreint, il est nécessaire de changer cette interprétation. Au lieu d'interpréter les fonctions comme des graphes, nous allons coller de plus près au calcul. Nous modifions la définition d'une interprétation extensionnelle en posant :

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D) \times \mathcal{P}(D)$$

et

$$X \to Y := \{(X', Y') \mid X \subseteq X' \wedge Y' \subseteq Y\}$$

Intuitivement, on ne retient des fonctions que leur interface, donnée par un couple (X, Y), avec $X \subseteq D, Y \subseteq D$. L'ensemble X (resp. Y) représente le domaine (resp. le codomaine) de la fonction. L'interprétation extensionnelle du type $t \rightarrow s$ est l'ensemble des « fonctions » (X, Y) avec $[\![t]\!] \subseteq X$ et $Y \subseteq [\![s]\!]$, c'est-à-dire celles dont l'interface donne une contrainte au moins aussi forte que le type $t \rightarrow s$. Pour étudier la relation de sous-typage que l'on obtient, il faut adapter le Lemme 4.9. On s'appuie sur deux constatations simples. Tout d'abord, on observe que :

$$\bigcap_{i \in P} X_i \to Y_i = (\bigcup_{i \in P} X_i) \to (\bigcap_{i \in P} Y_i)$$

Cela permet de traiter les intersections. On étudie ensuite l'assertion

$$X \to Y \subseteq \bigcup_{i \in N} X_i \to Y_i$$

Dans le membre gauche, on trouve l'élément (X, Y), qui doit donc se trouver également, si l'assertion est vraie, dans un des $X_i \to Y_i$, ce qui signifie $X_i \subseteq X \wedge Y \subseteq Y_i$. Réciproquement, si $(X, Y) \in X_i \to Y_i$ pour un certain $i \in N$, alors l'assertion est vraie. On en déduit l'analogue du Lemme 4.9 :

Lemme 5.39 *Soient $(X_i)_{i \in P}, (X_i)_{i \in N}, (Y_i)_{i \in P}, (Y_i)_{i \in N}$ quatre familles de parties d'un ensemble D. Alors :*

$$\bigcap_{i \in P} X_i \to Y_i \subseteq \bigcup_{i \in N} X_i \to Y_i$$
$$\Longleftrightarrow$$
$$\exists i_0 \in N.\ X_{i_0} \subseteq \bigcup_{i \in N} X_i \wedge \bigcap_{i \in P} Y_i \subseteq Y_{i_0}$$

On adapte facilement le reste de la théorie. Par exemple, la condition $C_{\mathbf{fun}}$ dans la Définition 4.11 devient :

$$C_{\mathbf{fun}} ::= \exists \theta'_1 \rightarrow \theta'_2 \in N.\ \begin{cases} \tau(\theta'_1) \backslash \left(\bigvee_{\theta_1 \times \theta_2 \in P} \tau(\theta_1) \right) \in \mathcal{S} \\ \left(\bigwedge_{\theta_1 \times \theta_2 \in P} \tau(\theta_2) \right) \backslash \tau(\theta'_2) \in \mathcal{S} \end{cases}$$

Pour construire un modèle universel, on se ramène toujours à des ensembles finis pour représenter les fonctions. On définit $X \to_f Y$ comme le sous-ensemble de $X \to Y$ constitué des couples (X', Y') avec Y' fini et X' *cofini*.

Le Lemme 5.1, qui exprime la disjonction forte des types flèche (une intersection est incluse dans une réunion si et seulement si elle est incluse dans un des termes de la réunion) est toujours valide.

On adapte le typage de l'application, en reprenant la démarche de la Section 4.8. Pour un couple (f, x) avec $f = (X, Y) \in \mathbb{E}_{\mathbf{fun}} D = \mathcal{P}(D) \times \mathcal{P}(D)$ et $x \in D$, on prend $(f, x) \triangleright y$ si $(x \in X \Rightarrow y \in Y)$. Cela permet de faire marcher le Théorème 4.42. Le Corollaire 4.45 devient simplement :

$$\mathbf{app} : t_1 \mathbf{\times} t_2 \mathbf{\to} s \iff t_1 \leq t_2 \mathbf{\to} s$$

Remarque 5.40 *Notre nouvelle définition de $X \to Y$ ignore le cas particulier $X = \emptyset$ (qui donne naissance à la condition $t_2 \simeq \mathbb{0}$ dans le Corollaire 4.45). Nous aurions pu définir $X \to Y$ comme l'ensemble des couples (X', Y') avec $(X \subseteq X' \wedge Y \subseteq Y') \vee X = \emptyset$, pour garder ce cas particulier.*

Pour étudier le sous-typage $t_1 \leq t_2 \mathbf{\to} s$, nous pouvons mettre en place l'analogue du Lemme 4.37. En plus du domaine, nous pouvons définir le codomaine d'un type fonctionnel.

Lemme 5.41 *Soit t un type tel que $t \leq \mathbb{0} \mathbf{\to} \mathbb{1}$. Alors il existe deux types $Dom(t)$ et $Codom(t)$ tels que :*

$$\forall t_1, t_2.\ (t \leq t_1 \mathbf{\to} t_2) \iff \begin{cases} t_1 \leq Dom(t) \\ Codom(t) \leq t_2 \end{cases}$$

Ce lemme permet de déduire une fonction de typage pour l'opérateur **app** (analogue au Lemme 5.29, avec une preuve plus simple).

Et, évidemment, nous obtenons comme attendu une version du Théorème 5.6 pour le calcul restreint. La seule modification à apporter se trouve dans le cas $d' \in \mathbb{E}_{\mathbf{fun}} D$ de la preuve du Lemme 5.11. On dispose d'un type

$$t' = \bigwedge_{t_1 \mathbf{\to} t_2 \in P} t_1 \mathbf{\to} t_2 \wedge \bigwedge_{t_1 \mathbf{\to} t_2 \in N} \neg(t_1 \mathbf{\to} t_2)$$

avec $t' \leq t$ et $t' \not\simeq \mathbb{0}$. On remarque alors que :

$$t' \simeq \left(\left(\bigvee_{t_1 \mathbf{\to} t_2 \in P} t_1 \right) \mathbf{\to} \left(\bigwedge_{t_1 \mathbf{\to} t_2 \in P} t_2 \right) \right) \wedge \bigwedge_{t_1 \mathbf{\to} t_2 \in N} \neg(t_1 \mathbf{\to} t_2)$$

On prend alors une abstraction close et bien typée dont l'unique type flèche de l'interface est $(\bigvee_{t_1 \to t_2 \in P} t_1) \mathbf{\to} (\bigwedge_{t_1 \to t_2 \in P} t_2)$, par exemple $\mu f(\ldots).\lambda x.\ f\ x$. La règle (*abstr*) donne à cette valeur le type t'.

Chapitre 6

Filtrage

Nous allons étendre le calcul introduit au chapitre précédent avec une opération de filtrage par motif (*pattern matching*). Un motif permet d'extraire et de nommer une ou plusieurs sous-valeurs d'une valeur en entrée. Les motifs sont, au même titre que les types, des objets récursifs, ce qui permet d'exprimer des extractions à profondeur non bornée.

6.1 Motifs

Nous allons utiliser le formalisme du Chapitre 2 pour définir l'algèbre des motifs. Nous devons donc définir un endofoncteur ensembliste G. Pour un ensemble Y, nous définissons GY comme l'ensemble des termes engendrés par les productions ci-dessous :

$$\begin{array}{rcll} p \in GY & ::= & x & x \text{ variable} \\ & | & t & t \in \widehat{T} \\ & | & (q_1, q_2) & q_1, q_2 \in Y \\ & | & p_1 | p_2 & p_1, p_2 \in GY \\ & | & p_1 \& p_2 & p_1, p_2 \in GY \\ & | & (x := c) & x \text{ variable}, c \in \mathcal{C} \end{array}$$

Nous choisissons une G-coalgèbre récursive et régulière $\mathbb{P} = (P, \sigma)$. Les éléments de P (resp. $\widehat{P} = GP$) sont appelés nœuds de motif (resp. expressions de motif, ou simplement motifs).

Les productions dans la définition du foncteur G introduisent des motifs appelés respectivement variable, test de type (ou simplement type), produit, alternative (ou réunion), conjonction (ou intersection), et constante. Notons que les composantes des motifs produit sont des nœuds de motif et non pas des motifs, et qu'il s'agit de la seule utilisation possible des nœuds de motif pour former des motifs : cela permet d'introduire des motifs récursifs, tout en garantissant que les cycles de récursion sont bien formés (c'est-à-dire qu'ils croisent un constructeur de motif produit).

Definition 6.1 *Un ensemble S de motifs est* **stable** *si :*
- $\forall (q_1, q_2) \in S.\ \sigma(q_1) \in S, \sigma(q_2) \in S$
- $\forall p_1 | p_2 \in S.\ p_1 \in S, p_2 \in S$

– $\forall p_1 \& p_2 \in S.\ p_1 \in S, p_2 \in S$

Definition 6.2 *Soient p_1 et p_2 deux motifs. On dit que p_2 est* **accessible depuis** *p_1 si p_2 est dans tout ensemble de motifs stable qui contient p_1.*

Lemme 6.3 *Soit p un motif. L'ensemble des motifs accessibles depuis p est stable et fini.*

| *Preuve*: Conséquence de la régularité de la coalgèbre $\mathbb{P}$. □

Remarque 6.4 *La notion d'ensemble stable et fini joue, pour les motifs, un rôle similaire à celle de socle pour les types, à savoir borner par ensemble fini l'ensemble des objets potentiellement manipulés par les algorithmes, ce qui assure leur terminaison.*

Definition 6.5 *Soit p un motif. On définit l'***ensemble des variables** *de p, noté $Var(p)$, par :*

$$Var(p) = \{x \mid x \text{ accessible depuis } p\} \cup \{x \mid (x := c) \text{ accessible depuis } p\}$$

Definition 6.6 *Un motif p est* **bien formé** *si :*
– *pour tout motif $p_1|p_2$ accessible depuis p : $Var(p_1) = Var(p_2)$;*
– *pour tout motif $p_1 \& p_2$ accessible depuis p : $Var(p_1) \cap Var(p_2) = \emptyset$.*

Notons qu'aucune condition de bonne formation ne contraint les ensembles de variables $\mathrm{Var}(\sigma(q_1))$ et $\mathrm{Var}(\sigma(q_2))$ dans un motif (q_1, q_2).

6.2 Sémantique

Pour définir la sémantique des motifs, nous nous plaçons dans un modèle *structurel* $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ (Définition 4.4). Pour un élément d de D et un motif p, nous notons d/p le résultat de p appliqué à d ; c'est soit une fonction γ de $\mathrm{Var}(p)$ dans D, soit Ω (qui représente l'échec du filtrage de d par p). Nous utilisons la lettre r pour désigner un tel résultat. La Figure 6.1 donne une définition de d/p, par induction sur le couple (d, p) ordonné lexicographiquement.

La même variable de capture x peut apparaître dans les deux nœuds de motif d'un motif produit (q_1, q_2). Si ce motif réussit sur un élément (d_1, d_2), nous obtenons un résultat d'_1 pour x par q_1 appliqué à d_1 et un autre résultat d'_2 pour x par q_2 appliqué à d_2. D'après la définition de l'opérateur $\oplus$, la sémantique est alors de construire le résultat (d'_1, d'_2) pour x. Cette sémantique nous permettra d'exprimer la capture de sous-séquences pour un encodage des séquences sous forme de couples imbriqués (Section 10.3).

6.3 Typage

Pour typer un motif bien formé p, nous devons calculer d'une part le type accepté par p (qui représente l'ensemble des valeurs qui ne font pas échouer p), et d'autre part le type du résultat par p pour un type d'entrée t (un environnement qui associe à chaque variable de capture $x \in \mathrm{Var}(p)$ un type qui représente l'ensemble des valeurs que l'on peut obtenir en appliquant p à une valeur de t).

$$
\begin{array}{lcl}
d/x & = & \{x \mapsto d\} \\
d/t & = & \begin{cases} \{\} & \text{si } d \in [\![t]\!] \\ \Omega & \text{si } d \in [\![\neg t]\!] \end{cases} \\
d/(q_1, q_2) & = & \begin{cases} d_1/\sigma(q_1) \oplus d_2/\sigma(q_2) & \text{si } d = (d_1, d_2) \\ \Omega & \text{sinon} \end{cases} \\
d/p_1|p_2 & = & d/p_1 \mid d/p_2 \\
d/p_1\&p_2 & = & d/p_1 \oplus d/p_2 \\
d/(x := c) & = & \{x \mapsto c\}
\end{array}
$$

où :

$$
\begin{array}{lcl}
\gamma|r & = & \gamma \\
\Omega|r & = & r \\
\Omega \oplus r & = & \Omega \\
r \oplus \Omega & = & \Omega \\
\gamma_1 \oplus \gamma_2 & = & \{x \mapsto \gamma_1(x) \mid x \in \mathrm{Dom}(\gamma_1) \backslash \mathrm{Dom}(\gamma_2)\} \\
 & \cup & \{x \mapsto \gamma_2(x) \mid x \in \mathrm{Dom}(\gamma_2) \backslash \mathrm{Dom}(\gamma_1)\} \\
 & \cup & \{x \mapsto (\gamma_1(x), \gamma_2(x)) \mid x \in \mathrm{Dom}(\gamma_1) \cap \mathrm{Dom}(\gamma_2)\}
\end{array}
$$

FIG. 6.1 – Sémantique du filtrage

Definition 6.7 *Pour un motif p, on note* $(\!|p|\!)$ *l'ensemble des éléments du modèle acceptés par p :*

$$(\!|p|\!) = \{d \in D \mid v/p \neq \Omega\}$$

Lemme 6.8 *On a les égalités suivantes :*

$$
\begin{array}{lcl}
(\!|x|\!) & = & D \\
(\!|t|\!) & = & [\![t]\!] \\
(\!|(q_1, q_2)|\!) & = & (\!|\sigma(q_1)|\!) \times (\!|\sigma(q_2)|\!) \\
(\!|p_1|p_2|\!) & = & (\!|p_1|\!) \cup (\!|p_2|\!) \\
(\!|p_1\&p_2|\!) & = & (\!|p_1|\!) \cap (\!|p_2|\!) \\
(\!|(x := c)|\!) & = & D
\end{array}
$$

Preuve: C'est immédiat en utilisant la définition de la Figure 6.1.
□

Theorème 6.9 *Pour tout motif p, on peut calculer un type* $\wr p \int$ *indépendant du modèle, tel que :*

$$[\![\wr p \int]\!] = (\!|p|\!)$$

Preuve: Nous allons en fait construire une fonction $\wr \_ \int$ des motifs dans les types, telle que :

$$
\begin{array}{lcl}
\wr x \int & = & \mathbb{1} \\
\wr t \int & = & t \\
\wr (q_1, q_2) \int & = & \wr \sigma(q_1) \int \times \wr\, \sigma(q_2) \int \\
\wr p_1|p_2 \int & = & \wr p_1 \int \vee \wr\, p_2 \int \\
\wr p_1\&p_2 \int & = & \wr p_1 \int \wedge \wr\, p_2 \int \\
\wr (x := c) \int & = & \mathbb{1}
\end{array}
$$

Si $\wr\_\int$ vérifie ces équations, alors $[\![\wr\_\int]\!]$ vérifie trivialement les mêmes équations que $(\!|\_|\!)$ dans le lemme ci-dessus. Une induction sur le couple (d, p) (ordonné lexicographiquement) montre alors que $d \in [\![\wr p \int]\!] \iff d \in (\!|p|\!)$ pour tout $d \in D$ et tout motif p.

Pour construire la fonction $\wr\_\int$, nous utilisons la technique développée à la Section 2.4.3. Il s'agit de définir un transfert entre la signature G des motifs, et la signature $\mathcal{B} \circ F$ des types. Nous avons vu qu'il suffit pour cela de définir une transformation naturelle $\beta : G \to \mathcal{B} \circ F(T + \_)$. Pour un ensemble Y, et $p \in GY$, nous définissons $\beta_Y(p)$ par induction sur p :

$$\begin{array}{lcl}
\beta_Y(x) & = & \mathbb{1} \\
\beta_Y(t) & = & t \\
\beta_Y((q_1, q_2)) & = & q_1 \times q_2 \\
\beta_Y(p_1 | p_2) & = & \beta_Y(p_1) \vee \beta_Y(p_2) \\
\beta_Y(p_1 \& p_2) & = & \beta_Y(p_1) \wedge \beta_Y(p_2) \\
\beta_Y((x := c)) & = & \mathbb{1}
\end{array}$$

Les membres droit de ces équations sont bien dans $\mathcal{B} \circ F(T + Y)$ (en utilisant l'inclusion implicite $@ : F(Y) \to \mathcal{B} \circ F(Y)$). On peut alors appliquer le Théorème 2.35, et l'on obtient une fonction $\wr\_\int$ qui vérifie les équations attendues. □

Definition 6.10 *Soit p un motif bien formé, t un type tel que $t \leq \wr p \int$, et x une variable de p. On pose :*

$$(t /\!\!/ p)(x) := \{(d/p)(x) \mid d \in [\![t]\!]\}$$

Lemme 6.11 *Dans les équations ci-dessous, lorsque le membre gauche est bien défini, alors le membre droit l'est aussi, et l'égalité est vraie :*

$$\begin{array}{lcl}
(t /\!\!/ x)(x) & = & [\![t]\!] \\
(t /\!\!/ (q_1, q_2))(x) & = & \begin{cases}
(\pi_1[t] /\!\!/ \sigma(q_1))(x) & si\ x \in Var(\sigma(q_1)) \setminus Var(\sigma(q_2)) \\
(\pi_2[t] /\!\!/ \sigma(q_2))(x) & si\ x \in Var(\sigma(q_2)) \setminus Var(\sigma(q_1)) \\
\displaystyle\bigcup_{(t_1,t_2)\in\pi(t)} (t_1 /\!\!/ \sigma(q_1))(x) \times (t_2 /\!\!/ \sigma(q_2))(x) & \\
\qquad si\ x \in Var(\sigma(q_2)) \cap Var(\sigma(q_1))
\end{cases} \\
(t /\!\!/ (p_1 | p_2))(x) & = & ((t \wedge \wr\, p_1 \int) /\!\!/ p_1)(x) \cup ((t \setminus \wr\, p_1 \int) /\!\!/ p_2)(x) \\
(t /\!\!/ (p_1 \& p_2))(x) & = & \begin{cases}
(t /\!\!/ p_1)(x) & si\ x \in Var(p_1) \\
(t /\!\!/ p_2)(x) & si\ x \in Var(p_2)
\end{cases} \\
(t /\!\!/ (x := c))(x) & = & \begin{cases}
\{c\} & si\ t \not\simeq \mathbb{0} \\
\mathbb{0} & si\ t \simeq \mathbb{0}
\end{cases}
\end{array}$$

Rappelons que l'opérateur $\pi_1[\_]$ sur les types a été défini par :

$$\pi_1[t] = \bigvee_{(t_1,t_2)\in\pi(t)} t_1$$

L'opérateur $\pi_2[\_]$ est défini de manière analogue.

Theorème 6.12 *Soit p un motif bien formé et t un type tel que $t \leq \lceil p \rfloor$. Alors on peut calculer un environnement de typage $(t/p) : Var(p) \to \widehat{T}$, qui ne dépend que de la classe d'équivalence du modèle (c'est-à-dire de la relation de sous-typage induite) tel que :*

$$\forall x \in Var(p).\ [\![(t/p)(x)]\!] = (t /\!\!/ p)(x)$$

Preuve: Fixons x. Soit S l'ensemble (fini) des motifs accessibles depuis p et dont x est une variable. Soit $\sqsupseteq$ un socle qui contient t, ainsi que tous les $\lceil p' \rfloor$ pour $p' \in S$. Posons :

$$V = \{[t', p'] \mid t' \in \sqsupseteq, p' \in S, t' \leq \lceil p' \rfloor\}$$

(nous utilisons $[t', p']$ pour désigner simplement le couple (t', p') tout en évitant de surcharger les équations avec des parenthèses toutes semblables). Définissons un système $\vee\times$ (Définition 4.47), c'est-à-dire une fonction $\phi : V \to \mathcal{P}_f(V + V \times V + \widehat{T})$:

$$\begin{array}{lcl}
\phi([t', x]) & = & \{t'\} \\
\phi([t', (q_1, q_2)]) & = & \left\{ \begin{array}{l}
\{[\pi_1[t'], \sigma(q_1)]\} \\
\qquad \text{si } x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2)) \\
\{[\pi_2[t'], \sigma(q_2)]\} \\
\qquad \text{si } x \in \mathrm{Var}(\sigma(q_2)) \backslash \mathrm{Var}(\sigma(q_1)) \\
\{([t_1, \sigma(q_1)], [t_2, \sigma(q_2)]) \mid (t_1, t_2) \in \pi(t')\} \\
\qquad \text{si } x \in \mathrm{Var}(\sigma(q_2)) \cap \mathrm{Var}(\sigma(q_1))
\end{array} \right. \\
\phi([t', p_1 | p_2]) & = & \{[t' \wedge \lceil p_1 \rfloor, p_1]; [t' \setminus \lceil p_1 \rfloor, p_2]\} \\
\phi([t', p_1 \& p_2]) & = & \left\{ \begin{array}{ll}
\{[t', p_1]\} & \text{si } x \in \mathrm{Var}(p_1) \\
\{[t', p_2]\} & \text{si } x \in \mathrm{Var}(p_2)
\end{array} \right. \\
\phi([t', (x := c)]) & = & \left\{ \begin{array}{ll}
\{b_c\} & \text{si } t' \not\simeq \mathbb{0} \\
\emptyset & \text{si } t' \simeq \mathbb{0}
\end{array} \right.
\end{array}$$

On voit que ce système ne dépend que de la classe d'équivalence du modèle (à cause de la définition de V, et aussi des opérateurs π, π_1, π_2, il n'est pas complètement indépendant du modèle).

D'après le lemme précédent, si l'on pose $\rho([t', p']) = (t' /\!\!/ p')(x)$, alors ρ est un point fixe de l'opérateur $\widehat{\phi}$ (Définition 4.48). Nous allons montrer qu'il s'agit de son plus petit point fixe. Il s'agit de montrer que pour tout point fixe ρ_0, on a :

$$\forall [t', p'] \in V.\ (t' /\!\!/ p')(x) \subseteq \rho_0([t', p'])$$

Cela s'écrit :

$$\forall [t', p'] \in V.\ \forall d \in [\![t']\!].\ (d/p')(x) \in \rho_0([t', p'])$$

Cette propriété se prouve par induction sur le couple (d, p') (simultanément pour tous les t'), en distinguant suivant la forme de p'. Considérons par exemple le cas $p' = (q_1, q_2)$, avec $x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2))$. Puisque $d \in [\![t']\!]$ et $t' \leq \lceil p' \rfloor \simeq \lceil \sigma(q_1) \rfloor \times \lceil \sigma(q_2) \rfloor$, l'élément d est forcément un couple (d_1, d_2). On a :

$$(d/p')(x) = (d_1/\sigma(q_1))(x)$$

et

$$\rho_0([t', p']) = \widehat{\phi}(\rho_0)([t', p']) = \rho_0([\pi_1[t'], \sigma(q_1)])$$

Or $d_1 \in [\![\pi_1[t']]\!]$, et d_1 est strictement plus petit que d, donc on peut appliquer l'hypothèse d'induction au couple $(d_1, \sigma(q_1))$; elle donne $(d_1/\sigma(q_1))(x) \in \rho_0([\pi_1[t'], \sigma(q_1)])$, ce qui permet de conclure. Tous les autres cas se traitent de manière similaire, en utilisant la sémantique du filtrage et le fait que ρ_0 est un point fixe de $\widehat{\phi}$.

Le Théorème 4.49 permet de construire des types qui représentent le plus petit point fixe du système donné par ϕ. On note $(t'/p')(x)$ ces types, et ils vérifient la propriété donnée dans l'énoncé du théorème. □

Remarque 6.13 *En pratique, le système d'équations* $\vee\times$ *introduit dans la preuve du théorème est construit de manière paresseuse : seuls les couples* $[t', p']$ *effectivement accessibles sont considérés. On peut également utiliser des simplifications justifiées par les définitions ensemblistes, par exemple en prenant* $\emptyset$ *pour* $\phi([t', p'])$ *lorsque* $t' \simeq \mathbb{0}$ *(car alors* $(t'/\!\!/p')(x) = \emptyset$*).*

Lemme 6.14 *Soit* p *un motif bien formé et* x *une variable de* p*. Alors la fonction* $t \mapsto (t/p)(x)$ *est croissante et commute avec l'opérateur* $\vee$ *:*

$$\forall t_1, t_2 \leq \lfloor p \rfloor . \ (t_1 \vee t_2/p)(x) \simeq (t_1/p)(x) \vee (t_2/p)(x)$$

Preuve: C'est immédiat, compte-tenu de la définition sémantique de $(t/\!\!/p)(x)$. □

6.4 Extension du calcul

Nous pouvons maintenant étendre le calcul introduit au Chapitre 5 avec une opération de filtrage par motifs.

Nous ajoutons une construction dans la syntaxe des expressions :

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \end{array}$$

où p_1 et p_2 désignent des motifs bien formés. Les variables de p_i sont considérées comme étant liées dans e_i. On définit donc $\mathrm{fv}(\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2)$ comme

$$\mathrm{fv}(e) \cup (\mathrm{fv}(e_1) \backslash \mathrm{Var}(p_1)) \cup (\mathrm{fv}(e_2) \backslash \mathrm{Var}(p_2))$$

Intuitivement, cette expression se comporte de la manière suivante : l'expression e est évaluée en une valeur v. Si le motif p_1 accepte cette valeur, l'expression e_1 est alors évaluée, dans l'environnement résultant du filtrage de v par p_1. Sinon, l'expression e_2 est évaluée, dans l'environnement résultant du filtrage de v par p_2, qui doit nécessairement réussir. Ce sera le rôle du système de types de garantir cette propriété (exhaustivité du filtrage), et aussi de calculer le type pour les variables liées par le motif p_i dans e_i.

Cette nouvelle construction est en fait une généralisation du test de type dynamique. En effet, on peut encoder l'expression $(x = e \in t \ ?\ e_1 | e_2)$ en $\texttt{match}\ e\ \texttt{with}\ (x\&t) \to e_1 \mid (x\&\neg t) \to e_2$.

Typage Nous prenons comme modèle d'amorce le modèle structurel utilisé jusqu'ici dans ce chapitre.

La règle de typage associée à la nouvelle construction est :

$$\frac{\left\{\begin{array}{l}\Gamma \vdash e : t_0\\ t_0 \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor\\ t_1 = t_0 \wedge \lfloor p_1 \rfloor\\ t_2 = t_0 \setminus \lfloor p_1 \rfloor\end{array}\right. \quad \forall i = 1..2.\ \left\{\begin{array}{ll} s_i = \mathbb{0} & \text{si } t_i \simeq \mathbb{0}\\ (t_i/p_i), \Gamma \vdash e_i : s_i & \text{si } t_i \not\simeq \mathbb{0}\end{array}\right.}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 : s_1 \vee s_2}\ (match)$$

Toutes les propriétés syntaxiques du système de types s'étendent sans problème, en particulier le Théorème 5.6.

Sémantique On dispose du modèle structurel des valeurs $[\![\_]\!]_{\mathcal{V}} : \widehat{T} \to \mathcal{P}(\mathcal{V})$. Pour toute valeur v et tout motif bien formé p, on peut calculer v/p, qui est soit Ω, soit une fonction de $\mathrm{Var}(p)$ dans $\mathcal{V}$ (que l'on voit comme une substitution).

Cela permet de définir la sémantique opérationnelle du filtrage. On étend les contextes d'évaluation pour autoriser à réduire l'expression filtrée :

$$\begin{array}{lcl} C[] & ::= & \ldots\\ & \mid & \texttt{match}\ []\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2\end{array}$$

Et l'on ajoute les règles de réduction ci-dessous :

$$\frac{v/p_1 \neq \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow e_1[v/p_1]}$$

$$\frac{v/p_1 = \Omega \quad v/p_2 \neq \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow e_2[v/p_2]}$$

$$\frac{v/p_1 = \Omega \quad v/p_2 = \Omega}{\texttt{match}\ v\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \rightsquigarrow \Omega}$$

Sûreté Le modèle des valeurs étant équivalent au modèle d'amorce, nous pouvons donner une interprétation des opérateurs de typage des motifs en termes de valeurs.

Lemme 6.15 *Soit p un motif bien formé et v une valeur. Alors :*

$$v/p \neq \Omega \iff \vdash v : \lfloor p \rfloor$$

Lemme 6.16 *Soit p un motif bien formé, t un type tel que $t \leq \lfloor p \rfloor$, x une variable de p, et v une valeur. Alors :*

$$(\exists v'.\ (\vdash v' : t) \wedge ((v'/p)(x) = v)) \iff \vdash v : (t/p)(x)$$

On en déduit facilement un théorème de préservation du typage par réduction.

Remarque 6.17 *À ce point, on pourrait se demander pourquoi nous n'avons pas directement choisi de travailler dans le modèle des valeurs pour définir les opérateurs de typage des motifs, ce qui aurait eu pour effet de remplacer les*

deux lemmes ci-dessus par des définitions. La raison est simple : la définition du modèle des valeurs suppose d'avoir défini le système de types, et celui-ci utilise ces opérateurs de typage. On peut noter également que le modèle des valeurs du Chapitre 5 n'est pas identique à celui du calcul étudié dans ce chapitre, car les valeurs ne sont pas les mêmes (dans le corps d'une abstraction, une valeur peut utiliser la nouvelle construction de filtrage).

Inférence de types L'approche de la Section 5.7 s'étend sans problème pour traiter la nouvelle construction.

Lemme 6.18 *Pour tout motif p et tout schéma $\mathbb{t}$ tel que $\mathbb{t} \leq \lfloor p \rfloor$, et toute variable x de p, on peut calculer un schéma $((\mathbb{t}/p))(x)$ tel que :*

$$\forall s.\ (((\mathbb{t}/p))(x) \leq s \iff \exists t.\ (\mathbb{t} \leq t) \wedge ((t/p))(x) \leq s)$$

Preuve: On définit $(\mathbb{t}/p)(x)$ pour tout motif $\mathbb{t}$ tel que $\mathbb{t} \leq \lfloor p \rfloor$ par induction sur le triplet $(\#\mathbb{t}, p, \mathbb{t})$ ordonné lexicographiquement, où $\#\mathbb{t}$ représente le nombre maximal de constructeurs $\otimes$ imbriqués dans $\mathbb{t}$. Souvenons nous que d'après le Lemme 5.26, nous avons $\#(t_0 \vee \mathbb{t}) \leq \#\mathbb{t}$. Nous nous laissons ensuite guider par le Lemme 6.11. Les équations ci-dessous traitent tous les cas, et elles sont bien fondées pour l'induction sur le triplet $(\#\mathbb{t}, p, \mathbb{t})$.

$$
\begin{array}{lcl}
(\mathbb{t}/x)(x) & = & \mathbb{t} \\
(t/(q_1,q_2))(x) & = & (t/(q_1,q_2))(x) \text{ (vu comme un schéma)} \\
((\mathbb{t}_1 \vee \mathbb{t}_2)/(q_1,q_2))(x) & = & (\mathbb{t}_1/(q_1,q_2))(x) \vee (\mathbb{t}_2/(q_1,q_2))(x) \\
((\mathbb{t}_1 \otimes \mathbb{t}_2)/(q_1,q_2))(x) & = & \left\{ \begin{array}{l} (\mathbb{t}_1/\sigma(q_1))(x) \\ \quad \text{si } x \in \mathrm{Var}(\sigma(q_1)) \backslash \mathrm{Var}(\sigma(q_2)) \\ (\mathbb{t}_2/\sigma(q_2))(x) \\ \quad \text{si } x \in \mathrm{Var}(\sigma(q_2)) \backslash \mathrm{Var}(\sigma(q_1)) \\ (\mathbb{t}_1/\sigma(q_1))(x) \otimes (\mathbb{t}_2/\sigma(q_2))(x) \\ \quad \text{si } x \in \mathrm{Var}(\sigma(q_1)) \cap \mathrm{Var}(\sigma(q_2)) \end{array} \right. \\
(\mathbb{t}/p_1|p_2)(x) & = & ((\lfloor p_1 \rfloor \vee \mathbb{t})/p_1)(x) \vee \\
 & & \quad (((\neg \lfloor p_1 \rfloor) \vee \mathbb{t})/p_2)(x) \\
(\mathbb{t}/p_1 \& p_2)(x) & = & \left\{ \begin{array}{ll} (\mathbb{t}/p_1)(x) & \text{si } x \in \mathrm{Var}(p_1) \\ (\mathbb{t}/p_2)(x) & \text{si } x \in \mathrm{Var}(p_2) \end{array} \right. \\
(\mathbb{t}/(x := c))(x) & = & \left\{ \begin{array}{ll} b_c & \text{si } \mathbb{t} \not\leq \mathbb{0} \\ \mathbb{0} & \text{si } \mathbb{t} \leq \mathbb{0} \end{array} \right.
\end{array}
$$

On établit alors facilement que chaque cas de cette définition inductive préserve la propriété de l'énoncé. Considérons par exemple le cas $p = p_1|p_2$. Soit s un type. On a :

$$
\begin{array}{ll}
 & (\mathbb{t}/p_1|p_2)(x) \leq s \\
\iff & \left\{ \begin{array}{l} ((\lfloor p_1 \rfloor \vee \mathbb{t})/p_1)(x) \leq s \\ (((\neg \lfloor p_1 \rfloor) \vee \mathbb{t})/p_2)(x) \leq s \end{array} \right. \\
\iff & \left\{ \begin{array}{l} \exists t_1 \geq \lfloor p_1 \rfloor \vee \mathbb{t}.\ (t_1/p_1)(x) \leq s \\ \exists t_2 \geq (\neg \lfloor p_1 \rfloor) \vee \mathbb{t}.\ (t_2/p_2)(x) \leq s \end{array} \right. \\
\iff & \exists t_1, t_2 \geq \mathbb{t}.\ \left\{ \begin{array}{l} ((\lfloor p_1 \rfloor \wedge t_1)/p_1)(x) \leq s \\ (((\neg \lfloor p_1 \rfloor) \wedge t_2)/p_2)(x) \leq s \end{array} \right. \\
\iff & \exists t \geq \mathbb{t}.\ (t/p_1|p_2)(x) \leq s
\end{array}
$$

Pour la dernière équivalence, on prend $t = t_1 \wedge t_2$ pour l'implication $\Rightarrow$ et $t_1 = t_2 = t$ pour l'implication $\Leftarrow$. Les autres cas se traitent d'une manière similaire. □

Nous pouvons alors définir le cas manquant dans la définition de l'algorithme de typage (Figure 5.3) :

$$\mathbb{\Gamma}[\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2] := \mathbb{s}_1 \vee \mathbb{s}_2$$

$$\text{où} \begin{cases} \mathbb{t}_0 := \mathbb{\Gamma}[e] \\ \mathbb{t}_1 := \lfloor p_1 \rfloor \wedge \neg \mathbb{t}_0 \\ \mathbb{t}_2 := (\neg \lfloor p_1 \rfloor) \wedge \mathbb{t}_0 \\ \mathbb{s}_i := \begin{cases} ((\mathbb{t}_i / p_i), \mathbb{\Gamma})[e_i] & \text{si } \mathbb{t}_i \not\leq \mathbb{0}, \mathbb{t}_i \leq \lfloor p_i \rfloor \\ \mathbb{0} & \text{si } \mathbb{t}_i \leq \mathbb{0} \\ \Omega & \text{si } \mathbb{t}_i \not\leq \lfloor p_i \rfloor \end{cases} & (i = 1..2) \end{cases}$$

L'analogue du Théorème 5.33 se prouve facilement.

Filtrage à n branches Nous avons choisi d'introduire un filtrage à deux branches pour simplifier les notations. Il n'y a aucun problème à considérer un filtrage à n branches (avec $n \geq 1$). En fait, nous pouvons encoder le filtrage à n branches dans celui à deux branches sans perdre de précision en terme de typage. Pour $n = 1$, nous posons :

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 := \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_1 \to e_1$$

Et pour $n \geq 3$, nous posons :

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots := \texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid x \to \texttt{match}\ x\ \texttt{with}\ \ldots$$

(x est une variable fraîche)

6.5 Normalisation

Dans cette section, nous montrons comment l'approche catégorique utilisée pour définir la G-coalgèbre $\mathbb{P} = (P, \sigma)$ permet d'exprimer des transformations de motifs qui préservent leur sémantique. Dans cette section, nous ne considérons que des motifs bien formés.

Definition 6.19 *Si p et p' sont deux motifs, on note $p \simeq p'$ si $Var(p) = Var(p')$ et $v/p = v/p'$ pour toute valeur v. Si q et q' sont deux nœuds de motifs, on note $q \simeq q'$ pour $\sigma(q) \simeq \sigma(q')$.*

6.5.1 Méthode

Nous considérons une spécification de normalisation donnée par une restriction de G, c'est-à-dire un endofoncteur ensembliste G' tel que, pour tout ensemble X, on a $G'X \subseteq GX$ et pour toute fonction $f : X \to Y$, $G'f$ est la restriction de Gf. On suppose également que si $X \subseteq Y$, alors $GX \cap G'Y \subseteq G'X$. Intuitivement, le foncteur G' impose certaines contraintes sur la structure des motifs. Évidemment, ces contraintes doivent s'appliquer récursivement aux $\sigma(q)$ pour les nœuds de motifs q qui apparaissent dans les motifs considérés.

Pour une G-coalgèbre $\mathbb{P}' = (P', \sigma')$, nous appelons G'-base une partie finie $B \subseteq P'$ telle que $\sigma'(B) \subseteq G'B$ (c'est *a fortiori* une base). Les motifs qui sont éléments de $G'B$ pour une certaine G'-base de $\mathbb{P}$ sont appelés G-motifs. Nous cherchons à calculer pour tout motif p un G'-motif $\phi(p)$ tel que $p \simeq \phi(p)$. Cette équivalence se prouve de manière coinductive par rapport aux motifs, ce qui rend une preuve directe assez pénible. Le formalisme que nous introduisons dans cette section permet de factoriser ce raisonnement coinductif. La section suivante donne un exemple d'application : on verra que la preuve se ramène alors à une simple induction qui utilise des propriétés algébriques de l'équivalence $\simeq$.

Voici comment procéder pour construire la fonction ϕ. On se donne une fonction de normalisation N qui associe à un motif p un nouveau motif $N(p)$ qui vérifie les contraintes de G', mais qui est construit sur des nœuds non pas de P, mais d'un autre ensemble P'. Formellement, N est une fonction $GP \to G'P'$. Elle effectue la transformation attendue au niveau supérieur. Elle sera en général définie par induction sur p. Pour pouvoir « revenir » dans P, on se donne également une fonction $f : P' \to P$, qui définit en quelque sorte la « sémantique » des éléments de P'. La correction de la transformation N s'exprime par :

$$\forall p \in \widehat{P}.\ Gf \circ N(p) \simeq p$$

et cette propriété se prouve par induction sur p.

On pose alors $\sigma' = N \circ \sigma \circ f$, ce qui définit une G-coalgèbre $\mathbb{P}' = (P', \sigma')$. Nous supposons que cette coalgèbre est régulière. Intuitivement, cela signifie que si l'on itère la fonction N, on ne rencontre qu'un nombre fini de nœuds différents dans P'.

Cette hypothèse de régularité garantit (Théorème 2.33) l'existence d'un morphisme de G-coalgèbres $\Phi : \mathbb{P}' \to \mathbb{P}$. Nous prenons alors $\phi = G\Phi \circ N : GP \to GP$.

Nous allons prouver que pour tout motif p, $\phi(p)$ est un G'-motif et $\phi(p) \simeq p$, comme attendu.

Voyons tout d'abord que $\phi(p)$ est un G'-motif. Toute base B de $\mathbb{P}'$ est une G'-base (car $\sigma'(B) \subseteq GB$ et $\sigma'(B) = N(\sigma \circ f(B)) \subseteq G'P'$, et $GB \cap G'P' \subseteq G'B$ d'après l'hypothèse sur G'), et elle est envoyée par le morphisme Φ sur une G'-base de $\mathbb{P}$ (car $\sigma(\Phi(B)) = G\Phi(\sigma'(B)) \subseteq G\Phi(G'B) \subseteq G'(\Phi(B))$). Ainsi, tous les $G\Phi(p')$ pour $p' \in G'P'$ sont des G'-motifs, et $\phi(p)$ est bien de cette forme (avec $p' = N(p)$).

Il reste à établir que $\phi(p) \simeq p$ pour tout motif p, c'est-à-dire :

$$\forall v. \forall p.\ v/(G\Phi \circ N(p)) = v/p$$

Nous allons prouver cette assertion par induction sur v. Soit donc v une valeur. Par hypothèse, on sait que $v/p = v/(Gf \circ N(p))$. Il suffit donc de prouver que pour tout $p' \in G'P'$: $v/G\Phi(p') = v/Gf(p')$. On raisonne alors par induction (locale) sur p'. La fonctorialité de G et la compositionalité de la sémantique du filtrage permettent de traiter trivialement tous les cas sauf celui d'un motif couple $p' = (q'_1, q'_2)$ avec $q'_i \in P'$. On a $G\Phi(p') = (\Phi(q'_1), \Phi(q'_2))$ et $Gf(p') = (f(q'_1), f(q'_2))$. Si la valeur v n'est pas un couple, les deux membres valent alors Ω. Si $v = (v_1, v_2)$, il suffit de voir que $v_i/\sigma(\Phi(q'_i)) = v_i/\sigma(f(q'_i))$ pour $i = 1, 2$. Or $\sigma(\Phi(q'_i)) = G\Phi \circ \sigma'(q'_i)$ (Φ est un morphisme de G-coalgèbres) et $\sigma'(q'_i) = N \circ \sigma \circ f(q'_i)$. Posons $p_i = \sigma \circ f(q'_i)$. Il s'agit ainsi de prouver que $v_i/(G\Phi \circ N(p_i)) = v_i/p_i$, ce que donne l'hypothèse d'induction globale, appliquée aux valeurs v_1 et v_2.

6.5.2 Élimination de l'intersection

Nous illustrons la formalisme introduit ci-dessus par l'exemple suivant : nous voulons mettre les motifs dans une forme normale, où les tests de type sont restreints à des types t avec $t\mathbf{\wedge}\mathbb{1}_{\mathbf{prod}} \simeq \mathbb{0}$ (où $\mathbb{1}_{\mathbf{prod}} = \mathbb{1}\mathbf{\times}\mathbb{1}$) et où les motifs intersection $\mathbf{\&}$ sont éliminés (sauf ceux nécessaires pour exprimer la capture).

Définissons l'endofoncteur ensembliste G' qui capture ces contraintes. On pose :

$$\begin{array}{lcll} p \in G'Y & ::= & x\mathbf{\&}p & \\ & | & (x := c)\mathbf{\&}p & \\ & | & t & t \leq \neg\mathbb{1}_{\mathbf{prod}} \\ & | & p_1 \mid p_2 & p_1, p_2 \in GY \\ & | & (q_1, q_2) & q_1, q_2 \in Y \end{array}$$

Ce foncteur vérifie les hypothèses de la section précédente. Pour l'ensemble P', nous prenons :

$$P' = \{(t, S) \mid t \in \widehat{T}, S \in \mathcal{P}_f(P)\}$$

Nous notons $\mathfrak{q}$ les éléments de P'. La fonction $f : P' \to P$ donne son sens intuitif aux éléments de P'. Pour un élément $\mathfrak{q} = (t, S)$ avec $S = \{q_1, \ldots, q_n\}$, nous posons $f(\mathfrak{q}) = q$ où q est choisi de sorte que $\sigma(q) = t\mathbf{\&}\sigma(q_1)\mathbf{\&}\ldots\mathbf{\&}\sigma(q_n)$ (un tel nœud de motif existe car $\mathbb{P}$ est récursive). Posons $g = \sigma \circ f$.

Nous avons une notion naturelle d'intersection sur P' ; si $\mathfrak{q} = (t, S)$ et $\mathfrak{q}' = (t', S')$, nous posons :

$$\mathfrak{q}_1 \cap \mathfrak{q}_2 = (t_1\mathbf{\wedge}t_2, S_1 \cup S_2)$$

On a clairement : $g(\mathfrak{q}_1 \cap \mathfrak{q}_2) \simeq g(\mathfrak{q}_1)\mathbf{\&}g(\mathfrak{q}_2)$.

On définit également un opérateur d'intersection sur $G'P'$ (lorsque qu'aucune règle ne permet de définir $p \cap p'$, on pose $p \cap p' = p' \cap p$) :

$$\begin{array}{lcl} (x\mathbf{\&}p) \cap p' & = & x\mathbf{\&}(p \cap p') \\ ((x := c)\mathbf{\&}p) \cap p' & = & (x := c)\mathbf{\&}(p \cap p') \\ (p_1 \mid p_2) \cap p' & = & (p_1 \cap p') \mid (p_2 \cap p') \\ t \cap t' & = & t\mathbf{\wedge}t' \\ t \cap ((t_1, S_1), (t_2, S_2)) & = & ((\mathbb{0}, S_1), (\mathbb{0}, S_2)) \\ (\mathfrak{q}_1, \mathfrak{q}_2) \cap (\mathfrak{q}'_1, \mathfrak{q}'_2) & = & (\mathfrak{q}_1 \cap \mathfrak{q}'_1, \mathfrak{q}_2 \cap \mathfrak{q}'_2) \end{array}$$

On voit facilement, en déroulant ces définitions, que si $p, p' \in G'P'$, alors $G'f(p \cap p') \simeq G'f(p)\mathbf{\&}G'f(p')$.

Remarque 6.20 *Pour le cas $t \cap (\mathfrak{q}_1, \mathfrak{q}_2)$, il s'agit de calculer un motif p qui échoue toujours ($\lfloor p \rfloor \simeq \mathbb{0}$), mais qui préserve l'ensemble des variables (pour préserver la bonne formation des motifs). Nous aurions tout aussi bien pu prendre le motif $x_1\mathbf{\&}\ldots x_n\mathbf{\&}\mathbb{0}$, où $\{x_i \mid i = 1..n\} = Var((\mathfrak{q}_1, \mathfrak{q}_2))$.*

Nous pouvons alors définir la traduction $N : GP \to G'P'$, par une simple induction sur les motifs. Pour un type t, nous notons $\mathfrak{q}(t)$ l'élément $(t, \emptyset) \in P'$ de sorte que $g(\mathfrak{q}(t)) \simeq t$. De même, pour un nœud de motif $q \in P$, nous notons $\mathfrak{q}(q)$ l'élément $(\mathbb{1}, \{q\}) \in P'$, de sorte que $g(\mathfrak{q}(q)) \simeq \sigma(q)$. Pour un type $t \leq \mathbb{1}_{\mathbf{prod}}$, nous choisissons un ordre d'énumération des éléments de $\pi(t)$.

$$
\begin{array}{lcl}
N(t) & = & (t\backslash\mathbb{1}_{\mathbf{prod}}) \mid (\mathfrak{q}(t_1^1), \mathfrak{q}(t_2^1)) \mid \ldots \mid (\mathfrak{q}(t_1^n), \mathfrak{q}(t_2^n)) \\
 & & \text{où } \pi(t \wedge \mathbb{1}_{\mathbf{prod}}) = \{(t_1^i, t_2^i) \mid i = 1..n\} \\
N(x) & = & x \& N(\mathbb{1}) \\
N((x := c)) & = & (x := c) \& N(\mathbb{1}) \\
N((q_1, q_2)) & = & (\mathfrak{q}(q_1), \mathfrak{q}(q_2)) \\
N(p_1 | p_2) & = & N(p_1) \mid N(p_2) \\
N(p_1 \& p_2) & = & N(p_1) \cap N(p_2)
\end{array}
$$

Une induction sur p donne immédiatement $Gf(N(p)) \simeq p$ pour tout p. Pour appliquer la construction de la section précédente, il ne reste donc plus qu'à vérifier que la G-coalgèbre $\mathbb{P}' = (P', N \circ \sigma \circ f)$ est régulière. Il suffit de vérifier que pour toute base B de $\mathbb{P}$, et tout socle $\sqsupseteq$ qui contient au moins $\mathbb{1}_{\mathbf{prod}}$, le sous-ensemble $B'_{\sqsupseteq} = \sqsupseteq \times \mathcal{P}_f(B)$ est une base de $\mathbb{P}'$ (en effet, tout élément de P' est dans un tel $B'_{\sqsupseteq}$). Or si $\mathfrak{q} \in B'_{\sqsupseteq}$, on voit, d'après la définition de f que $g(\mathfrak{q})$ est dans GB et que le seul type qui apparaît dedans est dans $\sqsupseteq$. On prouve alors facilement par induction sur p que si $p \in GB$ et que tous les types qui apparaissent dedans sont dans $\sqsupseteq$, alors $N(p) \in G'(B'_{\sqsupseteq})$, ce qui permet de conclure. Pour le cas $p = p_1 \& p_2$, on utilise le fait que $G'(B'_{\sqsupseteq})$ est stable pour l'opérateur binaire $\cap$.

Deuxième partie

Aspects algorithmiques

Chapitre 7

Algorithme de sous-typage

Dans ce chapitre, nous nous intéressons à l'aspect algorithmique de la relation de sous-typage induite par les modèles universels (Section 4.5).

Nous ramenons l'étude du sous-typage à celle du test de vide, en utilisant l'équivalence $t_1 \leq t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. Nous modularisons la présentation de l'algorithme de sous-typage en exprimant le prédicat unaire $[\![\_]\!] \neq \emptyset$ sous forme inductive, c'est-à-dire comme le plus petit point fixe d'un opérateur monotone, et en étudiant séparément le calcul de ce genre de prédicats. Cette modularisation permet de séparer les difficultés et d'optimiser séparément les deux étapes. La Section 7.1 introduit le formalisme sous-jacent et deux algorithmes de calcul, la Section 7.2 montre comment déduire de la définition des modèles universels la définition d'un opérateur monotone dont le prédicat $[\![\_]\!] \neq \emptyset$ est le plus petit point fixe, et la Section 7.3 présente des variantes et des optimisations sur la définition de cet opérateur.

7.1 Calcul d'un prédicat inductif

Dans cette section, nous travaillons avec un ensemble fini $\beth$ de variables, notées $t, t', \ldots$. Nous notons $B = \{0, 1\}$. Les opérateurs binaires $\vee, \wedge$ sont définis naturellement sur B, ainsi que la relation d'ordre $0 \leq 1$.

7.1.1 Formules, systèmes, induction

Definition 7.1 *Une* **formule** *est un terme engendré par les productions ci-dessous :*

$$\phi ::= t \mid 0 \mid 1 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$$

L'ensemble des formules est noté $\mathcal{F}$.

Definition 7.2 *Une* **congruence** *est une fonction $\rho : \mathcal{F} \to B$ telle que $\rho(0) = 0$, $\rho(1) = 1$, $\rho(\phi_1 \vee \phi_2) = \rho(\phi_1) \vee \rho(\phi_2)$, $\rho(\phi_1 \wedge \phi_2) = \rho(\phi_1) \wedge \rho(\phi_2)$. On identifie les congruences aux fonctions $\beth \to B$.*

Definition 7.3 *Un* **système** *S est un ensemble de contraintes de la forme $(\phi \Rightarrow t)$. Pour une congruence ρ, on note $\rho \vDash S$ lorsque $\rho(\phi) \leq \rho(t)$ pour toute contrainte $(\phi \Rightarrow t)$ de S.*

Definition 7.4 *Pour un système S et une formule ϕ, on écrit $S \vDash \phi$ lorsque :*

$$\forall \rho \vDash S.\ \rho(\phi) = 1$$

On étend cette notation à un ensemble de formules : $S \vDash \{\phi_1, \ldots, \phi_n\} \iff \forall i.\ S \vDash \phi_i$. Si S et S' sont deux systèmes, on écrit $S \simeq S'$ pour : $\forall \phi.\ S \vDash \phi \iff S' \vDash \phi$.

Nous voulons *calculer* ce jugement $S \vDash \phi$. Nous commençons par donner une présentation alternative sous forme axiomatique, qui, à défaut de donner directement un algorithme, permet d'*observer* toutes les propriétés dont on aura besoin sur $\vDash$.

Definition 7.5 *On définit le jugement $S \vdash \phi$ par le système inductif ci-dessous :*

$$\frac{S \vdash \phi \quad (\phi \Rightarrow t) \in S}{S \vdash t} \qquad \frac{}{S \vdash 1}$$

$$\frac{S \vdash \phi_1}{S \vdash \phi_1 \vee \phi_2} \qquad \frac{S \vdash \phi_2}{S \vdash \phi_1 \vee \phi_2} \qquad \frac{S \vdash \phi_1 \quad S \vdash \phi_2}{S \vdash \phi_1 \wedge \phi_2}$$

On définit une fonction $[\![S]\!] : \mathcal{F} \to B$ par :

$$[\![S]\!](\phi) = 1 \iff S \vdash \phi$$

Lemme 7.6 *La fonction $[\![S]\!]$ est une congruence et $[\![S]\!] \vDash S$.*

> *Preuve*: Pour voir que $[\![S]\!]$ est une congruence, il suffit d'inspecter les règles qui définissent le jugement $\vdash$. Prouvons que $[\![S]\!] \vDash S$. Soit $(\phi \Rightarrow t)$ une contrainte de S. Supposons $[\![S]\!](\phi) = 1$. Cela signifie que $S \vdash \phi$. En appliquant une règle, on obtient $S \vdash t$, c'est-à-dire $[\![S]\!](t) = 1$. On a bien prouvé que $[\![S]\!](\phi) \leq [\![S]\!](t)$. □

Theorème 7.7 *Pour tout système S et toute formule ϕ, on a :*

$$S \vdash \phi \iff S \vDash \phi$$

> *Preuve*: Supposons que $S \vDash \phi$, c'est-à-dire $\forall \rho \vDash S.\ \rho(\phi) = 1$. En prenant $\rho = [\![S]\!]$, on obtient $[\![S]\!](\phi) = 1$, c'est-à-dire $S \vdash \phi$.
>
> L'implication $S \vdash \phi \Rightarrow S \vDash \phi$ se prouve par induction sur la dérivation de $S \vdash \phi$. Tous les cas sont faciles. □

La définition axiomatique du jugement ne donne pas directement un algorithme ; en effet, si l'on essaie de construire une dérivation pour $S \vdash \phi$, il se peut que l'on retombe sur le même jugement, et que l'on essaie de construire une dérivation infinie.

Definition 7.8 *On définit le jugement $S, N \vdash \phi$, où S est un système, N un ensemble de variables et ϕ une formule par le système inductif :*

$$\frac{S, N \cup \{t\} \vdash \phi \quad (\phi \Rightarrow t) \in S \quad t \notin N}{S, N \vdash t} \qquad \frac{}{S, N \vdash 1}$$

$$\frac{S, N \vdash \phi_1}{S, N \vdash \phi_1 \vee \phi_2} \qquad \frac{S, N \vdash \phi_2}{S, N \vdash \phi_1 \vee \phi_2} \qquad \frac{S, N \vdash \phi_1 \quad S, N \vdash \phi_2}{S, N \vdash \phi_1 \wedge \phi_2}$$

L'ensemble N permet de « bloquer » la construction des dérivations, en empêchant de dérouler plusieurs fois la même variable sur une branche.

On voit que les dérivations pour le jugement $S, \emptyset \vdash \phi$ correspondent bijectivement et naturellement aux dérivations pour $S \vdash \phi$ qui n'utilisent pas deux fois la même variable sur la même branche. Or si $S \vdash \phi$, il existe toujours une telle dérivation : il suffit de considérer une dérivation de taille minimale. En effet, une dérivation qui utilise la même variable deux fois sur la même branche peut être « court-circuitée » pour obtenir une dérivation strictement plus petite. De même, on voit que les dérivations pour le jugement $S, N \vdash \phi$ correspondent aux dérivations pour le jugement $S_N, \emptyset \vdash \phi$, où $S_N = \{(\phi \Rightarrow t) \in S \mid t \notin N\}$.

Theorème 7.9 *Pour tout système S, toute formule ϕ et $N \subseteq \beth$:*

$$S, N \vdash \phi \iff S_N \vdash \phi$$

En particulier :

$$S, \emptyset \vdash \phi \iff S \vdash \phi$$

Pour un système S fini, la définition du jugement $S, N \vdash \phi$ donne bien un algorithme non déterministe (qui termine).

Lemme 7.10 *Soit S un système et $(\phi \Rightarrow t)$ une contrainte. Alors :*

$$S \nVdash \phi \Rightarrow S \simeq S \cup \{(\phi \Rightarrow t)\}$$

Preuve: Notons $S' = S \cup \{(\phi \Rightarrow t)\}$. On a clairement $[\![S]\!] \leq [\![S']\!]$ car $S \subseteq S'$. Posons $\rho = [\![S]\!]$. On a $\rho \vDash \{(\phi \Rightarrow t)\}$ car $S \nVdash \phi$, c'est-à-dire $\rho(\phi) = 0$. En combinant cela avec $\rho \vDash S$, on obtient $\rho \vDash S'$, et donc $[\![S']\!] \leq \rho$. □

Lemme 7.11 *Soit S un système et $N \subseteq \beth$. Alors :*

$$t \in N \Rightarrow S_N \nVdash t$$

Preuve: Évident, avec la définition axiomatique de $S_N \vdash t$. □

7.1.2 Cache

Dans cette section, nous présentons un algorithme de calcul de $[\![S]\!]$ qui garde soigneusement trace des résultats intermédiaires pour éviter des calculs inutiles. Nous supposons que le système S est donné par une fonction Φ des variables dans les formules :

$$S = \{(\Phi(t) \Rightarrow t) \mid t \in \beth\}$$

Présentation informelle L'algorithme maintient deux ensembles disjoints N et P de variables. Les variables t dans P sont celles pour lesquelles l'algorithme a déjà établi $[\![S]\!](t) = 1$ de manière sûre. L'ensemble N joue un rôle similaire à celui qu'il joue dans la définition du prédicat $S, N \vdash \phi$ de la section précédente : il garde trace des variables t qui ont déjà été considérées, et que l'on s'interdit de dérouler de nouveau, pour éviter une récursion infinie. Lorsque l'algorithme considère une variable t qui est dans N (resp. dans P), il répond

immédiatement 0 (resp. 1). Sinon, il ajoute temporairement t à N et commence à évaluer la formule $\Phi(t)$. Si le résultat est 0, l'algorithme renvoie 0 pour t ; mais contrairement au prédicat $S, N \vdash \phi$ de la section précédente, l'algorithme laisse alors t dans N, ainsi que toutes les autres variables qui y ont été ajoutées lors du calcul de $\Phi(t)$. Si le résultat du calcul de $\Phi(t)$ est 1, l'algorithme renvoie bien entendu 1 pour t, qu'il ajoute à P et qu'il enlève de N. Mais il doit également supprimer de N toutes les variables qui y ont été ajoutées lors du calcul de $\Phi(t)$. En effet, ces variables t' sont celles pour lesquelles l'algorithme a cru avoir prouvé $[\![S]\!](t') = 0$, en supposant, à tort, que $[\![S]\!](t) = 0$, ce qui s'avère faux.

Présentation formelle La Figure 7.1 définit un algorithme pour calculer une fonction $\texttt{eval}(\phi, N, P)$ où ϕ est une formule, $N, P \subseteq \sqsupseteq$, $N \cap P = \emptyset$. Le résultat est un triplet (ϵ, N', P'), avec $\epsilon \in B$, et $N', P' \subseteq \sqsupseteq$, $N \subseteq N'$, $P \subseteq P'$, et $N' \cap P' = \emptyset$.

```
eval(t, N, P) :=
 if t ∈ P then (1, N, P)
 else if t ∈ N then (0, N, P)
 else match eval(Φ(t), N ∪ {t}, P) with
      | (0, N', P') → (0, N', P')
      | (1, N', P') → (1, N, P' ∪ {t})
eval(φ1 ∧ φ2, N, P) :=
 match eval(φ1, N, P) with
      | (0, N', P') → (0, N', P')
      | (1, N', P') → eval(φ2, N', P')
eval(φ1 ∨ φ2, N, P) :=
 match eval(φ1, N, P) with
      | (0, N', P') → eval(φ2, N', P')
      | (1, N', P') → (1, N', P')
eval(0, N, P) := (0, N, P)
eval(1, N, P) := (1, N, P)
```

FIG. 7.1 – Algorithme avec cache

Theorème 7.12 (Terminaison) *L'algorithme donné à la Figure 7.1 termine.*

> *Preuve*: Induction sur le couple (N, ϕ) ordonné lexicographiquement, avec $N' \leq N \iff N \subseteq N'$ (ordre bien fondé car $\sqsupseteq$ est fini). □

Remarque 7.13 *On pourrait croire que si* $\texttt{eval}(\phi, N, P) = (1, N', P')$*, alors* $N' = N$*, ce qui permettrait de remplacer l'expression de retour* $(1, N, P' \cup \{t\})$ *dans la dernière ligne de la définition de* $\texttt{eval}(t, N, P)$ *par* $(1, N' \backslash \{t\}, P' \cup \{t\})$*. Cela est faux à cause du cas* $\phi_1 \vee \phi_2$*, car le premier appel récursif peut renvoyer* $(0, N', P')$*, avec* $N \subsetneq N'$*.*

Pour retrouver N à partir de N', il ne suffit donc pas de supprimer l'élément t : il peut être nécessaire de supprimer un nombre a priori *quelconque d'éléments.*

On pose :

$$S(N,P) = \{\Phi(t) \Rightarrow t \mid t \notin N\} \cup \{1 \Rightarrow t \mid t \in P\}$$

Ce système capture l'invariant courant de l'algorithme : les implications $\Phi(t) \Rightarrow t$ avec t dans N ne sont pas prises en compte (parce que l'on suppose que la valeur de vérité de t est 0), et les variables t dans P sont supposées avec une valeur de vérité 1, ce qui se traduit par des contraintes $1 \Rightarrow t$.

Theorème 7.14 (Invariant) *Si* $\texttt{eval}(\phi, N, P) = (\epsilon, N', P')$ *avec* $\epsilon \in \{0,1\}$, *alors :* $S(N,P) \simeq S(N',P')$ *et* $[\![S(N,P)]\!](\phi) = \epsilon$.

> *Preuve*: La preuve se fait par induction, en utilisant naturellement le même ordre que pour la preuve de terminaison.
> Cas $\phi = t$: si $t \in P$, alors clairement $S(N,P) \vDash t$. Si $t \in N$, alors $S(N,P) \nvDash t$ en utilisant le Lemme 7.11. Supposons maintenant $t \notin P \cup N$, et posons $(\epsilon, N', P') = \texttt{eval}(\Phi(t), N \cup \{t\}, P)$. Par induction, on a : $S(N \cup \{t\}, P) \simeq S(N', P')$, $[\![S(N \cup \{t\}, P)]\!](\Phi(t)) = \epsilon$.
>
> Considérons d'abord le cas $\epsilon = 0$. En appliquant le Lemme 7.10 au système $S(N \cup \{t\}, P)$ et à la contrainte $(\Phi(t) \Rightarrow t)$, on obtient $S(N \cup \{t\}, P) \simeq S(N,P)$. On en déduit $S(N,P) \simeq S(N',P')$ et $S(N,P) \nvDash t$ (car $S(N \cup \{t\}, P) \nvDash t$ d'après le Lemme 7.11).
>
> Considérons maintenant le cas $\epsilon = 1$. Comme $S(N \cup \{t\}, P) \subseteq S(N,P)$, on obtient, *a fortiori* : $S(N,P) \vDash \Phi(t)$ et $S(N,P) \vDash P'$. Mais $(\Phi(t) \Rightarrow t) \in S(N,P)$, et donc $S(N,P) \vDash t$. Puisque $S(N,P) \vDash P' \cup \{t\}$, on a bien $S(N,P) \simeq S(N, P' \cup \{t\})$.
> Cas $\phi = \phi_1 \wedge \phi_2$: posons $(\epsilon, N', P') = \texttt{eval}(\phi_1, N, P)$. Par induction, on a : $S(N',P') \simeq S(N,P)$ et $[\![S(N,P)]\!](\phi_1) = \epsilon$. Si $\epsilon = 0$, alors on a aussi $[\![S(N,P)]\!](\phi) = 0$, ce qui conclut ce cas. Si $\epsilon = 1$, alors, en posant $(\epsilon', N'', P'') = \texttt{eval}(\phi_2, N, P)$, on obtient, par induction : $S(N'',P'') \simeq S(N',P')$ et $[\![S(N',P')]\!](\phi_2) = \epsilon'$. On en déduit donc $S(N,P) \simeq S(N'',P'')$ et $[\![S(N,P)]\!](\phi_2) = \epsilon'$, ce qui permet de conclure ce cas.
> Cas $\phi = \phi_1 \vee \phi_2$: similaire au cas précédent.
> Cas $\phi = 0$ ou $\phi = 1$: triviaux. □

En partant de $N = P = \emptyset$, on obtient un algorithme pour décider si $S \vDash \phi$, pour n'importe quelle formule ϕ. En effet, on a clairement $S(\emptyset, \emptyset) = S$. On récupère de plus en sortie de l'algorithme deux ensembles P' et N' qui peuvent servir pour une utilisation ultérieure de l'algorithme (plus ces ensembles sont grands, plus l'algorithme termine rapidement). Notons que si $\phi = t$, alors t est dans un des deux ensembles P' ou N' ; cela montre que si l'on a déjà utilisé l'algorithme pour calculer $[\![S]\!](t)$, un appel ultérieur avec le même argument termine immédiatement.

Implémentation Nous avons présenté une version purement fonctionnelle de l'algorithme. En pratique, pour des raisons d'efficacité, on peut implémenter les ensembles P et N par des structures de données impératives comme des tables de hachage. Pour l'ensemble P, cela ne pose aucun problème, car il ne fait que croître au cours de l'algorithme. Il faut être plus prudent avec l'ensemble N. En effet, le dernier cas dans la définition de $\texttt{eval}(t, N, P)$ renvoie la valeur initiale

de N, et non celle obtenue après l'appel récursif. Il faut donc pouvoir revenir en arrière pour retrouver la valeur initiale de N. Cela peut se faire facilement, sans sauvegarder complètement la table de hachage : il suffit de garder dans une pile l'historique des éléments ajoutés à N ; pour revenir à la valeur initiale de N, il suffit de dépiler jusqu'à tomber de nouveau sur t, et de supprimer dans N, en parallèle, tous les éléments dépilés. La pile et la table de hachage contiennent en permanence les mêmes objets (ceux de N) : la pile est utilisée pour pouvoir revenir en arrière, et la table de hachage pour tester rapidement si un élément est dans N.

Bien entendu, il est possible d'utiliser la même table de hachage pour représenter en même temps P et N (ce qui permet de n'avoir qu'un seul accès pour les deux tests $t \in P$, $t \in N$).

Remarque 7.15 *Hosoya [Hos01] signale que pour implémenter l'algorithme de sous-typage de XDuce, il n'est pas envisageable d'utiliser une structure modifiable en place, comme une table de hachage, pour représenter les hypothèses intermédiaires qui peuvent devoir être annulées lors d'un retour-arrière (ce qui correspond à notre ensemble N). La technique que l'on propose ci-dessus permet cependant d'utiliser une telle structure de donnée, sans devoir la copier complètement.*

Optimisation Dans la définition de $\texttt{eval}(t, N, P)$, on peut remplacer la contrainte $\Phi(t)$ par une contrainte équivalente modulo les hypothèses présentes : remplacer une variable t' telle que $t' \in P$ (resp. $t' \in N$) par 1 (resp. 0), et simplifier en utilisant des tautologies booléennes (telle que $\phi \wedge 0 \simeq 0$). Par exemple, si $\Phi(t) = t_1 \wedge t_2$ et $t_2 \in N$, alors on peut remplacer $\Phi(t)$ par 0, ce qui évite de « dérouler » t_1. La preuve de la correction de cette optimisation ne pose aucun problème.

Retour-arrière Lorsque, dans la dernière ligne du cas $\phi = t$, on renvoie N, cela revient à invalider toutes les hypothèses négatives qui ont été ajoutées à N' lors de l'appel récursif. En effet, le résultat $(1, N', P')$ signifie que la valeur de vérité des $t' \in N'$ est 0, *en supposant* que celle de t est 0, mais cela est faux.

Exemple 7.16 *Voici un exemple qui illustre cela. Soit $\sqsupset = \{t_1, t_2\}$, et $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) = t_1$. On veut calculer la valeur de vérité pour t_1. L'algorithme calcule $\texttt{eval}(t_2 \vee 1, \{t_1\}, \emptyset) = (1, \{t_1, t_2\}, \emptyset)$, et donc $\texttt{eval}(t_1, \emptyset, \emptyset) = (1, \emptyset, \{t_1\})$. Il est nécessaire de supprimer non seulement t_1, mais aussi t_2 de l'ensemble N intermédiaire, car la valeur de vérité pour t_2 est 1 (il se trouve dans $N = \{t_1, t_2\}$ à cause d'une hypothèse qui s'avère fausse).*

7.1.3 Suppression du retour-arrière

L'algorithme de la section précédente, avant de dérouler la définition de $\Phi(t)$ lorsqu'il considère une variable t, commence par supposer que la valeur de vérité de t est 0. Pendant le calcul de $\Phi(t)$, l'algorithme peut établir que la valeur de vérité d'une autre variable t' est 0 ; dans la mesure où cette déduction dépend éventuellement de l'hypothèse que la valeur de vérité de t est 0, il faut l'invalider si cette hypothèse s'avère fausse. C'est le phénomène de retour-arrière

mis en évidence dans l'Exemple 7.16. Cependant, dans certains cas, ce retour-arrière systématique de l'ensemble N invalide des faits qui ne dépendent pas de l'hypothèse qui s'avère fausse, comme le montre l'exemple ci-dessous.

Exemple 7.17 *Modifions l'Exemple 7.16 pour prendre* $\Phi(t_2) = 0$. *On obtient encore* $\texttt{eval}(t_2 \vee 1, \{t_1\}, \emptyset) = (1, \{t_1, t_2\}, \emptyset)$, *mais le fait que* t_2 *se retrouve dans l'ensemble* $N = \{t_1, t_2\}$ *en sortie ne dépend pas du fait que* t_1 *se trouvait dans l'ensemble* $N = \{t_1\}$ *en entrée. L'algorithme de la section précédente ne peut distinguer cette situation de celle de l'Exemple 7.16, car il oublie les dépendances entre les hypothèses (qui peuvent être invalidées), et leurs conclusions.*

Dans cette section, nous présentons un nouvel algorithme qui calcule $[\![S]\!]$ en évitant complètement les retours-arrière. L'idée, plutôt que de supposer qu'une variable t a une valeur de vérité 0 pendant le calcul de $\Phi(t)$, est de garder trace des dépendances de t, c'est-à-dire des contraintes qu'il faut prendre en compte s'il s'avère *in fine* que la valeur de vérité de t est 1.

Présentation informelle L'algorithme garde pour chaque variable t déjà rencontrée un état. Il peut s'agir d'une constante 0 ou 1, qui signifie que la valeur de vérité pour cette variable est connue de manière sûre. Il peut également s'agir d'un ensemble de contraintes $[C_1; \ldots; C_n]$ qu'il faut prendre en compte si jamais on se rend compte que la valeur de vérité de t est 1, et que l'on peut ignorer sinon.

L'état global de l'algorithme, qui réunit les états de toutes les variables, est appelé table courante. Lorsque l'on considère une nouvelle variable t, on commence par lui donner l'état [] (ensemble vide de contraintes), et on « active » la contrainte $\Phi(t) \Rightarrow t$. On fait, nous allons considérer des contraintes étendues, de la forme $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. Activer une telle contrainte avec $n = 0$ revient à positionner l'état de t à 1, et à activer successivement les contraintes $C_1, \ldots, C_n$ si l'état de t était auparavant $[C_1; \ldots; C_n]$. Si $n > 0$, activer une contrainte $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ se fait en considérant la forme de ϕ_1. Si $\phi_1 = 1$, on active $\phi_2 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. Si $\phi_1 = 0$, il n'y a rien à faire. Si $\phi_1 = t'$, on commence par considérer t' ; si son état est alors 0 ou 1, on procède comme dans les cas $\phi_1 = 0$ ou $\phi_1 = 1$ ci-dessus ; si son état est un ensemble de contraintes $[C_1; \ldots; C_n]$, on ajoute à cet ensemble la contrainte supplémentaire $\phi_2 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$. En effet, s'il s'avère que valeur de vérité de t' est 1, il faudra de nouveau considérer cette contrainte que l'on suspend provisoirement. Enfin, si $\phi_1 = \phi \wedge \phi'$, on active $\phi \Rightarrow \phi' \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, et si $\phi_1 = \phi \vee \phi'$, on active successivement les deux contraintes $\phi \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ et $\phi' \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$.

Lorsque toutes les contraintes à activer ont été traitées, on peut remplacer tous les états de la forme $[C_1; \ldots; C_n]$ par 0, car on a alors l'assurance que la valeur de vérité des variables correspondantes ne pourra pas être 1. Les états possibles pour toutes les variables considérées sont alors 0 et 1.

On peut ajouter un certain nombre de remarques, qui se traduiront par des optimisations dans l'algorithme.

- Au moment d'activer une contrainte $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, si on l'a déjà établi que la valeur de vérité de t est 1, alors on peut tout de suite ignorer cette contrainte.
- De même, une contrainte de la forme $t \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$ peut être ignorée, car elle est forcément satisfaite.

– Lorsque l'algorithme considère une nouvelle variable t, il positionne son état à l'ensemble vide de contraintes [] et active la contrainte $\Phi(t) \Rightarrow t$. Après avoir effectué cette opération, si l'état de t est un ensemble de contraintes $[C_1; \ldots; C_n]$, on peut parfois le remplacer par 0, ce qui évitera d'étendre cet ensemble par la suite. Il suffit d'être sûr que la valeur de vérité de t est bien 0 ; c'est bien le cas si aucune des contraintes en suspens dans la table courante ne « pointe » vers t.

Présentation formelle Nous allons donner une présentation formelle de l'algorithme décrit ci-dessus. Commençons par définir formellement les notions de contrainte étendue et de table.

Definition 7.18 *Une* **contrainte longue** *est soit une variable t, soit un terme de la forme $\phi \Rightarrow C$, où ϕ est une formule et C est une contrainte longue. On peut voir une contrainte longue C comme une liste finie de formules, suivie d'une variable, que l'on note $v(C)$. Les définitions sur les contraintes et les systèmes de contraintes se transfèrent sur les contraintes longues et les systèmes de contraintes longues, en associant à la contrainte longue $C = \phi_1 \Rightarrow \ldots\ldots \phi_n \Rightarrow v(C)$ la contrainte $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow v(C)$ (si $n = 0$, on prend $1 \Rightarrow v(C)$).*

Definition 7.19 *Une* **table** *est une fonction σ qui associe à toute variable soit 0, soit 1, soit $\bot$, soit une séquence finie de contraintes longues $L = [C_1; \ldots; C_n]$ telles que $\sigma(v(C_i)) \notin \{0, \bot\}$.*

On note $\sigma\{t \to X\}$ la table σ' telle que $\sigma'(t) = X$ et $\sigma'(t') = \sigma(t')$ pour $t' \neq t$.

Definition 7.20 *Le* **domaine** *d'une table σ est l'ensemble $Dom(\sigma)$ des variables t telles que $\sigma(t) \neq \bot$. On définit un préordre sur les tables par :*

$$\sigma \leq \sigma' \iff \begin{cases} Dom(\sigma') \subseteq Dom(\sigma) \\ \{t \mid \sigma'(t) = 1\} \subseteq \{t \mid \sigma(t) = 1\} \end{cases}$$

Cet préordre est bien fondé (car $\sqsupset$ est fini).

Une table avec aucune contrainte en suspens et qui est correcte par rapport au système S est appelée solution partielle.

Definition 7.21 *Une* **solution partielle** *est une table σ telle que :*

$$\forall t \in Dom(\sigma).\ \sigma(t) = [\![S]\!](t)$$

La Figure 7.2 définit deux fonctions `extend` et `trigger` mutuellement récursives. Elles prennent une table courante en entrée et renvoient une table mise à jour en sortie. Le rôle de la fonction `extend` est de « considérer » une nouvelle variable t, c'est-à-dire de s'assurer que la table courante est définie sur t. La fonction `trigger` s'occupe d'activer une contrainte ; elle utilise une fonction auxiliaire `trigger'` qui n'est appelée que lorsque l'état de la variable pointée par la contrainte n'est pas 1. La fonction auxiliaire `may` est utilisée pour vérifier qu'aucune contrainte en suspens ne pointe vers une variable donnée. Enfin, la fonction `clean` supprime toutes les contraintes en suspens et remplace les états correspondants par 0.

Theorème 7.22 *Soit σ une solution partielle et t une variable. Posons $\sigma' =$* `clean(extend(`σ, t`))`*, tel que défini à la Figure 7.2. Alors σ' est une solution partielle bien définie (i.e. l'algorithme termine), avec $\sigma' \leq \sigma$ et $t \in Dom(\sigma')$.*

Nous donnerons la preuve de ce théorème plus bas.

```
extend(σ, t) :=
  if σ(t) = ⊥ then
    let σ' = trigger(σ{t→[]}, Φ(t)⇒t) in
    if may(σ', t) then σ' else σ'{t→0}
  else
    σ

trigger(σ, C) := if σ(v(C)) = 1 then σ else trigger'(σ, C)
trigger'(σ, t) := let L = σ(t) in trigger*(σ{t→1}, L)
trigger'(σ, t⇒C) :=
  if v(C) = t then σ else
  let σ' = extend(σ, t) in
  match σ'(t) with
  | 0 → σ'
  | 1 → trigger(σ', C)
  | L → σ'{t→C :: L}
trigger'(σ, 0⇒C) := σ
trigger'(σ, 1⇒C) := trigger'(σ, C)
trigger'(σ, φ1∧φ2⇒C) := trigger'(σ, φ1⇒(φ2⇒C))
trigger'(σ, φ1∨φ2⇒C) := trigger(trigger'(σ, φ1⇒C), φ2⇒C)

trigger*(σ, []) := σ
trigger*(σ, C :: L) := trigger*(trigger(σ, C), L)

may(σ, t) := σ(t) = 1 ∨ ∃t'.∃C∈σ(t'). v(C) = t

clean(σ) := (t ↦ if σ(t) ∈ {0, 1, ⊥} then σ(t) else 0)
```

FIG. 7.2 – Algorithme sans retour-arrière

En partant de la table de domaine vide, on obtient un algorithme pour décider si $S \vDash t$ pour une variable t. On récupère en sortie une table σ' qui peut être utilisée comme entrée pour l'appel suivant de l'algorithme (cache global).

Implémentation Comme pour l'algorithme avec cache, nous avons présenté une version fonctionnelle de l'algorithme. Ici, cependant, la table σ est utilisée de manière purement linéaire (c'est en ce sens qu'il s'agit d'un algorithme sans retour-arrière). On peut ainsi facilement implémenter l'algorithme en utilisant une structure de donnée impérative, comme une table de hachage, pour représenter σ. Pour calculer la fonction `may`, on peut tenir à jour un compteur, pour chaque variable t, du nombre de contraintes longues $C \in \sigma(t')$ telles que

$v(C) = t$. Cela permet de calculer cette fonction en temps constant. Pour implémenter la fonction `clean`, on garde simplement une liste des variables qui ont été ajoutées au domaine de σ.

Optimisations L'optimisation mentionnée pour l'algorithme avec cache se transpose. Dans la définition de $\texttt{extend}(\sigma, t)$, on peut simplifier la contrainte $\Phi(t)$ en utilisant les hypothèses présentes (si $\sigma(t') \in \{0, 1\}$, on peut remplacer t' par $\sigma(t')$), et des tautologies booléennes. De même, on peut simplifier une contrainte longue $\phi_1 \Rightarrow \ldots \Rightarrow \phi_n \Rightarrow t$, et aussi changer l'ordre des ϕ_i (pour considérer d'abord les formules plus petites, par exemple).

Exemples Donnons quelques exemples de déroulement de l'algorithme. Nous montrons à chaque fois l'arbre des appels récursifs entre les fonctions `trigger`, `extend` et `may`. Nous notons $\{\}$ la table vide, et nous nommons toutes les tables intermédiaires qui apparaissent dans l'algorithme, par ordre d'apparition.

Exemple 7.23 *Reprenons l'Exemple 7.17, qui illustrait un retour-arrière effectué inutilement par l'algorithme de la section précédente. On a* $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) = 0$. *On cherche à calculer la valeur de vérité de* t_1. *Voici le déroulement de l'algorithme pour l'appel* $\texttt{extend}(\{\}, t_1)$ *:*

$\texttt{extend}(\{\}, t_1) = \sigma_4$
- $\texttt{trigger}(\sigma_1, t_2 \vee 1 \Rightarrow t_1) = \sigma_4$
  - $\texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_3$
    - $\texttt{extend}(\sigma_1, t_2) = \sigma_3$
      - $\texttt{trigger}(\sigma_2, 0 \Rightarrow t_2) = \sigma_2$
      - $\texttt{may}(\sigma_2, t_2) = \texttt{false}$
  - $\texttt{trigger}(\sigma_3, 1 \Rightarrow t_1) = \sigma_4$
    - $\texttt{trigger}(\sigma_3, t_1) = \sigma_4$
- $\texttt{may}(\sigma_4, t_1) = \texttt{true}$

où :

$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [], t_2 \to 0\}$
$\sigma_4 = \{t_1 \to 1, t_2 \to 0\}$

Le résultat final est $\{t_1 \to 1, t_2 \to 0\}$ *(qui est invariant par* `clean`*). Lorsque l'on a déroulé la définition de* $\Phi(t_2)$*, on a vu qu'il n'y a aucun moyen de prouver* t_2 *(en utilisant* `may`*), et donc on a pu calculer tout de suite sa valeur de vérité* 0.

Exemple 7.24 *Reprenons maintenant l'Exemple 7.16 :* $\Phi(t_1) = t_2 \vee 1$, $\Phi(t_2) =$

t_1. *Voici le déroulement de l'algorithme pour l'appel* $\texttt{extend}(\{\}, t_1)$ *:*

$\texttt{extend}(\{\}, t_1) = \sigma_6$
| $\texttt{trigger}(\sigma_1, t_2 \vee 1 \Rightarrow t_1) = \sigma_6$
| | $\texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_4$
| | | $\texttt{extend}(\sigma_1, t_2) = \sigma_3$
| | | | $\texttt{trigger}(\sigma_2, t_1 \Rightarrow t_2) = \sigma_3$
| | | | | $\texttt{extend}(\sigma_2, t_1) = \sigma_2$
| | | | $\texttt{may}(\sigma_3, t_2) = \texttt{true}$
| | $\texttt{trigger}(\sigma_4, 1 \Rightarrow t_1) = \sigma_6$
| | | $\texttt{trigger}(\sigma_4, t_1) = \sigma_6$
| | | | $\texttt{trigger}(\sigma_5, t_2) = \sigma_6$
| | | | | $\texttt{trigger}(\sigma_6, t_1) = \sigma_6$
| $\texttt{may}(\sigma_6, t_1) = \texttt{true}$

où :
$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [t_2], t_2 \to []\}$
$\sigma_4 = \{t_1 \to [t_2], t_2 \to [t_1]\}$
$\sigma_5 = \{t_1 \to 1, t_2 \to [t_1]\}$
$\sigma_6 = \{t_1 \to 1, t_2 \to 1\}$

Le résultat final est $\{t_1 \to 1, t_2 \to 1\}$ *(qui est invariant par* `clean`*). Au cours du calcul, la dépendance* $t_1 \Rightarrow t_2$ *a été conservée ; ainsi, lorsque* t_1 *s'est avéré vrai (du fait du* 1 *dans* $\Phi(t_1) = t_2 \vee 1$*), on a su tout de suite que* t_2 *était vrai également.*

Exemple 7.25 *Donnons un autre exemple, avec* $\Phi(t_1) = t_2$ *et* $\Phi(t_2) = t_1$*. Voici la trace de l'algorithme :*

$\texttt{extend}(\{\}, t_1) = \sigma_4$
| $\texttt{trigger}(\sigma_1, t_2 \Rightarrow t_1) = \sigma_4$
| | $\texttt{extend}(\sigma_1, t_2) = \sigma_3$
| | | $\texttt{trigger}(\sigma_2, t_1 \Rightarrow t_2) = \sigma_3$
| | | | $\texttt{extend}(\sigma_2, t_1) = \sigma_2$
| | $\texttt{may}(\sigma_3, t_2) = \texttt{true}$
| $\texttt{may}(\sigma_4, t_1) = \texttt{true}$

où :
$\sigma_1 = \{t_1 \to []\}$
$\sigma_2 = \{t_1 \to [], t_2 \to []\}$
$\sigma_3 = \{t_1 \to [t_2], t_2 \to []\}$
$\sigma_4 = \{t_1 \to [t_2], t_2 \to [t_1]\}$

L'algorithme enregistre la dépendance mutuelle entre t_1 *et* t_2*. Comme il n'y a plus de contrainte « implicite » (*$\kappa = \emptyset$ *pour les appels globaux de l'algorithme), on peut éliminer ces dépendances. C'est le rôle de* `clean`*, qui transforme la table* $\{t_1 \to [t_2], t_2 \to [t_1]\}$ *en* $\{t_1 \to 0, t_2 \to 0\}$*.*

Preuve du Théorème 7.22 La suite de cette section est consacrée à la preuve du Théorème 7.22. Nous allons introduire quelques notions auxiliaires.

Definition 7.26 *Le système de contraintes longues associé à une table σ est défini par :*

$$\begin{array}{ll} S(\sigma) & = \{\Phi(t) \Rightarrow t \mid \sigma(t) = \bot\} \\ & \cup \{t \mid \sigma(t) = 1\} \\ & \cup \{t \Rightarrow C_i \mid \sigma(t) = [C_1; \ldots; C_n], i = 1..n\} \end{array}$$

Definition 7.27 *Un* **état** *est un couple (σ, κ) où σ est une table et κ est un ensemble de contraintes longues, tel que :*

$$\begin{cases} S \simeq S(\sigma) \cup \kappa \\ \forall C \in \kappa.\ \sigma(v(C)) \notin \{0, \bot\} \end{cases}$$

Lemme 7.28 *Si (σ, κ) est un état et $\sigma(t) \in \{0, 1\}$, alors $[\![S]\!](t) = \sigma(t)$.*

Preuve: Si $\sigma(t) = 1$, alors la contrainte t est dans $S(\sigma)$, et donc $[\![S]\!](t) = [\![S(\sigma) \cup \kappa]\!](t) = 1$. Si $\sigma(t) = 0$, alors aucune contrainte dans $S(\sigma) \cup \kappa$ ne pointe vers t, et donc $[\![S(\sigma) \cup \kappa]\!](t) = 0$. □

Dans un état, l'ensemble de contraintes κ représente les contraintes implicitement stockées dans le flot de contrôle, en attente d'être « intégrées » à la table σ.

Le Théorème 7.22 est une conséquence des deux lemmes et du théorème ci-dessous.

Lemme 7.29 *Si σ est une solution partielle, alors $(\sigma, \emptyset)$ est un état.*

Preuve: Il s'agit de prouver que $S \simeq S(\sigma)$. On a :

$$\begin{array}{lll} S & = & \{\Phi(t) \Rightarrow t \mid t \in \beth\} \\ S(\sigma) & = & \{\Phi(t) \Rightarrow t \mid \sigma(t) = \bot\} \cup \{t \mid \sigma(t) = 1\} \end{array}$$

Prouvons d'abord que $[\![S]\!] \vDash S(\sigma)$. Les contraintes de la forme $\Phi(t) \Rightarrow t$ avec $\sigma(t) = \bot$ sont évidemment satisfaites sous $[\![S]\!]$. Si $\sigma(t) = 1$, alors $[\![S]\!](t) = 1$ par définition d'une solution partielle, et donc la contrainte t est bien satisfaite sous $[\![S]\!]$. On en déduit bien $[\![S]\!] \vDash S(\sigma)$, et donc $[\![S(\sigma)]\!] \leq [\![S]\!]$.

Prouvons maintenant que $[\![S(\sigma)]\!] \vDash S$. Soit $\Phi(t) \Rightarrow t$ une contrainte de S. Si $\sigma(t) = \bot$, alors la contrainte est dans $S(\sigma)$ et c'est bon. Si $\sigma(t) = 1$, alors $S(\sigma) \vDash t$, et donc *a fortiori* $S(\sigma) \vDash \Phi(t) \Rightarrow t$. Si $\sigma(t) = 0$, alors $[\![S]\!](t) = 0$, ce qui implique $[\![S]\!](\Phi(t)) = 0$. Or nous avons vu que $[\![S(\sigma)]\!] \leq [\![S(\sigma)]\!]$, ce qui donne $[\![S(\sigma)]\!](\Phi(t)) = 0$, et donc $S(\sigma) \vDash \Phi(t) \Rightarrow t$. Le cas $\sigma(t) = [C_1; \ldots; C_n]$ est impossible car σ est une solution partielle. On a bien prouvé que $[\![S(\sigma)]\!] \vDash S$, ce qui donne $[\![S]\!] \leq [\![S(\sigma)]\!]$. □

Lemme 7.30 *Si $(\sigma, \emptyset)$ est un état, alors* `clean`(σ) *est une solution partielle σ' avec $\sigma' \leq \sigma$ (pour le préordre sur les tables).*

Preuve: Posons $\sigma' =$ `clean`(σ). Considérons la fonction $\rho : \beth \to B$ telle que $\rho(t) = 1 \iff \sigma(t) \in \{1, \bot\}$. On voit facilement que $\rho \vDash S(\sigma)$: les contraintes $\Phi(t) \Rightarrow t$ avec $\sigma(t) = \bot$ et t avec

$\sigma(t) = 1$ sont vérifiées car $\rho(t) = 1$, et les contraintes $t \Rightarrow C_i$, avec $\sigma(t) = [C_1; \ldots; C_n]$ sont vérifiées car $\rho(t) = 0$. On en déduit $[\![S]\!] = [\![S(\sigma)]\!] \leq \rho$, ce qui prouve que $\sigma'(t) = 0 \Rightarrow [\![S]\!](t) = 0$. Si $\sigma'(t) = 1$, alors $\sigma(t) = 1$, et donc $[\![S]\!] = [\![S(\sigma)]\!] \vDash t$ (puisque $t \in S(\sigma)$), c'est-à-dire $[\![S]\!](t) = 1$.

La relation $\sigma' \leq \sigma$ est immédiate. En fait, on a aussi $\sigma \leq \sigma'$. □

Theorème 7.31 (Terminaison et invariant) *Soit (σ, κ) un état et $t \in \beth$. Alors* `extend`(σ, t) *est une table σ' bien définie, avec $t \in Dom(\sigma')$, $\sigma' \leq \sigma$, et telle que (σ', κ) est un état.*

Soit $(\sigma, \kappa \cup \{C\})$ un état. Alors `trigger`(σ, C) *est une table σ' bien définie, avec $\sigma' \leq \sigma$, et telle que (σ', κ) est un état.*

Preuve: Nous prouvons les deux assertions par une induction mutuelle sur σ (un utilisant le préordre bien fondé $\leq$ sur les tables).

Commençons par l'assertion sur `extend`. Si $t \in \mathrm{Dom}(\sigma)$, le résultat est σ, et il n'y a rien à prouver. Dans le cas contraire, nous cherchons à appliquer l'hypothèse d'induction pour l'appel à `trigger`. La table $\sigma\{t \to []\}$ est strictement plus petite que σ. De plus, on constate que $(\sigma\{t \to []\}, \kappa \cup \{\Phi(t) \Rightarrow t\})$ est un état car $S(\sigma\{t \to []\}) \cup \{\Phi(t) \Rightarrow t\}) = S(\sigma)$. Nous pouvons donc appliquer l'hypothèse d'induction : l'appel à `trigger` termine et renvoie une table σ' telle que (σ', κ) est un état. Si `may`(σ', t) est vrai, alors c'est fini. Sinon, on pose $\sigma'' = \sigma'\{t \to 0\}$, et il s'agit de voir que (σ'', κ) est un état. Si $\sigma'(t) = 0$, il n'y a rien à prouver. Le cas $\sigma'(t) = \bot$ est impossible car $\sigma' \leq \sigma\{t \to []\}$. Le cas $\sigma'(t) = 1$ est impossible car `may`(σ', t) est faux. La seule possibilité restante est donc $\sigma'(t) = [C_1; \ldots; C_n]$. Aucune contrainte de $S(\sigma') \cup \kappa$ ne pointe vers t (c'est-à-dire $v(C) \neq t$ pour toute contrainte $C \in S(\sigma') \cup \kappa$) ; en effet, si $C \in \kappa$, alors $\sigma(v(C)) \notin \{0, \bot\}$ mais $\sigma(t) = \bot$, et si $C \in S(\sigma')$, alors on conclut en utilisant le fait que `may`(σ', t) est faux. Ainsi, $[\![S(\sigma') \cup \kappa]\!](t) = 0$, et l'on peut donc enlever de ce système toute contrainte de la forme $t \Rightarrow C$ sans en changer la classe d'équivalence pour $\simeq$. On peut ainsi obtenir $S(\sigma'') \cup \kappa$, ce qui donne bien $S(\sigma'') \cup \kappa \simeq S$. On a prouvé que (σ'', κ) est un état.

Passons à la preuve de l'assertion sur `trigger`. Nous allons faire (pour une classe d'équivalence de σ fixée) une deuxième induction (dite locale) sur la taille de la contrainte C. Nous définissons la taille d'une contrainte C comme le couple (n, m) où n est le nombre de symboles $\wedge$ qui apparaissent dans C, et m est le nombre total de symboles de C. On voit facilement que tous les appels récursifs à `trigger'` depuis `trigger'` peuvent être remplacés par un appel à `trigger` sans changer le résultat ; en effet, `trigger'` n'est appelé que sur des couples (σ, C) tels que $\sigma(v(C)) \neq 1$.

Supposons donc que $(\sigma, \kappa \cup \{C\})$ est un état. Considérons d'abord le cas $\sigma(v(C)) = 1$. Il s'agit de voir que (σ, κ) est aussi un état, c'est-à-dire que $S(\sigma) \cup \kappa \simeq S(\sigma) \cup \kappa \cup \{C\}$; cela découle du fait que $[\![S(\sigma) \cup \kappa]\!] \vDash C$ (car $v(C) \in S(\sigma)$).

À partir de maintenant, nous supposons $\sigma(v(C)) \neq 1$, et nous procédons par disjonction de cas suivant la forme de C.

Cas $C = t$: Puisque $(\sigma, \kappa \cup \{C\})$ est un état et $\sigma(v(C)) \neq 1$, on a forcément $\sigma(v(C)) = \sigma(t)$ de la forme $L = [C_1; \ldots; C_n]$. On constate que $S(\sigma) \cup \kappa \cup \{t\} = S(\sigma\{t \to 1\}) \cup \kappa \cup \{t \Rightarrow C_1, \ldots, t \Rightarrow C_n\} \simeq S(\sigma\{t \to 1\}) \cup \kappa \cup \{C_1, \ldots, C_n\}$. Ainsi, on voit que $(\sigma\{t \to 1\}, \kappa \cup \{C_1, \ldots, C_n\})$ est un état, et $\sigma\{t \to 1\}$ est strictement plus petit que σ. On conclut ce cas en appliquant n fois l'hypothèse d'induction globale (dans $\texttt{trigger}^*$).
Cas $C = t \Rightarrow C'$: Considérons d'abord le cas où $t = v(C')$. La contrainte longue C est alors vraie pour toute congruence, donc $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa$, et (σ, κ) est bien un état.

Considérons maintenant le cas général. En utilisant l'assertion sur `extend`, on obtient que $(\sigma', \kappa \cup \{t \Rightarrow C'\})$ est un état, avec $t \in \mathrm{Dom}(\sigma')$ et $\sigma' \leq \sigma$. Le filtrage est donc exhaustif. Si $\sigma'(t) = 0$, alors $[\![S]\!](t) = [\![S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\}]\!](t) = 0$, et donc $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} \simeq S(\sigma') \cup \kappa$. On voit ainsi que (σ', κ) est un état. Si $\sigma'(t) = 1$, alors $[\![S]\!](t) = [\![S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\}]\!](t) = 1$, et donc $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} \simeq S(\sigma') \cup \kappa \cup \{C'\}$. On voit ainsi que $(\sigma', \kappa \cup \{C'\})$ est un état. On peut appliquer l'hypothèse d'induction globale ou locale, suivant que σ' est strictement plus petite que σ ou non (et alors C' est strictement plus petite que C). Enfin, si $\sigma'(t) = [C_1; \ldots; C_n]$, alors $S(\sigma') \cup \kappa \cup \{t \Rightarrow C'\} = S(\sigma'\{t \to C' :: L\}) \cup \kappa$, et donc $(S(\sigma'\{t \to C' :: L\}), \kappa)$ est un état.
Cas $C = \phi_1 \wedge \phi_2 \Rightarrow C'$: On constate que $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{\phi_1 \Rightarrow (\phi_2 \Rightarrow C')\}$. Donc $(\sigma, \kappa \cup \{\phi_1 \Rightarrow (\phi_2 \Rightarrow C')\})$ est un état. De plus, la contrainte $\phi_1 \Rightarrow (\phi_2 \Rightarrow C')$ est strictement plus petite que C, avec l'ordre que l'on a choisit sur les contraintes (le nombre de symboles $\wedge$ diminue strictement). On conclut ce cas en appliquant l'hypothèse d'induction locale.
Cas $C = \phi_1 \vee \phi_2 \Rightarrow C'$: On constate que $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{\phi_1 \Rightarrow C', \phi_2 \Rightarrow C'\}$. Donc $(\sigma, \kappa \cup \{\phi_1 \Rightarrow C'\} \cup \{\phi_2 \Rightarrow C'\})$ est un état, et l'on conclut ce cas en appliquant deux fois l'hypothèse d'induction (la première fois, il s'agit de l'induction locale ; la deuxième, de l'induction locale ou de l'induction globale, suivant que σ décroit strictement ou non lors du premier appel).
Cas $C = 0 \Rightarrow C'$: On constate que $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa$, et donc (σ, κ) est un état.
Cas $C = 1 \Rightarrow C'$: On constate que $S(\sigma) \cup \kappa \cup \{C\} \simeq S(\sigma) \cup \kappa \cup \{C'\}$, et donc $(\sigma, \kappa \cup \{C'\})$ est un état. On conclut en appliquant l'hypothèse d'induction locale. □

Remarque 7.32 *Dans la preuve, on voit que l'algorithme tel qu'il est présenté inclut deux optimisations qui sont en fait facultatives :*

- *dans la définition de* `extend`*, on peut renvoyer* σ' *même si* `may` *renvoie faux ;*
- *dans la définition de* $\texttt{trigger}'(\sigma, t \Rightarrow C)$*, on peut ignorer le cas particulier* $t = v(C)$*.*

7.1.4 Application : test de vide pour un automate d'arbres

Nous donnons maintenant un exemple simple d'application de nos algorithmes de calculs de prédicats inductifs. Considérons des arbres binaires, dont les feuilles sont étiquetées par des symboles dans un alphabet fini Σ (les nœuds ne sont pas étiquetés). Un automate d'arbres est un couple (Q, R), où Q est un ensemble fini d'états, et R est un ensemble fini de transitions de la forme $a \to q$ ($a \in \Sigma$, $q \in Q$) ou $(q_1, q_2) \to q$ ($q, q_1, q_2 \in Q$). Chaque état q définit de manière classique un ensemble régulier d'arbres, que l'on note $[\![q]\!]$. Nous nous intéressons à la question de savoir si $[\![q]\!] = \emptyset$ ou non, pour un état q donné.

On voit facilement que le prédicat $[\![q]\!] \neq \emptyset$ est défini *inductivement* par :

$$[\![q]\!] \neq \emptyset \iff \begin{cases} \exists a.\ (a \to q) \in R \\ \vee \\ \exists (q_1, q_2).\ ((q_1, q_2) \to q) \in R \wedge [\![q_1]\!] \neq \emptyset \wedge [\![q_2]\!] \neq \emptyset \end{cases}$$

Cela rentre bien dans le formalisme introduit. On prend $\sqsupseteq = Q$, et :

$$\Phi(q) = \bigvee_{(a \to q) \in R} 1 \vee \bigvee_{(q_1, q_2) \to q \in R} q_1 \wedge q_2$$

Si l'automate est donné de manière descendante (*top-down*), c'est-à-dire si l'on peut obtenir pour chaque q l'ensemble des transitions de but q en temps linéaire en la taille du résultat, on peut prouver que l'algorithme sans retour-arrière donne un algorithme en temps linéaire (donc optimal) par rapport à la taille de l'automate (nombre d'état + nombre de transitions). Ce résultat est connu, mais il s'obtient en général par une méthode ascendante (saturer l'ensemble des états q tels que $[\![q]\!] \neq \emptyset$) ; l'avantage de procéder par une méthode descendante est de pouvoir ignorer certains états : ceux qui sont inaccessibles, évidemment, mais aussi, pour une transition $(q_1, q_2) \to q$, l'état q_2 lorsque l'on s'aperçoit que $[\![q_1]\!] = \emptyset$. Cela est particulièrement important lorsque l'automate n'est pas donné explicitement et que le calcul de ses états et de ses transitions est coûteux.

7.2 Algorithme de sous-typage

Voyons maintenant comment appliquer les algorithmes de la section précédent pour calculer la relation de sous-typage dans les modèles universels. Nous prenons un modèle universel $[\![\_]\!]$ et nous notons simplement $\leq$ la relation de sous-typage qu'il induit. Nous nous ramenons évidemment au test du vide, en constatant que $t_1 \leq t_2 \iff [\![t_1 \backslash t_2]\!] = \emptyset$. Le point de départ est le Lemme 4.27, que l'on peut réécrire de la manière suivante. Soit t un type et $\sqsupseteq$ un socle qui le contient. Alors :

$$[\![t]\!] \neq \emptyset \iff \forall \mathcal{S} \subseteq \sqsupseteq.\ \mathcal{S} \subseteq \mathbb{E}\mathcal{S} \Rightarrow t \notin \mathcal{S}$$

Associons à un sous-ensemble $\mathcal{S} \subseteq \sqsupseteq$ la fonction $\rho_{\mathcal{S}} : \sqsupseteq \to \{0, 1\}$ définie par $\rho_{\mathcal{S}}(t) = 1 \iff t \notin \mathcal{S}$ (fonction caractéristique du complémentaire). Pour tout $t \in \sqsupseteq$, on peut déduire (voir plus bas) de la définition de $\mathbb{E}S$ une formule booléenne $\Phi(t)$ telle que :

$$\rho_{\mathbb{E}\mathcal{S}}(t) = 1 \iff \rho_S \vDash \Phi(t)$$

On considère alors le système de contraintes :

$$S = \{\Phi(t) \Rightarrow t \mid t \in \beth\}$$

Ainsi :

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S} \iff \rho_{\mathcal{S}} \vDash S$$

et donc :

$$[\![t]\!] \neq \emptyset \iff \forall \rho \vDash S.\ \rho(t) = 1$$

c'est-à-dire :

$$[\![t]\!] \neq \emptyset \iff [\![S]\!](t) = 1$$

On peut alors utiliser l'un des algorithmes présentés à la section précédente pour calculer ce prédicat $[\![t]\!] \neq \emptyset$. L'ensemble $\beth$ peut être très grand, mais les algorithmes ne vont en considérer en général qu'une partie. De même, les formules $\Phi(t)$ sont complexes, mais dans la mesure où les algorithmes ne les considèrent pas forcément entièrement, on peut avoir intérêt à les construire de manière paresseuse.

Explicitons maintenant la définition des formules $\Phi(t)$. On suit simplement la Définition 4.11. Pour éviter la confusion entre les opérateurs logiques $\vee, \wedge$ et les connecteurs booléens sur les types $\mathsf{V}, \mathsf{\Lambda}$, nous allons noter $[t]$ au lieu de t pour dénoter la variable associée au type t (et qui correspond donc intuitivement au prédicat affirmant que le type t n'est pas vide).

$$\Phi([t]) ::= \bigvee_{\substack{(P,N)\in t \\ u \mid P \subseteq T_u}} \Phi_u(P,N)$$

$$\Phi_{\mathbf{basic}}(P,N) ::= \begin{cases} 0 & \text{si } \mathcal{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!] \\ 1 & \text{sinon} \end{cases}$$

$$\Phi_{\mathbf{prod}}(P,N) ::= \bigvee_{N' \subseteq N \cap T_{\mathbf{prod}}} \begin{cases} \left[\bigwedge_{t_1\times t_2 \in P} t_1 \setminus \bigvee_{t_1\times t_2 \in N'} t_1\right] \\ \qquad \wedge \\ \left[\bigwedge_{t_1\times t_2 \in P} t_2 \setminus \bigvee_{t_1\times t_2 \in N\setminus N'} t_2\right] \end{cases}$$

$$\Phi_{\mathbf{fun}}(P,N) ::= \bigwedge_{t'_1\to t'_2 \in N} \bigvee_{P' \subseteq P} \begin{cases} \left[t'_1 \setminus \bigvee_{t_1\times t_2 \in P'} t_1\right] \\ \qquad \wedge \\ \begin{cases} 1 & \text{si } P = P' \\ \left[\left(\bigwedge_{t_1\times t_2 \in P\setminus P'} t_2\right) \setminus t'_2\right] & \text{si } P \neq P' \end{cases} \end{cases}$$

Implémentation Nous avons choisi de présenter séparément d'une part des algorithmes génériques pour calculer des prédicats définis inductivement par des formules booléennes, et d'autre part la fonction Φ qui engendre le prédicat de sous-typage (en fait, la négation du test de vide). Une implémentation

naïve consisterait à manipuler effectivement des termes syntaxiques pour représenter les formules booléennes $\Phi([t])$, et à utiliser des implémentations génériques des algorithmes de la section précédente (avec ou sans retour-arrière). En fait, en spécialisant ces algorithmes à la définition particulière de Φ, on peut éviter de produire ces termes (qui sont immédiatement déconstruits par les algorithmes). Les formules correspondent alors à des points de contrôle de l'implémentation. Pour l'algorithme sans retour-arrière, cependant, il faut parfois stocker des contraintes dans la table σ ; en pratique, une telle contrainte peut-être représentée comme une fermeture obtenue par l'application partielle de `trigger` à la contrainte.

7.3 Variante et optimisation

7.3.1 Autre formules

Les formules obtenues à la section précédente, dérivées de la définition d'une simulation, proviennent *in fine* du résultat ensembliste énoncé dans le Lemme 4.6. Nous allons maintenant montrer comment, partant d'une propriété ensembliste différente, nous pouvons obtenir un algorithme potentiellement plus efficace.

Lemme 7.33 *Soient $X, X' \subseteq D_1$, $Y, Y' \subseteq D_2$. Alors :*

$$(X \times Y) \backslash (X' \times Y') = ((X \backslash X') \times Y) \cup (X \times (Y \backslash Y'))$$

Considérons maintenant un type t, et $(P, N) \in t$, avec $P \subseteq T_{\mathbf{prod}}$. On écrit :

$$P = \{t_1^1 \mathbf{\times} t_2^1; \ldots; t_1^n \mathbf{\times} t_2^n\}$$

$$N \cap T_{\mathbf{prod}} = \{s_1^1 \mathbf{\times} s_2^1; \ldots; s_1^m \mathbf{\times} s_2^m\}$$

Posons $t_1 = \bigwedge_{i=1..n} t_1^i$ et $t_2 = \bigwedge_{i=1..n} t_2^i$. On s'intéresse à la question de savoir si l'assertion ci-dessous est vraie :

$$(*) \quad [\![t_1]\!] \times [\![t_2]\!] \subseteq \bigcup_{i=1..m} [\![s_1^i]\!] \times [\![s_2^i]\!]$$

Clairement, si $[\![t_1]\!] = \emptyset$ ou $[\![t_2]\!] = \emptyset$, alors l'assertion $(*)$ est vraie. Sinon, si $m = 0$, l'assertion est fausse, et si $m \neq 0$, on utilise le Lemme 7.33 (en isolant le premier terme de la réunion), et l'on obtient que $(*)$ est vraie si et seulement si les deux assertions ci-dessous le sont :

$$\begin{cases} [\![t_1']\!] \times [\![t_2]\!] \subseteq \bigcup_{i=2..m} [\![s_1^i]\!] \times [\![s_2^i]\!] \\ [\![t_1]\!] \times [\![t_2']\!] \subseteq \bigcup_{i=2..m} [\![s_1^i]\!] \times [\![s_2^i]\!] \end{cases}$$

où $t_1' = t_1 \backslash s_1^1$ et $t_2' = t_2 \backslash s_2^1$. Or chacune de ces deux assertions à la même forme que $(*)$. On peut donc poursuivre cette décomposition pour chacun des produits $s_1^i \mathbf{\times} s_2^i$. Cela donne une définition alternative de l'ensemble $\mathbb{E}\mathcal{S}$, et donc des formules $\Phi(t)$ différentes :

$$\Phi_{\mathbf{prod}}(P, N) ::= [t_1] \wedge [t_2] \wedge \Phi'_{\mathbf{prod}}(t_1, t_2, N \cap T_{\mathbf{prod}})$$

où :

$$t_1 = \bigwedge_{t_1 \times t_2 \in P} t_1$$

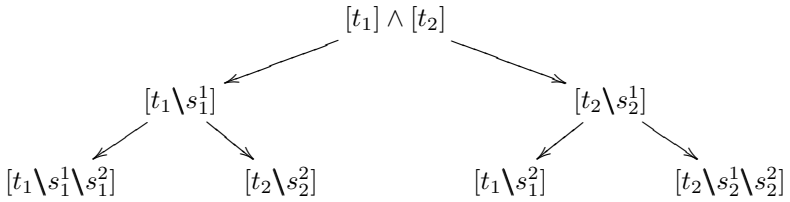
$$t_2 = \bigwedge_{t_1 \times t_2 \in P} t_2$$

et les formules $\Phi'_{\mathbf{prod}}(t_1, t_2, N)$ sont définies par :

$$\begin{array}{lcl} \Phi'_{\mathbf{prod}}(t_1, t_2, \emptyset) & ::= & 1 \\ \Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N') & ::= & \left\{ \begin{array}{c} [t_1 \backslash s_1] \wedge \Phi'_{\mathbf{prod}}(t_1 \backslash s_1, t_2, N') \\ \vee \\ [t_2 \backslash s_2] \wedge \Phi'_{\mathbf{prod}}(t_1, t_2 \backslash s_2, N') \end{array} \right. \end{array}$$

(le choix de $s_1 \times s_2$ dans N peut se faire de manière arbitraire)

On peut voir ces formules comme des arbres binaires. Ainsi pour $N = \{s_1^1 \times s_2^1; s_1^2 \times s_2^2\}$, l'arbre est :



La formule booléenne correspondante peut s'interpréter comme l'existence d'une branche dont tous les nœuds sont vrais. On voit immédiatement que ces nouvelles formules permettent un élagage important : dès qu'un nœud est faux, il n'est pas nécessaire de considérer le sous-arbre correspondant (et les algorithmes de la section précédente vont bien tenir compte cet élagage).

Sur cette présentation, on peut introduire une optimisation supplémentaire, motivée par la propriété ensembliste ci-dessous, qui complète le Lemme 7.33.

Lemme 7.34 *Soient $X, X' \subseteq D_1$, $Y, Y' \subseteq D_2$. Si l'on a $X \cap X' = \emptyset$ ou $Y \cap Y' = \emptyset$, alors :*

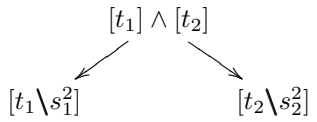
$$(X \times Y) \backslash (X' \times Y') = (X \times Y)$$

Cela suggère de prendre :

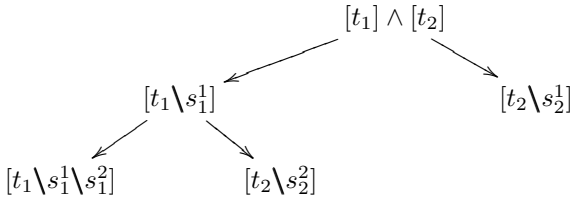
$$\Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N') ::= \Phi'_{\mathbf{prod}}(t_1, t_2, N')$$

au lieu de la formule générique pour $\Phi'_{\mathbf{prod}}(t_1, t_2, \{s_1 \times s_2\} \cup N')$, lorsque que l'on sait déjà que $[\![t_1 \wedge s_1]\!] = \emptyset$ ou que $[\![t_2 \wedge s_2]\!] = \emptyset$, en s'appuyant sur les hypothèses courantes des algorithmes (pour l'algorithme avec cache, en considérant l'ensemble des hypothèses négatives, et pour l'algorithme sans retour-arrière, en considérant la table σ).

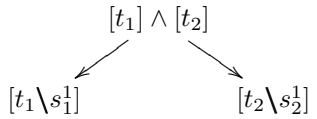
Par exemple, reprenons l'exemple avec $N = \{s_1^1 \times s_2^1; s_1^2 \times s_2^2\}$. Si l'on sait déjà (ou si l'on est en train de supposer) que, disons, $[\![t_1 \wedge s_1^1]\!] = \emptyset$, alors l'arbre de décision devient :



De même, si l'on sait que $[\![t_1 \wedge s_1^2]\!] = \emptyset$, ou que $[\![(t_2 \backslash s_2^1) \wedge s_2^2]\!] = \emptyset$, alors l'arbre de décision devient :



Notons que dans le cas $[\![t_1 \wedge s_1^2]\!] = \emptyset$, il aurait été plus judicieux de considérer le produit $s_1^2 \times s_2^2$ avant $s_1^1 \times s_2^1$, ce qui aurait donné l'arbre de décision :



De manière générale, même sans cette optimisation, il est préférable de considérer d'abord les produits de N qui sont les plus « gros » ensemblistement, de sorte à couper plus haut le parcours des arbres de décision. On peut ainsi envisager l'utilisation d'heuristiques (qui estiment rapidement une certaine mesure de la « taille » des types) pour décider dans quel ordre considérer les produits dans N. Nous n'avons pas obtenu de résultats dans cette direction.

Finalement, remarquons que tout ce qui a été dit dans cette section s'applique également *mutatis mutandis* aux formules $\Phi_{\mathbf{fun}}$, qui proviennent aussi du Lemme 4.6 (via le Lemme 4.9).

7.3.2 Approximations

Nous avons vu comment modifier les formules booléennes qui permettent de décider si un type est vide ou non, pour optimiser l'algorithme de sous-typage. Une autre approche consiste à chercher des approximations, c'est-à-dire des conditions nécessaires et des conditions suffisantes, pour qu'un type soit vide. Cela peut permettre de court-circuiter l'algorithme générique dans des cas simples et typiques. Supposons que l'on dispose de deux systèmes de contraintes sur les variables $[t]$ (pour t dans un socle donné), disons, S_0 et S_1 qui approximent la relation de sous-typage :

$$[\![S_0]\!](t) = 0 \Rightarrow [\![t]\!] = \emptyset$$

$$[\![S_1]\!](t) = 1 \Rightarrow [\![t]\!] \neq \emptyset$$

On suppose que ces systèmes sont donnés par des formules :

$$S_i = \{\Phi_i(t) \Rightarrow t \mid t \in \beth\}$$

Si les prédicats $[\![S_0]\!]$ et $[\![S_1]\!]$ sont nettement plus faciles à calculer que $[\![S]\!]$, on peut remplacer $\Phi(t)$ par :

$$\Phi'(t) = [\![S_1]\!](t) \vee ([\![S_0]\!](t) \wedge \Phi(t))$$

Ainsi, on n'utilise les formules complexes pour $\Phi(t)$ seulement lorsque les approximations ne sont pas suffisantes pour vérifier si $[\![t]\!] = \emptyset$.

Éventuellement, on peut imbriquer le calcul de $[\![S_0]\!]$, $[\![S_1]\!]$ et $[\![S]\!]$, en utilisant des systèmes mutuellement récursifs (formellement, chaque type t donne lieu à trois variables $[t], [t]_0, [t]_1$, et l'on utilise une unique fonction des variables dans les formules booléennes).

Voici un exemple typique d'approximation. Considérons l'instance de sous-typage :

$$t \times s \leq \bigvee_{i=1..n} t_i \times s_i \quad (*)$$

Clairement, pour avoir $(*)$, il suffit d'avoir $t \leq t_{i_0}$ et $s \leq s_{i_0}$ pour un certain i_0. Inversement, une condition nécessaire pour avoir $(*)$ est d'avoir soit $t \leq \mathbb{0}$, soit la conjonction de $t \leq \bigvee_{i=1..n} t_i$ et de $s \leq \bigvee_{i=1..n} s_i$. Ces approximations permettent de définir des systèmes de contraintes S_0 et S_1, dont les plus petits points fixes sont plus faciles à calculer que celui de S.

Chapitre 8

Compilation du filtrage

Dans ce chapitre, nous étudions l'implémentation du filtrage (Chapitre 6). Il s'agit, étant donné une valeur v et un motif p, de calculer efficacement le résultat v/p du filtrage de v par p. Nous nous plaçons dans un cadre de compilation : nous connaissons p à l'avance, ce qui permet d'effectuer certains calculs, éventuellement coûteux, en vue de préparer l'évaluation efficace du filtrage v/p pour des valeurs v disponibles lors de l'exécution.

Considérons l'évaluation de la construction $\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid p_2 \to e_2$. Lorsque e a été réduit à une valeur e, il faut, si l'on suit la sémantique de cette construction, calculer v/p_1, puis éventuellement v/p_2. Nous allons développer des techniques pour évaluer efficacement en parallèle plusieurs motifs sur la même valeur.

Nous nous intéressons principalement à l'aspect structurel du filtrage, c'est-à-dire que nous supposons traitée l'évaluation efficace des tests de type sur des types t « simples », c'est-à-dire tels que $t \wedge \mathbb{1}_{\mathbf{prod}} \simeq \mathbb{0}$ (où $\mathbb{1}_{\mathbf{prod}} = \mathbb{1} \times \mathbb{1}$). Nous voyons les valeurs comme des arbres binaires dont les feuilles sont des constantes ou des abstractions (c'est-à-dire des éléments de $[\![\mathbb{1} \setminus \mathbb{1}_{\mathbf{prod}}]\!]$). Les nœuds internes (et donc les sous-arbres) correspondent aux sous-valeurs. Pour une valeur $v = (v_1, v_2)$, nous voyons v_1 (resp. v_2) comme le sous-arbre gauche (resp. droit). Une valeur est dite simple si elle n'est pas un couple.

Tous les motifs que nous allons considérer dans ce chapitre seront supposés être en forme normale, au sens de la Section 6.5.2. En particulier, tous les types qui apparaissent dans les motifs sont simples. Nous fixons une G'-base B (toujours au sens de la Section 6.5.2), et nous nous intéressons uniquement aux motifs $\sigma(q)$ pour $q \in B$, et à leurs sous-motifs. Il n'y a qu'un nombre fini de tels motifs.

On note v/q au lieu de $v/\sigma(q)$, $\lbag q \rbag$ au lieu de $\lbag \sigma(q) \rbag$, et $\mathrm{Var}(q)$ au lieu de $\mathrm{Var}(\sigma(q))$.

8.1 Évaluation naïve

La sémantique du filtrage (Figure 6.1) donne directement un algorithme naïf pour évaluer le filtrage, par induction sur la couple (v, p) ordonné lexicographiquement. Pour calculer $v/p_1|p_2$, on commence par calculer v/p_1, et on ne calcule v/p_1 que si le résultat est Ω. De même, pour calculer $(v_1, v_2)/(q_1, q_2)$, on com-

mence, par exemple, par calculer $v_1/\sigma(q_1)$ et on ne calcule $v_2/\sigma(q_2)$ que si le résultat n'est pas Ω.

8.2 Idées d'optimisation

Ignorer un sous-arbre Une première optimisation par rapport à l'algorithme naïf consiste à détecter les motifs p tels que $\lbag p \rbag \simeq \mathbb{1}$ et $\mathrm{Var}(p) \simeq \emptyset$; lorsque l'on évalue v/p pour un tel motif p, on peut complètement ignorer le sous-arbre v et renvoyer directement le résultat $\{\}$.

Ordonnancer les calculs Dans le calcul de $v/p_1|p_2$, le choix de commencer par p_1 est assez naturel compte-tenu de la politique *first-match* (si p_1 réussit, il n'est pas nécessaire de considérer p_2). Néanmoins, il est parfois préférable de commencer par p_2, par exemple si $\lbag p_1 \rbag \wedge \lbag p_2 \rbag \simeq \mathbb{0}$ (car alors si p_2 réussit, il n'est pas nécessaire de considérer p_1) et si le calcul de v/p_2 est plus rapide que v/p_1 (ou si l'on observe statistiquement que v/p_2 réussit plus souvent que v/p_1).

Memoization L'algorithme naïf esquissé à la section précédente peut être amené à calculer plusieurs fois v/p pour la même sous-valeur v et le même motif p (et même un nombre non borné de fois, ce qui donne une complexité exponentielle en la taille de v). Nous pouvons éviter ces calculs inutiles avec une technique de *memoization* (garder trace des calculs déjà effectués). Cela garantit un temps d'évaluation linéaire en la taille de la valeur v (car le nombre de motifs différents est fini).

Déterminisation En fait, on peut imposer de ne parcourir chaque sous-valeur qu'une seule fois au maximum. Pour cela, il suffit de calculer en parallèle tous les v/q pour $q \in B$ en partant des feuilles de v. Connaissant les deux familles $(v_1/q)_{q\in B}$ et $(v_2/q)_{q\in B}$, on peut facilement calculer $((v_1, v_2)/q)_{q\in B}$. Cela correspond à l'algorithme classique de déterminisation des automates d'arbres ascendants. Cette technique d'évaluation est évidemment très coûteuse, car on doit parcourir tous les nœuds de l'arbre (on ne peut pas ignorer un sous-arbre), et effectuer les mêmes calculs partout, alors que pour une valeur (v_1, v_2) et un motif (q_1, q_2), on n'a besoin que de v_1/q_1 et de v_2/q_2, pas de v_1/q_2 et de v_2/q_1. On effectue donc beaucoup de calculs inutiles.

Déterminisation descendante On peut raffiner la technique de déterminisation, en calculant en parallèle non pas tous les v/q, mais seulement un certain nombre. Si l'on doit calculer $(v/q)_{q\in R}$ pour $R \subseteq B$, et que $v = (v_1, v_2)$, on calcule (à la compilation) deux ensembles $R_1 \subseteq B$ et $R_2 \subseteq B$ tels que $(v/q)_{q\in R}$ puisse se calculer en n'utilisant que $(v_1/q)_{q\in R_1}$ et $(v_2/q)_{q\in R_2}$. Pour minimiser la quantité de calcul à effectuer sur chaque sous-arbre, on cherche à prendre R_1 et R_2 aussi petits que possible. Par exemple, si $R = \{q_0\}$, $\sigma(q_0) = (q_1, q'_1)| \ldots |(q_n, q'_n)$, alors on peut prendre $R_1 = \{q_i \mid i = 1..n\}$, et $R_2 = \{q'_i \mid i = 1..n\}$.

Propagation latérale Reprenons l'idée du paragraphe précédent. Comme les calculs de $(v_1/q)_{q\in R_1}$ et de $(v_2/q)_{q\in R_2}$ vont s'évaluer séquentiellement, on peut faire dépendre, disons, R_2 du résultat de $(v_1/q)_{q\in R_1}$ (plutôt, du fait que chaque

v_1/q vaut Ω ou non). Dans l'exemple du paragraphe précédent, on peut prendre $R_1 = \{q_i \mid i = 1..n\}$, et $R_2 = \{q'_i \mid v_1/q_i \neq \Omega\}$. Évidemment, on pourrait faire le choix symétrique (commencer par v_2, et faire dépendre R_1 du résultat).

Déterminiser ? Même en appliquant les idées des deux paragraphes précédents, la déterminisation peut forcer à effectuer des calculs inutiles. Si $\sigma(q_0) = (q_1, q'_1)|(q_2, q'_2)$, il peut être préférable, plutôt que d'évaluer q_1 et q_2 en parallèle sur le sous-arbre gauche, de commencer par q_1, et en cas de réussite, de continuer avec q'_1 sur le sous-arbre droit. On ne considère q_2 seulement en cas d'échec d'un de ces deux calculs, ce qui peut être plus rapide. Il y a une part d'heuristique dans le choix d'une approche ou de l'autre (déterminisation ou calcul séquentiel), car le meilleur choix dépend des données effectivement présentes à l'exécution.

Prise en compte des types Nous n'avons pas besoin en général d'évaluer v/p pour n'importe quelle valeur v. En effet, le système de type statique du langage donne un type pour les entrées du filtrage. On peut supposer que v est dans un certain type t_0. Cette information statique peut permettre d'éviter certains calculs inutiles. Par exemple, pour une feuille, on peut éviter un test de type t (type simple) si l'information de type t_0 à ce niveau est telle que $t_0 \leq t$ (le test réussit nécessairement). Cela se généralise au cas d'un motif arbitraire p tel que $t_0 \leq \lfloor p \rfloor$ et $\mathrm{Var}(p) = \emptyset$. De même, si $t_0 \wedge \lfloor p \rfloor \simeq \mathbb{0}$, on peut complètement ignorer le sous-arbre correspondant car le résultat sera nécessairement Ω.

Lorsque l'on évalue $v/p_1|p_2$ avec une information statique, indépendamment de l'ordre d'évaluation (entre p_1 et p_2) et d'un éventuel calcul en parallèle (déterminisation), on peut supposer, pour le calcul de p_2 une information statique $t_0 \setminus \lfloor p_1 \rfloor$, qui permet potentiellement de simplifier ce calcul (par rapport à la seule information statique t_0). En effet, si l'hypothèse supplémentaire que l'on a faite (à savoir que la valeur est dans $\neg \lfloor p_1 \rfloor$) s'avère fausse, le résultat de ce calcul sera de toute manière ignoré, et il est donc acceptable de renvoyer un résultat faux pour v/p_2.

On interprète donc l'information statique t_0 non pas comme une garantie que la valeur v est dans $[\![t_0]\!]$, mais comme le contrat qui demande un résultat correct seulement lorsque cette condition est réalisée (et qui autorise n'importe quel résultat dans le cas contraire).

Pour compiler le filtrage `match` e `with` $p_1 \to e_1 \mid p_2 \to e_2$ avec un type d'entrée[1] t_0, on peut calculer p_1 et p_2 en parallèle (ou non) ; pour p_1, l'information statique est t_0 et pour p_2, on prend $t_0 \setminus \lfloor p_1 \rfloor$.

Propagation des types L'information statique se propage aux sous-valeurs. Ainsi, si l'on sait que $v = (v_1, v_2)$ est dans t_0, on sait aussi que v_1 est dans $\pi_1[t_0 \wedge \mathbb{1}_{\mathbf{prod}}]$ et que v_2 est dans $\pi_2[t_0 \wedge \mathbb{1}_{\mathbf{prod}}]$.

Au cours de l'évaluation du filtrage, les calculs effectués permettent de gagner des informations sur le type de la valeur et des ses sous-valeurs, et donc de raffiner l'information statique. Ainsi, si l'on effectue un calcul v_1/q_1 sur le sous-arbre gauche, et que ce calcul échoue, on sait que la valeur $v = (v_1, v_2)$ est non

[1] Le filtrage peut être typé plusieurs fois par l'algorithme de typage, s'il se trouve à l'intérieur d'une fonction surchargée. Dans ce cas, le type t_0 est la réunion de tous les types calculés pour l'expression e.

seulement de type t_0 (information *a priori*), mais aussi de type $\neg(\lceil q_1 \rfloor \times \mathbb{1})$, ce qui peut être utilisé pour des calculs ultérieurs sur v (et v_1). Si le calcul v_1/q_1 réussit, on sait que v est aussi de type $\lceil q_1 \rfloor \times \mathbb{1}$, ce qui donne éventuellement en retour une information plus précise pour v_2.

Factorisation des captures Dans certains cas, pour calculer v/p, on peut complètement ignorer v. Nous avons vu que c'est le cas si l'information statique t_0 dont on dispose sur v est telle que $t_0 \wedge \lceil p \rfloor \simeq \mathbb{0}$ (le résultat est alors Ω), ou telle que $t_0 \leq \lceil p \rfloor$ et $\mathrm{Var}(p) = \emptyset$ (le résultat est alors $\{\}$). Lorsque $t_0 \leq \lceil p \rfloor$, mais que $\mathrm{Var}(p) \neq \emptyset$, on peut parfois encore calculer directement le résultat, par exemple si p est de la forme $x_1 \& x_2 \& \ldots x_n \& p_0$, avec $\mathrm{Var}(p_0) = \emptyset$ (ou en remplaçant x_i par $(x_i := c_i)$). Il peut être nécessaire de réécrire le motif p pour le mettre sous cette forme. Par exemple, si $\sigma(q) = (q, q) | x$, alors $\sigma(q) \simeq x$. Éventuellement, il faut prendre en compte le type t_0. Par exemple, si $p = (q_0, q_1)$ avec $\sigma(q_0) = x$, $\sigma(q_1) = (x := c)$ et $t_0 = \mathbb{1} \times b_c$, alors on peut remplacer p par x.

Même si $t_0 \not\leq \lceil p \rfloor$, il peut être souhaitable de « remonter » ainsi les captures dans les motifs, pour simplifier les calculs dans les sous-arbres. Par exemple, si $\sigma(q_0) = (q_1, q_0) | ((x := c) \& b_c)$, $\sigma(q_1) = x \& t_1$, alors on peut remplacer $\sigma(q_0)$ par $x \& q_0'$ où $\sigma(q_0') = (q_1', q_0') | b_c$, $\sigma(q_1') = t_1$. On évite ainsi de devoir reconstruire des couples qui existent déjà.

Fusionner des motifs Considérons un motif alternatif $(q_1, q_2) | (q_3, q_4)$. Si q_1 et q_3 sont équivalents, au moins étant donné l'information statique dont on dispose (c'est-à-dire que pour toute valeur dans $\pi_1[t_0]$, ils renvoient le même résultat), alors on peut fusionner les termes du motif alternatif en (q_1, q) avec $\sigma(q) = q_2 | q_4$. Éventuellement, calculer un résultat pour q peut être plus simple que de calculer un résultat pour q_2 et un autre pour q_4, comme on pourrait être tenté de le faire avec le motif de départ. Par exemple, si $\sigma(q_2)$ et $\sigma(q_4)$ n'ont pas de variables et $\lceil \sigma(q_2) \rfloor = \neg \lceil \sigma(q_4) \rfloor$, alors calculer un résultat pour q est trivial, tandis que calculer un résultat même pour q_2 seulement peut être compliqué.

8.3 Formalisation

Nous n'allons pas donner un algorithme figé pour compiler les motifs. En effet, nous avons vu, dans la discussion de la section précédente, que certains choix doivent être fait de manière heuristique. Pour pouvoir expérimenter plusieurs heuristiques, éventuellement dynamiquement ou en utilisant des informations statistiques collectées lors d'exécutions précédentes (*profiling*), nous allons mettre en place un formalisme qui permettra d'exprimer les différentes optimisations et stratégies présentées plus haut.

Nous fixons toujours une G'-base B (au sens de la Section 6.5.2). Nous fixons également un socle $\sqsupseteq$, qui va nous permettre d'exprimer les informations statiques t_0. Nous supposons que $\sqsupseteq$ contient $\mathbb{1}_{\mathbf{prod}}$, ainsi que tous les $\lceil p \rfloor$ pour les sous-motifs p des $\sigma(q)$ $(q \in B)$.

Nous notons $p \xrightarrow{t_0} x$ le prédicat :

$$(x \in \mathrm{Var}(p)) \wedge (\forall v \in [\![t_0 \wedge \lceil p \rfloor]\!].\ (v/p)(x) = v)$$

et $p \xrightarrow{t_0} (x := c)$ le prédicat :

$$(x \in \mathrm{Var}(p)) \wedge (\forall v \in [\![t_0 \barwedge \wr p \int]\!].\ (v/p)(x) = c)$$

Nous donnerons dans la Section 8.4 des algorithmes pour calculer ces prédicats, qui servent à exprimer qu'une variable de capture (ou un motif constante) peut être factorisée à l'extérieur d'un motif.

8.3.1 Requêtes

Une requête est une spécification de calcul à effectuer sur une valeur. Dans le cas de l'évaluation naïve, nous pouvons identifier une requête à un nœud de motif q. Pour pouvoir effectuer plusieurs calculs en parallèle (déterminisation), nous devons autoriser des requêtes avec plusieurs nœuds. Pour pouvoir simplifier les calculs en tenant compte d'une hypothèse sur le type de la valeur d'entrée, nous associons à chaque nœud q de la requête un type t_0. Nous pouvons avoir des types différents pour chaque nœuds de la requête. En effet, ainsi que nous l'avons vu, ce type t_0 ne représente pas une contrainte forte sur la valeur en entrée : la valeur peut ne pas être dans t_0, mais alors nous sommes libres de renvoyer n'importe quel résultat pour le calcul associé. Enfin, pour pouvoir factoriser les captures, on peut se contenter de la restriction d'un nœud q à un certain ensemble de variables $X \subseteq \mathrm{Var}(q)$.

Definition 8.1 (Requêtes) *Un* **bon triplet** *est un triplet (p, t_0, X) où p est un motif construit sur B, $t_0 \in \beth$ et $X \subseteq Var(p)$.*

Une **requête atomique** *r est un triplet (q, t_0, X) avec $q \in B$, $t_0 \in \beth$, et $X \subseteq Var(q)$. On note $Var(r) = X$ et $\sigma(r)$ le bon triplet $(\sigma(q), t_0, X)$.*

Une **requête** *R est un ensemble fini de requêtes atomiques.*

Lemme 8.2 *Il n'y a qu'un nombre fini de requêtes.*

Preuve: Les ensembles B et $\beth$ sont finis. □

8.3.2 Résultats

Definition 8.3 *Un* **résultat atomique** *$\mathbb{r}$ pour un ensemble fini X de variables est soit une fonction de X dans les valeurs, soit Ω. On note parfois $\mathbb{r}^X$ pour rappeler le support de $\mathbb{r}$.*

Si $\mathbb{r}$ est un résultat pour X et $X' \subseteq X$, on note $\mathbb{r}_{|X'}$ la restriction de $\mathbb{r}$ à X' ($\Omega_{|X'} = \Omega$, $\gamma_{|X'} = \{x \mapsto \gamma(x) \mid x \in X'\}$).

Un résultat atomique pour une requête atomique r est un résultat atomique pour $Var(r)$.

Un **résultat** *$\mathbb{R}$ pour une requête R est une famille $\mathbb{R} = (\mathbb{r}_r)_{r \in R}$ de résultats atomiques pour ses composantes. On pose aussi $Dom(\mathbb{R}) = R$.*

L'unique résultat atomique pour $R = \emptyset$ est noté $\mathbb{R}^\emptyset$.

Definition 8.4 (Résultats corrects) *Un résultat atomique $\mathbb{r}$ pour X est dit* **correct** *pour un bon triplet (p, t_0, X) si :*

$$v \in [\![t_0]\!] \Rightarrow \mathbb{r} = (v/p)_{|X}$$

On note alors $v \Vdash \mathbb{r} : (p, t_0, X)$.

On définit la correction d'un résultat atomique $\mathbb{r}$ *pour une requête* r *et une valeur* v *par :*

$$v \Vdash \mathbb{r} : r \iff v \Vdash \mathbb{r} : \sigma(r)$$

On définit la correction d'un résultat $\mathbb{R} = (\mathbb{r}_r)_{r \in R}$ *pour une requête* R *et une valeur* v *par :*

$$v \Vdash \mathbb{R} \iff \forall r \in R.\ v \Vdash \mathbb{r}_r : r$$

Le lemme ci-dessous est trivial à établir.

Lemme 8.5 *Pour toute valeur* v *et toute requête atomique* r *ou toute requête* R*, il existe toujours (au moins) un résultat correct pour* v*.*

Nous cherchons à calculer efficacement *un* résultat correct pour une requête R et une valeur v données.

Definition 8.6 *Si* $\mathbb{R}_1$ *et* $\mathbb{R}_2$ *sont deux résultats, on note* $\mathbb{R}_1 \cup \mathbb{R}_2$ *le résultat de domaine* $Dom(\mathbb{R}_1) \cup Dom(\mathbb{R}_2)$ *défini par :*

$$(\mathbb{R}_1 \cup \mathbb{R}_2)_r = \begin{cases} (\mathbb{R}_1)_r & si\ r \in Dom(\mathbb{R}_1) \backslash Dom(\mathbb{R}_2) \\ (\mathbb{R}_2)_r & si\ r \in Dom(\mathbb{R}_2) \end{cases}$$

8.3.3 Compilation des requêtes

La propriété de finitude du nombre de requêtes nous permet de nous placer dans le cadre de la compilation : chaque requête R va donner lieu, dans le code généré, à une fonction qui associe à une valeur v un résultat correct de R sur v. Pour ce faire, elle peut appliquer, un nombre fini de fois, des requêtes sur les *sous-valeurs* v_1 et v_2 lorsque $v = (v_1, v_2)$. Cela correspond à des appels récursifs entre les différentes fonctions associées aux requêtes, et la terminaison est garantie (car les valeurs sont des arbres finis). Globalement, on peut voir l'évaluation d'un filtrage sur une valeur comme le parcours d'un arbre (la valeur) par un automate d'arbres, dont les états correspondent aux requêtes.

Évidemment, le nombre de requêtes, même s'il est fini, est gigantesque (exponentiel en le produit du cardinal de B, du cardinal de $\beth$, et de l'exponentielle du nombre de variables), mais en pratique, nous ne générons que le code correspondant aux requêtes potentiellement utilisées, et ce nombre est raisonnable, en général.

Le code chargé d'évaluer les requêtes a la forme (en pseudo-ML) :

```
let rec eval_{R_1} v = ...
    and eval_{R_2} v = ...
    ...
    and eval_{R_n} v = ...
```

Nous pouvons introduire une certaine dose de memoization pour les fonctions associées aux requêtes, ou certaines d'entre elles (c'est un choix heuristique). Il s'agit simplement de garder trace des résultats déjà calculés, et si une requête doit être évaluée de nouveau sur une valeur déjà considérée, elle peut renvoyer le résultat déjà calculé.

Il reste à définir ce que sont les ... dans le pseudo-code ci-dessus, c'est-à-dire la manière d'évaluer une requête R sur une valeur v. On peut distinguer

deux cas. Tout d'abord, il peut être possible de calculer un résultat correct sans considérer la forme de la valeur v. C'est le cas si pour toutes les requêtes atomiques $r = (q, t_0, X)$ dans R, on a :

- soit $t_0 \wedge \lceil q \rfloor \simeq \mathbb{0}$;
- soit $t_0 \leq \lceil q \rfloor$ et toutes les variables $x \in \mathrm{Var}(q)$ se factorisent ($\sigma(q) \xrightarrow{t_0} x$ ou $\sigma(q) \xrightarrow{t_0} (x := c)$ pour une certaine constante c).

Dans les deux cas, on peut calculer un résultat correct pour r sans considérer v.

Sinon, il faut distinguer suivant la forme de v : constante, abstraction ou couple. Un pseudo-code pour la fonction chargée d'évaluer la requête R aura alors la forme :

```
eval_R v = match v with
 | (v1,v2) -> ... (* Code pour les couples *)
 | c -> ... (* Code pour les constantes *)
 | f -> ... (* Code pour les abstractions *)
```

Si v est une valeur simple (constante ou abstraction), il faut effectuer un certain nombre de tests de type sur v, suivant les motifs t qui apparaissent dans les $\sigma(q)$ (pour $(q, t_0, X) \in R$). On voit t comme une combinaison booléennes d'atomes : $t \simeq \bigvee_{(P,N)\in t}(\bigwedge_{a\in P} a)\wedge(\bigwedge_{a\in B}\neg a)$. En utilisant le fait que $v \in [\![t_1 \vee t_2]\!] \iff (v \in [\![t_1]\!]) \vee (v \in [\![t_2]\!])$ et les propriétés similaires pour $t_1 \wedge t_2$ et $\neg t$, on se ramène à des tests avec des types atomiques a. Il est possible de prendre en compte les informations statiques des requêtes atomiques pour optimiser ces tests (on peut savoir qu'ils réussissent ou qu'ils échouent, seulement en regardant l'information statique t_0). Si v et a n'ont pas le même genre, le test échoue. Si v est une constante et a est un type de base atomique, on peut utiliser diverses techniques de *dispatch*, suivant les types de base considérés (par exemple, des arbres de recherche, des *trie*, ...). Si v est une abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$ et a est un type flèche, on utilise l'équivalence :

$$v \in [\![a]\!] \iff \bigwedge_{i=1..n} t_i \to s_i \leq a \qquad (*)$$

Ainsi, on a seulement besoin d'extraire l'interface de la fonction. Si l'on travaille dans une hypothèse de monde fermé, on connaît l'ensemble des abstractions dans le programme et l'ensemble des tests de type que l'on peut avoir à faire sur les interfaces de ces abstractions. Il est possible de précalculer les tests $(*)$ pour toutes les abstractions et tous les tests de type sur des types flèche. Cela permet de ne conserver par exemple qu'un entier à l'exécution pour représenter l'interface d'une fonction et d'éviter des tests de sous-typage.

Dans la suite de ce chapitre, nous nous intéressons uniquement au cas d'une valeur couple $v = (v_1, v_2)$. On doit alors effectuer certains calculs sur les sous-valeurs v_1 et v_2, c'est-à-dire appliquer dessus des sous-requêtes R' (en appelant les fonctions $\texttt{eval}_{R'}$). Chaque sous-requête permet de collecter certaines informations pertinentes pour le calcul à effectuer. Lorsque suffisamment d'information est disponible, on peut enfin calculer le résultat pour la requête R. Le choix des sous-requêtes à évaluer sur les sous-valeurs se fait séquentiellement : si la première sous-requête ne peut dépendre que de R, le choix des sous-requêtes suivantes peut dépendre des résultats des sous-requêtes déjà évaluées. Plus précisément, ce choix peut dépendre de la réussite ou de l'échec de chaque requête

atomique des sous-requêtes évaluées. Il faut choisir à la compilation une stratégie pour faire ce choix. Nous formalisons dans les sections suivantes un langage cible utilisé pour formaliser ces choix de sous-requêtes.

8.3.4 Langage cible : syntaxe

Definition 8.7 (Résultats statiques) *L'ensemble des* **résultats atomiques statiques** *est* $\{\Omega, \checkmark\}$*. On utilise la lettre* $\mathtt{r}$ *pour désigner un résultat statique.*

Un **résultat statique de requête** *est une fonction partielle de l'ensemble des requêtes atomiques dans* $\{\Omega, \checkmark\}^R$*. On utilise la notation* $\mathtt{R}$ *pour désigner un résultat statique de requête* $(\mathtt{r}_r)_{r\in R}$*. On pose* $Dom(\mathtt{R}) = R$*, et on écrit* $\mathtt{R} : R$ *pour signifier que* $Dom(\mathtt{R}) = R$*. On note* $\mathtt{R}^\emptyset$ *l'unique résultat statique pour la requête vide.*

La fonction $|\_|$ *associe à un résultat (atomique) un résultat (atomique) statique défini par :* $|\Omega| = \Omega$*,* $|\gamma| = \checkmark$*,* $|(\mathbb{r}_r)_{r\in R}| = (|\mathbb{r}_r|)_{r\in R}$*.*

Nous allons représenter par un couple $I = (R_1, R_2)$ les sous-requêtes évaluées sur les sous-valeurs. On pose $I^\emptyset = (\emptyset, \emptyset)$. On note $I \cup (1 : R) = (R_1 \cup R, R_2)$ et $I \cup (2 : R) = (R_1, R_2 \cup R)$.

Definition 8.8 (Langage cible) *Une* **expression cible** *pour un ensemble de variables* X *et un couple de sous-requêtes* $I = (R_1, R_2)$ *est un terme engendré par les productions :*

$$\begin{array}{rcll} \mathtt{e}^{X,I} & := & x & \text{où } X = \{x\} \\ & | & \mathbb{r}^X & \\ & | & (i, r_{|X}) & \text{où } i = 1..2, r \in R_i, X \subseteq Var(r) \\ & | & \mathtt{e}_1^{X_1,I} \oplus \mathtt{e}_2^{X_2,I} & \text{où } X = X_1 \cup X_2 \end{array}$$

où $\mathbb{r}^X$ *est un résultat atomique pour* X*,* $i = 1..2$*,* r *est une requête atomique,* X *un ensemble fini de variables.*

Une stratégie *pour une requête* R_0 *et un couple de sous-requêtes* I *est un terme fini engendré par les productions :*

$$\begin{array}{rcl} \mathtt{s}^{R_0,I} & := & (\mathtt{e}_r^{Var(r),I})_{r\in R_0} \\ & | & (i, R) \mapsto (\mathtt{s}_{\mathtt{R}}^{R_0, I\cup(i:R)})_{\mathtt{R}:R} \end{array}$$

où $i = 1..2$*,* R *est une requête, et les* $\mathtt{e}_r$ *sont des expressions cibles.*

Les exposants sont là pour assurer des contraintes de bonne formation des objets introduits. Nous les omettons parfois lorsque cela n'introduit pas d'ambiguïté.

Une stratégie représente de manière abstraite un morceau de code produit, chargé de calculer un résultat correct pour une requête R_0 sur une valeur d'entrée $v = (v_1, v_2)$. Ce code a la forme d'un arbre fini. Un nœud interne $(i, R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R}$ correspond à l'évaluation de la sous-requête R sur la valeur v_i. La famille $(\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R}$ indique comment continuer le calcul, en fonction de la réussite ou de l'échec de chaque requête atomique dans R. Les feuilles de l'arbre sont atteintes lorsque suffisamment d'information est disponible pour calculer un résultat correct pour R_0. Le résultat pour une requête atomique $r \in R_0$ est calculé par une expression cible. L'expression cible $\mathbb{r}^X$ renvoie le résultat indiqué. L'expression cible x

renvoie l'environnement $\{x \mapsto v\}$ où v est la valeur courante. L'expression cible $\mathtt{e}_1 \oplus \mathtt{e}_2$ permet de combiner deux résultats. L'expression cible $(i, r_{|X})$ utilise le résultat d'une sous-requête atomique r sur v_i, en le restreignant aux variables de X. Ce résultat doit être disponible, c'est-à-dire qu'il doit avoir été calculé par la stratégie lors de l'appel d'une sous-requête R sur v_i, avec $r \in R$.

8.3.5 Langage cible : sémantique

Nous allons formaliser la sémantique du langage cible. Il faut conserver les résultats des sous-requêtes évaluées au cours d'une stratégie. Nous définissons pour cela une notion de paquet d'information.

Definition 8.9 (Paquets) *Un* **paquet d'information** $\mathbb{I}$ *est un couple de résultats de requêtes* $(\mathbb{R}_1, \mathbb{R}_2)$. *On pose* $Dom(\mathbb{I}) = (Dom(\mathbb{R}_1), Dom(\mathbb{R}_2))$. *On note* $\mathbb{I}^\emptyset$ *le paquet d'information* $(\mathbb{R}^\emptyset, \mathbb{R}^\emptyset)$.

Definition 8.10 *On définit la correction d'un paquet d'information* $(\mathbb{R}_1, \mathbb{R}_2)$ *pour une valeur* $v = (v_1, v_2)$ *par :*

$$v \Vdash (\mathbb{R}_1, \mathbb{R}_2) \iff \forall i.\ v_i \Vdash \mathbb{R}_i$$

Si $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$ est un paquet d'information, on note $\mathbb{I} \cup (1 : \mathbb{R}) = (\mathbb{R}_1 \cup \mathbb{R}, \mathbb{R}_2)$ et $\mathbb{I} \cup (2 : \mathbb{R}) = (\mathbb{R}_1, \mathbb{R}_2 \cup \mathbb{R})$.

Lemme 8.11 *Soit* $v = (v_1, v_2)$ *une valeur et* $i = 1..2$. *Si* $v_i \vDash \mathbb{R}$ *et* $v \vDash \mathbb{I}$, *alors* $v \vDash \mathbb{I} \cup (i : \mathbb{R})$.

La Figure 8.1 introduit deux jugements, qui formalisent la sémantique du langage cible :

- Évaluation d'une stratégie : $\mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0$, où v est une valeur de la forme (v_1, v_2), $\mathrm{Dom}(\mathbb{I}) = I$, et $\mathrm{Dom}(\mathbb{R}_0) = R_0$.
- Évaluation d'une expression cible : $\mathtt{e}^{X,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}$ où $\mathbb{r}$ est un résultat pour X.

Le choix d'un résultat correct pour la sous-requête R dans la règle (*subreq*) est arbitraire. Le non-déterminisme de la sémantique provient uniquement de ces choix, et les conditions de bonne formation dans la syntaxe du langage cible garantissent l'absence d'erreur d'exécution (c'est-à-dire qu'il est toujours possible de construire une dérivation, pour n'importe quel choix de R dans la règle (*subreq*)). En particulier, dans la règle $(fetch)$, le résultat $\mathbb{R}_i$ est bien défini sur la requête atomique r.

Definition 8.12 (Correction) *On définit la correction d'une stratégie* $\mathtt{s}^{R_0,I}$ *pour un paquet d'information* $\mathbb{I}$ *(avec* $Dom(\mathbb{I}) = I$*) par :*

$$\mathbb{I} \Vdash \mathtt{s}^{R_0,I} \iff \forall v. \forall \mathbb{R}_0. \left\{ \begin{array}{ll} \mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0 & \\ v \Vdash \mathbb{I} & \end{array} \right. \quad \Rightarrow \quad v \Vdash \mathbb{R}_0$$

Lorsque $\mathbb{I} = \mathbb{I}^\emptyset$, *on s'autorise à l'omettre dans l'écriture de ce jugement.*

Intuitivement, le prédicat $\mathbb{I} \Vdash \mathtt{s}^{R_0,I}$ signifie que l'on peut utiliser la stratégie $\mathtt{s}$ pour calculer un résultat correct pour la requête R_0 sur toute valeur de la forme $v = (v_1, v_2)$, à condition de déjà disposer du paquet d'information $\mathbb{I}$ (correct pour v). Comme le paquet $\mathbb{I}^\emptyset$ est correct pour toute valeur v, le prédicat $\Vdash \mathtt{s}^{R_0,I^\emptyset}$ signifie que l'on peut utiliser la stratégie $\mathtt{s}$ pour implémenter le calcul d'un résultat correct pour R_0 (pour les valeurs qui sont des couples).

$$\frac{(\forall r \in R_0)\ \mathtt{e}_r \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_r}{(\mathtt{e}_r)_{r\in R_0} \overset{v,\mathbb{I}}{\rightsquigarrow} (\mathbb{r}_r)_{r\in R_0}}\ (assemble)$$

$$\frac{v_i \Vdash \mathbb{R} \quad \mathrm{Dom}(\mathbb{R}) = R \quad \mathtt{s}_{|\mathbb{R}|} \overset{v,\mathbb{I}\cup(i:\mathbb{R})}{\rightsquigarrow} \mathbb{R}_0}{(i,R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0}\ (subreq)$$

$$\frac{}{x \overset{v,\mathbb{I}}{\rightsquigarrow} \{x \mapsto v\}}\ (capture)$$

$$\frac{}{\mathbb{r}^X \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}^X}\ (const)$$

$$\frac{\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2) \quad \mathbb{r} = (\mathbb{R}_i)_r}{(i, r_{|X}) \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_{|X}}\ (fetch)$$

$$\frac{(\forall i)\ \mathtt{e}_i \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_i}{\mathtt{e}_1 \oplus \mathtt{e}_2 \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_1 \oplus \mathbb{r}_2}\ (compose)$$

FIG. 8.1 – Sémantique du langage cible

8.3.6 Information statique

Definition 8.13 *Un* **paquet d'information statique** $\mathtt{I}$ *est un couple de résultats statiques de requêtes* $(\mathtt{R}_1, \mathtt{R}_2)$. *On pose* $Dom(\mathtt{I}) = (Dom(\mathtt{R}_1), Dom(\mathtt{R}_2))$.

On note $\mathtt{I}^{\emptyset}$ le paquet d'information statique $(\mathtt{R}^{\emptyset}, \mathtt{R}^{\emptyset})$. Si $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$ est un paquet d'information statique, on note $\mathtt{I} \cup (1 : \mathtt{R}) = (\mathtt{R}_1 \cup \mathtt{R}, \mathtt{R}_2)$ et $\mathtt{I} \cup (2 : \mathtt{R}) = (\mathtt{R}_1, \mathtt{R}_2 \cup \mathtt{R})$.

Pour un paquet d'information $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$, on note $|\mathbb{I}|$ le paquet d'information statique $(|\mathbb{R}_1|, |\mathbb{R}_2|)$.

Lemme 8.14 *On a :*

$$|\mathbb{I} \cup (i : \mathbb{R})| = |\mathbb{I}| \cup (i : |\mathbb{R}|)$$

Un résultat correct $\mathbb{r}$ pour une valeur v sur un bon triplet (p, t_0, X) donne une certaine information sur le type de v. En effet, si le résultat est Ω, la valeur est dans $\neg \wr p \int$ et sinon, la valeur est dans $\wr p \int$. On voit que cela ne dépend que du résultat statique $|\mathbb{r}|$. En fait, cela n'est vrai que si la valeur est dans t_0, car sinon le résultat correct peut être n'importe quoi.

Definition 8.15 *Pour un bon triplet* (p, t_0, X) *et un résultat atomique statique* $\mathtt{r}$, *on pose :*

$$\wr \mathtt{r} : (p, t_0, X) \int = \begin{cases} \wr p \int \mathsf{V} \neg t_0 & si\ \mathtt{r} = \checkmark \\ (\neg \wr p \int) \mathsf{V} \neg t_0 & si\ \mathtt{r} = \Omega \end{cases}$$

et pour une requête atomique $r = (q, t_0, X)$ *:*

$$\wr \mathtt{r} : r \int = \wr \mathtt{r} : \sigma(r) \int$$

Le type associé au résultat statique $\mathtt{R} = (\mathtt{r}_r)_{r\in R}$ *d'une requête* R *est défini par :*

$$\wr\mathtt{R}\int = \bigwedge_{r\in R} \wr\mathtt{r}_r : r\int$$

Le type associé au paquet statique $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$ *est défini par :*

$$\wr\mathtt{I}\int = \wr\mathtt{R}_1\int \times \wr\mathtt{R}_2\int$$

Remarque 8.16 *Les types* $\wr\mathtt{R}\int$ *sont dans* $\beth$*. Ce n'est pas forcément le cas pour les types* $\wr\mathtt{I}\int$ *(les socles ne sont pas stables pour le constructeur* $\times$*).*

Lemme 8.17

$$\begin{array}{lcl}
(v \Vdash \mathtt{r} : (p, t_0, X)) & \Longrightarrow & v \in [\![\wr|\mathtt{r} : (p, t_0, X)|\int]\!] \\
(v \Vdash \mathtt{r} : r) & \Longrightarrow & v \in [\![\wr|\mathtt{r} : r|\int]\!] \\
(v \Vdash \mathbb{R}) & \Longrightarrow & v \in [\![\wr|\mathbb{R}|\int]\!] \\
((v_1, v_2) \Vdash \mathbb{I}) & \Longrightarrow & (v_1, v_2) \in [\![\wr|\mathbb{I}|\int]\!]
\end{array}$$

Une information statique t_0 sur une valeur $v = (v_1, v_2)$ donne des informations sur v_1 et v_2, à savoir les types $\pi_1[t_0]$ et $\pi_2[t_0]$. Si l'on tient compte d'un paquet d'information statique $\mathtt{I}$, on peut raffiner l'information t_0 en $t'_0 = t_0 \wedge \wr\mathtt{I}\int$. Ce type n'est pas forcément dans $\beth$ (Remarque 8.16). Cependant, les projections $\pi_1[t'_0]$ et $\pi_2[t'_0]$ peuvent être représentées par des types dans $\beth$.

Lemme 8.18 *Si* $t_0 \in \beth$*,* $\mathtt{I}$ *est un paquet d'information statique, et* $i = 1..2$*, alors on peut calculer un type* $\pi_i[t_0; \mathtt{I}] \in \beth$ *tel que* $\pi_i[t_0 \wedge \wr\mathtt{I}\int] \simeq \pi_i[t_0; \mathtt{I}]$

Preuve: Prenons par exemple $i = 1$. On peut supposer $t_0 \leq \mathbb{1}_{\mathbf{prod}}$ (car $\mathbb{1}_{\mathbf{prod}} \in \beth$). On écrit :

$$t_0 \simeq \bigvee_{(t_1,t_2)\in\pi(t_0)} t_1 \times t_2$$

Si $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$, et si l'on pose $t'_i = \wr\mathtt{R}_i\int$, alors $\wr\mathtt{I}\int = t'_1 \times t'_2$. On a $t'_i \in \beth$, et :

$$t'_0 := t_0 \wedge \wr\mathtt{I}\int \simeq \bigvee_{(t_1,t_2)\in\pi(t_0)} (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$$

Ainsi :

$$\pi_1[t'_0] \simeq \left(\bigvee_{(t_1,t_2)\in\pi(t_0) \ \mid \ (*)} t_1 \right) \wedge t'_1 \quad \in \beth$$

où $(*)$ est la condition $t_2 \wedge t'_2 \not\simeq \mathbb{0}$. □

Remarque 8.19 *On constate facilement que* $\pi_i[t_0; \mathtt{I}]$ *est un sous-type (parfois strict) de* $\pi_i[t_0] \wedge \wr\mathtt{R}_i\int$*, où* $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$*.*

8.3.7 Élaboration de stratégies

Nous nous intéressons à la manière de produire des stratégies correctes pour une requête donnée. Considérons tout d'abord la construction d'expressions cible.

La Figure 8.2 donne une méthode pour calculer, étant donné un bon triplet (p, t_0, X) et un paquet d'information statique $\mathtt{I}$, une expression cible $\mathtt{e}^{X,I}$ qui calcule un résultat correct pour (p, t_0, X). La figure définit un jugement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}^{X,I}$ où $I = \mathrm{Dom}(\mathtt{I})$. Elle utilise un jugement auxiliaire $\mathtt{I} \vdash_i (q, t_0, X) \Rightarrow \mathtt{e}^{X,I}$.

$$\frac{t_0 \wedge \lceil \mathtt{I} \rfloor \wedge \lceil p \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow \Omega}\ (fail)$$

$$\frac{\mathtt{I} \vdash (p_1, t_0, X) \Rightarrow \mathtt{e} \quad t_0 \wedge \lceil \mathtt{I} \rfloor \leq \lceil p_1 \rfloor}{\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}}\ (first)$$

$$\frac{\mathtt{I} \vdash (p_2, t_0, X) \Rightarrow \mathtt{e} \quad t_0 \wedge \lceil \mathtt{I} \rfloor \wedge \lceil p_1 \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}}\ (second)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lceil \mathtt{I} \rfloor} x \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\}) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow x \oplus \mathtt{e}}\ (factor)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \wedge \lceil \mathtt{I} \rfloor} (x := c) \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\}) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (p, t_0, X) \Rightarrow \{x \mapsto c\} \oplus \mathtt{e}}\ (factor)$$

$$\frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash (x \& p, t_0, X) \Rightarrow \mathtt{e}}\ (skip) \qquad \frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}}{\mathtt{I} \vdash ((x := c) \& p, t_0, X) \Rightarrow \mathtt{e}}\ (skip)$$

$$\frac{(\forall i = 1..2)\ \mathtt{I} \vdash_i (q_i, \pi_i[t_0; \lceil \mathtt{I} \rfloor], X \cap \mathrm{Var}(q_i)) \Rightarrow \mathtt{e}_i}{\mathtt{I} \vdash ((q_1, q_2), t_0, X) \Rightarrow \mathtt{e}_1 \oplus \mathtt{e}_2}\ (prod)$$

$$\frac{t_0 \leq \lceil q \rfloor}{\mathtt{I} \vdash_i (q, t_0, \emptyset) \Rightarrow \{\}}\ (succeed)$$

$$\frac{r = (q, t_0', X') \in \mathrm{Dom}(\mathtt{R}_i) \quad t_0 \leq t_0' \quad X \subseteq X'}{(\mathtt{R}_1, \mathtt{R}_2) \vdash_i (q, t_0, X) \Rightarrow (i, r_{|X})}\ (fetch)$$

FIG. 8.2 – Production d'expressions cible

Avant d'énoncer et de prouver la correction des expressions produites par ce jugement, commentons les règles. Pour un bon triplet (p, t_0, X), le type $\lceil \mathtt{I} \rfloor \wedge t_0$ est l'information statique sous laquelle il faut travailler. Si ce type n'intersecte pas $\lceil p \rfloor$, le résultat Ω est correct (si la valeur est dans $\lceil \mathtt{I} \rfloor \wedge t_0$, c'est le vrai résultat, et sinon n'importe quel résultat est correct). Les règles $(first)$ et $(second)$ permettent de traiter le cas d'un motif alternatif $p_1 | p_2$. Pour pouvoir les appliquer, il faut savoir *a priori* (c'est-à-dire, connaissant $\mathtt{I}$) si le premier motif réussit ou échoue. Les deux règles $(factor)$ traitent en particulier le cas des motifs $x \& p$ et $(x := c) \& p$, lorsque $x \in X$ (on utilise ensuite $(skip)$ lorsque l'on a éliminé x de X). Elles sont plus générales, car elles permettent de factoriser

une capture de variable (ou une liaison $(x := c)$). La règle $(prod)$ permet de traiter le cas d'un motif (q_1, q_2) On utilise le jugement auxiliaire $\vdash_i$ de chaque coté, en propageant l'information statique sur chaque composante. Ce jugement $\vdash_i$ peut renvoyer $\{\}$ pour une requête atomique (q, t_0, X) si q réussit nécessairement pour toute valeur dans t_0, et si de plus $X = \emptyset$ (ce qui peut être une conséquence de l'utilisation des règles $(factor)$). Si l'une de ces deux conditions n'est pas vérifiée, on doit utiliser une des sous-requêtes atomiques (q', t'_0, X') déjà évaluées sur v_i. On s'autorise à avoir $X \subseteq X'$ et $t_0 \leq t'_0$; cela signifie que la sous-requête que l'on utilise peut avoir effectué plus de travail que ce qui est nécessaire pour (q, t_0, X), mais ce n'est pas grave.

Le théorème suivant énonce la correction des expressions cible $\mathtt{e}$ produites par le jugement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$.

Theorème 8.20 *Pour toute valeur $v = (v_1, v_2)$ et tout paquet d'information $\mathbb{I}$, on a l'implication :*

$$\begin{cases} v \Vdash \mathbb{I} \\ |\mathbb{I}| \vdash (p, t_0, X) \Rightarrow \mathtt{e} \\ \mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r} \end{cases} \implies v \Vdash \mathbb{r} : (p, t_0, X)$$

Preuve: Soit $v = (v_1, v_2)$ et $\mathbb{I}$ tel que $v \Vdash \mathbb{I}$. Posons $\mathtt{I} = |\mathbb{I}|$. On a donc $v \in [\![\lceil \mathtt{I} \rfloor]\!]$.

On raisonne alors par induction sur la dérivation de $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. Si $v \notin [\![t_0]\!]$, l'assertion $v \Vdash \mathbb{r} : (p, t_0, X)$ est automatique. On suppose donc $v \in [\![t_0]\!]$, et nous allons prouver que $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} (v/p)_{|X}$ (ce qui prouve le résultat, étant donné que la sémantique des expressions cible est déterministe).

Règle $(fail)$: puisque la valeur v est dans $\lceil \mathtt{I} \rfloor \wedge t_0$, elle n'est pas dans $\lceil p \rfloor$, et donc $v/p = \Omega$.

Règle $(first)$: d'après la prémisse, on obtient que v est dans $\lceil p_1 \rfloor$, et donc $v/p_1|p_2 = v/p_1$, ce qui permet de conclure en utilisant l'hypothèse d'induction.

Règle $(second)$: on obtient que v n'est pas dans $\lceil p_1 \rfloor$, et donc $v/p_1|p_2 = v/p_2$.

Règles $(factor)$: traitons par exemple le cas $p \xrightarrow{t_0 \wedge \lceil \mathbb{I} \rfloor} x$. On a alors $\overline{(v/p)_{|X} = \{x \mapsto v\} \oplus (v/p)_{|X \setminus \{x\}}}$.

Règles $(skip)$: Si $x \notin X$, on a $(v/x\&p)_{|X} = (v/(x := c)\&p)_{|X} = \overline{(v/p)_{|X}}$.

Règle $(prod)$: on a $(v/(q_1, q_2))_{|X} = (v_1/q_1)_{|X \cap \mathrm{Var}(q_1)} \oplus \overline{(v_2/q_2)_{|X \cap \mathrm{Var}(q_2)}}$, et on sait que v_i est dans $\pi_i[t_0; \lceil \mathtt{I} \rfloor]$.

Il suffit donc de voir que si $\mathtt{I} \vdash_i (q, t_0, X) \Rightarrow \mathtt{e}$ avec v_i dans t_0, alors $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} (v_i/\sigma(q))_{|X}$.

Règle $(succeed)$: On a $\mathtt{e} = \{\}$ et $X = \emptyset$. Or $v_i/\sigma(q)$ n'est pas Ω, car v_i est dans $t_0 \leq \lceil q \rfloor$. Donc $(v_i/\sigma(q))_{|X} = \{\}$.

Règle $(fetch)$: Supposons $\mathtt{e} = (i, r_{|X})$ avec $r = (q', t'_0, X') \in \mathrm{Dom}(\mathtt{R}_i)$, $\mathtt{I} = (\mathtt{R}_1, \mathtt{R}_2)$, et $\mathbb{I} = (\mathbb{R}_1, \mathbb{R}_2)$. On a alors $\mathtt{e} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_{|X}$ où $\mathbb{r} = (\mathbb{R}_i)_r$. Mais on a supposé que $v \Vdash \mathbb{I}$, ce qui implique $v_i \Vdash \mathbb{r} : r$. Or v est dans t_0, et donc *a fortiori* dans t'_0, et l'on obtient donc $\mathbb{r} = (v_i/\sigma(q))_{|X'}$. Or $X \subseteq X'$, et donc $\mathbb{r}_{|X} = (v_i/\sigma(q))_{|X}$. □

Considérons maintenant la construction de stratégies correctes pour une requête R_0 donnée. La Figure 8.3 définit un jugement $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$ où $I = \mathrm{Dom}(\mathtt{I})$.

$$\frac{(\forall r \in R_0)\ \mathtt{I} \vdash \sigma(r) \Rightarrow \mathtt{e}_r}{\mathtt{I} \vdash R_0 \Rightarrow (\mathtt{e}_r)_{r \in R_0}}\ (\mathit{assemble})$$

$$\frac{(\forall \mathtt{R} : R)\ \mathtt{I} \cup \mathtt{I}(i, \mathtt{R}) \vdash R_0 \Rightarrow \mathtt{s}_{\mathtt{R}}}{\mathtt{I} \vdash R_0 \Rightarrow (i, R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R}}\ (\mathit{subreq})$$

FIG. 8.3 – Production de stratégies

Il y a deux possibilités pour construire une stratégie. Soit l'on dispose de suffisamment d'information dans $\mathtt{I}$ pour calculer un résultat correct pour toutes les requêtes atomiques dans R_0, et dans ce cas on peut appliquer la règle (*assemble*), soit il faut se décider à appliquer une sous-requête R (à choisir) sur v_i, ce qui permet, suivant le résultat statique qui en ressort, d'obtenir plus d'information (règle (*subreq*)).

Theorème 8.21 *Si* $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$*, alors, pour tout paquet d'information* $\mathbb{I}$ *tel que* $\mathtt{I} = |\mathbb{I}|$*, on a :*

$$\mathbb{I} \Vdash \mathtt{s}^{R_0,I}$$

En particulier, si $\mathtt{I}^{\emptyset} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$*, alors :*

$$\Vdash \mathtt{s}^{R_0,I}$$

Preuve: Par induction sur la dérivation de $\mathtt{I} \vdash R_0 \Rightarrow \mathtt{s}^{R_0,I}$. Soit $v = (v_1, v_2)$, et $\mathbb{I}$ tel que $\mathtt{I} = |\mathbb{I}|$ et $v \Vdash \mathbb{I}$. Il s'agit de voir que si $\mathtt{s}^{R_0,I} \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{R}_0$, alors $v \Vdash R_0$.

Règle (*assemble*) : La sémantique donne $\mathbb{R}_0 = (\mathbb{r}_r)_{r \in R_0}$, avec $\mathtt{e}_r \overset{v,\mathbb{I}}{\rightsquigarrow} \mathbb{r}_r$ où $\mathtt{I} \vdash \sigma(r) \Rightarrow \mathtt{e}_r$ (pour chaque $r \in R_0$). Le Théorème 8.20 donne $v \Vdash \mathbb{r}_r : r$, et donc $v \Vdash \mathbb{R}_0$.

Règle (*subreq*) : La sémantique donne $\mathtt{s}_{|\mathbb{R}|} \overset{v,\mathbb{I} \cup (i:\mathbb{R})}{\rightsquigarrow} \mathbb{R}_0$ pour un certain résultat $v_i \Vdash \mathbb{R}$ avec $\mathrm{Dom}(\mathbb{R}) = R$. Le Lemme 8.11 donne $v \Vdash \mathbb{I} \cup (i : \mathbb{R})$. Prenons $\mathtt{R} = |\mathbb{R}|$. On constate que $|\mathbb{I} \cup (i : \mathbb{R})| = \mathtt{I} \cup (i : \mathtt{R})$, et la prémisse de (*subreq*) qui correspond à $\mathtt{R}$ donne bien, par induction : $v \Vdash \mathbb{R}_0$. □

Élaboration de stratégies Les systèmes définis dans les figures 8.2 et 8.3 ne donnent pas des constructions effectives pour les expressions cible ou les stratégies. Un schéma d'élaboration de stratégie est défini par :

- la manière de construire une expression cible pour un paquet d'information statique et un bon triplet (p, t_0, X) donnés, lorsque c'est possible ;
- la manière de choisir la sous-requête (i, R) a appliquer lorsque qu'il n'est pas possible de construire une expression cible pour toutes les requêtes atomiques de R avec le paquet d'information statique donné.

La manière de construire une expression cible lorsque c'est possible revient à spécifier une manière de construire une dérivation pour le jugement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. Une approche naturelle consiste à appliquer les règles $(fail)$, $(factor)$ et $(succeed)$ dès que possible.

Nous allons maintenant donner un outil qui permet de choisir les sous-requêtes à appliquer. Pour cela, nous allons rendre explicite ce qui empêche de construire une expression cible pour un bon triplet (p, t_0, X), et montrer comment le choix des sous-requêtes permet de supprimer ces obstructions. Considérons le jugement $\mathtt{I} \vdash (p, t_0, X)$ défini à la Figure 8.4. L'élément essentiel est la règle $(daemon)$, qui permet de suppléer la règle $(fetch)$ lorsque celle-ci ne peut pas s'appliquer.

$$\frac{t_0 \mathbin{\&} \lceil \mathtt{I} \rfloor \mathbin{\&} \lceil p \rfloor \simeq \mathbb{0}}{\mathtt{I} \vdash (p, t_0, X)}\ (fail)$$

$$\frac{\mathtt{I} \vdash (p_1, t_0, X) \quad \mathtt{I} \vdash (p_2, t_0 \mathbin{\backslash} \lceil p_1 \rfloor, X)}{\mathtt{I} \vdash (p_1 | p_2, t_0, X)}\ (alt)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \mathbin{\&} \lceil \mathtt{I} \rfloor} x \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\})}{\mathtt{I} \vdash (p, t_0, X)}\ (factor)$$

$$\frac{x \in X \quad p \xrightarrow{t_0 \mathbin{\&} \lceil \mathtt{I} \rfloor} (x := c) \quad \mathtt{I} \vdash (p, t_0, X \backslash \{x\})}{\mathtt{I} \vdash (p, t_0, X)}\ (factor)$$

$$\frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X)}{\mathtt{I} \vdash (x \& p, t_0, X)}\ (skip) \qquad \frac{x \notin X \quad \mathtt{I} \vdash (p, t_0, X)}{\mathtt{I} \vdash ((x := c) \& p, t_0, X)}\ (skip)$$

$$\frac{(\forall i = 1..2)\ \mathtt{I} \vdash_i (q_i, \pi_i[t_0; \lceil \mathtt{I} \rfloor], X \cap \mathrm{Var}(q_i))}{\mathtt{I} \vdash ((q_1, q_2), t_0, X)}\ (prod)$$

$$\frac{t_0 \leq \lceil q \rfloor}{\mathtt{I} \vdash_i (q, t_0, \emptyset)}\ (succeed)$$

$$\frac{r = (q, t_0', X') \in \mathrm{Dom}(\mathtt{R}_i) \quad t_0 \leq t_0' \quad X \subseteq X'}{(\mathtt{R}_1, \mathtt{R}_2) \vdash_i (q, t_0, X)}\ (fetch)$$

$$\frac{}{\mathtt{I} \vdash_i (q, t_0, X)}\ (daemon)$$

FIG. 8.4 – Squelette de production d'expressions cible

Lemme 8.22 *Si* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$ *et* $t_0' \mathbin{\&} \lceil \mathtt{I} \rfloor \leq t_0$, *alors* $\mathtt{I} \vdash (p, t_0', X) \Rightarrow \mathtt{e}$.

| *Preuve*: Par induction sur la dérivation de $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. □

Lemme 8.23 *Considérons une dérivation de* $\mathtt{I} \vdash (p, t_0, X)$ *qui n'utilise pas la règle* (*daemon*). *Alors :*

- *soit* $t_0 \mathbin{\&} \lceil \mathtt{I} \rfloor \leq \lceil p \rfloor$, *soit* $t_0 \mathbin{\&} \lceil \mathtt{I} \rfloor \mathbin{\&} \lceil p \rfloor \simeq \mathbb{0}$;
- *on peut calculer une expression cible* $\mathtt{e}$ *telle que* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$.

Preuve: La preuve se fait par induction sur la dérivation de $\mathtt{I} \vdash (p, t_0, X)$. Les règles $(fail)$, $(factor)$ et $(skip)$ sont immédiates.

Considérons la règle (alt). Par induction, on construit deux expressions $\mathtt{e}_i$ telles que $\mathtt{I} \vdash (p_1, t_0, X) \Rightarrow \mathtt{e}_1$ et $\mathtt{I} \vdash (p_2, t_0 \setminus \lbag p_1 \rbag, X) \Rightarrow \mathtt{e}_2$. Si $t_0 \wedge \lbag \mathtt{I} \rbag \leq \lbag p_1 \rbag$, alors $t_0 \wedge \lbag \mathtt{I} \rbag \leq \lbag p_1 | p_2 \rbag$, et donc $\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}_1$. Sinon, on a $t_0 \wedge \lbag \mathtt{I} \rbag \wedge \lbag p_1 \rbag \simeq \mathbb{0}$, et donc $t_0 \wedge \lbag \mathtt{I} \rbag \leq t_0 \setminus \lbag p_1 \rbag$. Le Lemme 8.22 donne ainsi $\mathtt{I} \vdash (p_2, t_0, X) \Rightarrow \mathtt{e}_2$, et la règle $(second)$ s'applique : $\mathtt{I} \vdash (p_1 | p_2, t_0, X) \Rightarrow \mathtt{e}_2$.

Considérons la règle $(prod)$, et posons $t_i = \pi_i[t_0; \mathtt{I}] \simeq \pi_i[t_0 \wedge \lbag \mathtt{I} \rbag]$. On constate d'abord que si $\mathtt{I} \vdash_i (q_i, t_i, X_i)$ avec l'une des règles $(succeed)$ ou $(fetch)$ (mais pas $(daemon)$), alors soit $t_i \wedge \lbag q_i \rbag \simeq \mathbb{0}$, soit $t_i \leq \lbag q_i \rbag$. Pour la règle $(succeed)$, c'est trivial. Pour la règle $(fetch)$, on prend $r = (q_i, t_i', X_i') \in \mathrm{Dom}(\mathtt{R}_i)$ tel que $t_i' \leq t_i'$, $X_i \subseteq X_i'$. Si $(\mathtt{R}_i)_r = \checkmark$, alors $\lbag \mathtt{R}_i \rbag \leq \lbag q_i \rbag \vee \neg t_i'$; or $t_i \leq \lbag \mathtt{R}_i \rbag$, et donc $t_i \leq \lbag q_i \rbag$. Si $(\mathtt{R}_i)_r = \Omega$, alors $\lbag \mathtt{R}_i \rbag \leq (\neg \lbag q_i \rbag) \vee \neg t_i'$, et donc $t_i \leq \neg \lbag q_i \rbag$, d'où finalement $t_i \wedge \lbag q_i \rbag \simeq \mathbb{0}$.

On a $t_0 \wedge \lbag \mathtt{I} \rbag \leq t_1 \times t_2$ et $\lbag (q_1, q_2) \rbag \simeq \lbag q_1 \rbag \times \lbag q_2 \rbag$. Donc si $t_1 \leq \lbag q_1 \rbag$ et $t_2 \leq \lbag q_2 \rbag$, alors $t_0 \wedge \lbag \mathtt{I} \rbag \leq \lbag (q_1, q_2) \rbag$. Sinon, par exemple si $t_1 \leq \neg \lbag q_1 \rbag$, alors $t_0 \wedge \lbag \mathtt{I} \rbag \wedge \lbag (q_1, q_2) \rbag \leq ((\neg \lbag q_1 \rbag) \times t_2) \wedge (\lbag q_1 \rbag \times \lbag q_2 \rbag) \simeq \mathbb{0}$. □

Remarque 8.24 *Ce résultat admet une réciproque : si* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$, *alors on peut construire une dérivation de* $\mathtt{I} \vdash (p, t_0, X)$ *sans utiliser la règle* $(daemon)$. *Essentiellement, il suffit d'effacer les* $\Rightarrow \mathtt{e}$ *dans la dérivation de* $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$. *Chaque instance de la règle* $(first)$ *ou* $(second)$ *se transforme en une instance de la règle* (alt), *et la prémisse manquante s'obtient avec la règle* $(fail)$.

Lemme 8.25 *Pour tout bon triplet* (p, t_0, X) *et tout paquet d'information statique* $\mathtt{I}$, *on peut construire une dérivation de* $\mathtt{I} \vdash (p, t_0, X)$.

En effet, il n'y a aucune « obstruction » grâce à la règle $(daemon)$. Comme pour le jugement $\mathtt{I} \vdash (p, t_0, X) \Rightarrow \mathtt{e}$, un choix naturel pour construire une dérivation de $\mathtt{I} \vdash (p, t_0, X)$ consiste à appliquer les règles $(fail)$,$(factor)$ et $(succeed)$ autant que possible.

Pour construire une stratégie pour R_0 en partant d'un paquet d'information statique $\mathtt{I}$, on commence par construire des dérivations pour les $\sigma(r)$ (pour $r \in R_0$), avec aussi peu de règles $(daemon)$ que possible (ce n'est pas une contrainte forte : ce n'est pas grave si l'on « rate » certaines règles qui évitent d'utiliser $(daemon)$, comme la factorisation, nous produirons juste une stratégie moins efficace). Si on arrive à le faire sans utiliser aucune règle $(daemon)$, c'est gagné, on peut construire une stratégie avec la règle $(assemble)$, c'est-à-dire sans faire de sous-requête. Il suffit d'appliquer le Lemme 8.23 pour construire une expression cible pour chaque requête atomique de R_0.

Sinon, il faut choisir $i = 1..2$ et une requête R à appliquer sur v_i. Regardons ce qu'il advient des instances de $(daemon)$ si l'on travaille non plus avec $\mathtt{I}$, mais avec $\mathtt{I}' = \mathtt{I} \cup (i : \mathtt{R})$ où $\mathtt{R}$ est un résultat statique pour R. On raffine l'information statique : $\lbag \mathtt{I}' \rbag \leq \lbag \mathtt{I} \rbag$. On voit que les dérivations des $\mathtt{I} \vdash \sigma(r)$ donnent trivialement des dérivations de $\mathtt{I}' \vdash \sigma(r)$. En fait, on peut remplacer une règle $(daemon)$ qui prouve $\mathtt{I}' \vdash_i (q, t_0, X)$ par une instance de $(fetch)$, pourvu

que la requête atomique (q, t_0, X) est dans R (ou si une requête (q', t'_0, X') est dans R avec les conditions de la règle $(fetch)$). En effet, si $\mathtt{I} = (\mathtt{R}'_1, \mathtt{R}'_2)$, on a : $\mathrm{Dom}(\mathtt{R}'_i) = \mathrm{Dom}(\mathtt{R}_i) \cup R$.

Mais l'information statique $\mathtt{I}'$ plus précise peut permettre de simplifier encore plus les dérivations des $\mathtt{I} \vdash \sigma(r)$, en autorisant à appliquer plus de règles $(factor)$, $(fail)$, $(succeed)$. En général, parmi les 2^n résultats statiques $\mathtt{R}$ (où n est le cardinal de R), nombreux sont ceux qui sont « impossible », c'est-à-dire tels que $\wr \mathtt{I}' \int \simeq \mathbb{0}$, ce qui permet d'appliquer immédiatement la règle $(fail)$ pour toutes les requêtes atomiques de R_0.

En tout cas, si l'on choisit i et R de sorte à garantir qu'il existe au moins une des règles $(daemon)$ qui peut se transformer en $(fetch)$, on arrive à diminuer le nombre de règles $(daemon)$ utilisées. En procédant par induction sur ce nombre, on arrive à construire une stratégie pour R_0.

La section suivante donne des exemples de constructions.

8.3.8 Exemples de constructions

Stratégie gauche-droite Dans une stratégie gauche-droite, on s'impose de n'effectuer qu'une seule requête sur chaque sous-arbre (gauche et droite), et de commencer pas le sous-arbre gauche (le choix symétrique est évidemment possible). Cela implémente l'idée des paragraphes « Déterminisation descendante » et « Propagation latérale » de la Section 8.2. Comme on ne visite qu'une fois chaque sous-arbre, il n'est pas utile d'utiliser la *memoization*.

Les stratégies gauche-droite ont la forme $(1, R_1) \mapsto (\mathtt{s}_\mathtt{R})_{\mathtt{R}:R_1}$ avec $\mathtt{s}_\mathtt{R} = (2, R_2(\mathtt{R})) \mapsto ((\mathtt{e}_{\mathtt{R},\mathtt{R}',r})_{r \in R_0})_{\mathtt{R}':R_2(\mathtt{R})}$, où R_1 est la requête à calculer sur v_1 et $R_2(\mathtt{R})$ est la requête à calculer sur v_2 (elle peut dépendre du résultat de R_1).

Pour construire $\mathtt{s}$, on part d'une dérivation $\mathtt{I}^{\emptyset} \vdash \sigma(r)$ pour chaque $r \in R_0$. On considère toutes les règles $(daemon)$ utilisées dans cette dérivation, avec $i = 1$. On prend pour R_1 l'ensemble des requêtes atomiques r correspondantes. Pour chaque $\mathtt{R} : R_1$, on simplifie la dérivation obtenue. Toutes les règles $(daemon)$ qui restent sont de la forme $\mathtt{I}' \vdash_2 r$. On prend pour $R_2(\mathtt{R})$ l'ensemble de ces requêtes atomiques r. Pour chaque $\mathtt{R}' : R_2(\mathtt{R})$, on obtient après simplification des dérivations qui n'utilisent pas la règle $(daemon)$, et on peut donc construire les expressions cible.

Stratégie itérative Une stratégie itérative consiste à appliquer uniquement des sous-requêtes constituées d'une unique requête atomique. Étant donné des dérivations de $\mathtt{I} \vdash \sigma(r)$ pour les $r \in R_0$, on choisit une instance de la règle $(daemon)$, soit $\mathtt{I} \vdash_i r$, et on construit la stratégie $(i, \{r\}) \mapsto (\mathtt{s}_\mathtt{R})$. Les stratégies $\mathtt{s}_\mathtt{R}$ sont construites de même, en partant de $\mathtt{I} \cup (i : \mathtt{R})$, après avoir simplifié la dérivation, et ainsi de suite. On diminue à chaque étape d'au moins 1 le nombre de règles $(daemon)$ utilisées, ce qui assure la terminaison du processus.

Un choix naturel pour trouver la règle $(daemon)$ consiste à la chercher d'abord dans la dérivation pour p_1, avant celle pour p_2, dans la règle (alt). En effet, si p_1 réussit, l'information statique sera suffisante pour ignorer p_2, alors que l'inverse n'est pas vrai (évidemment, si $\wr p_1 \int \wedge \wr p_2 \int \simeq \mathbb{0}$, cet argument ne tient pas).

Un exemple concret Nous allons donner un exemple concret de construction de stratégie. Cet exemple ne fait pas intervenir de variables de capture, ce qui

simplifie un peu les choses (en particulier, on peut toujours se passer de la règle $(fetch)$, et n'utiliser que $(fail)$ et $(succeed)$). Nous prenons $t_0 = (t_1 \times t_2) \vee (\mathbb{1} \times t_3)$ où $t_2 \wedge t_3 \simeq \mathbb{0}$, et q tel que $\sigma(q) = (q_1, q_2)$, avec $\mathrm{Var}(q_i) = \mathbb{0}$ et $\lbag q_i \rbag = t_i$ (on peut prendre $\sigma(q_i) = t_i$ si les t_i sont des types de base, et sinon, on peut appliquer la mise en forme normale de la Section 6.5.2). Nous nous intéressons à la requête $R_0 = \{(q, t_0, \emptyset)\}$.

Voici une dérivation obtenue avec les règles de la Figure 8.4 :

$$\dfrac{\dfrac{}{\mathtt{I}^\emptyset \vdash_1 (q_1, \mathbb{1}, \emptyset)}\,(daemon) \quad \dfrac{}{\mathtt{I}^\emptyset \vdash_2 (q_2, t_2 \vee t_3, \emptyset)}\,(daemon)}{\mathtt{I}^\emptyset \vdash ((q_1, q_2), t_0, \emptyset)}\,(prod)$$

Nous sommes obligés d'utiliser les règles $(daemon)$ parce qu'il n'y a aucune requête dans le paquet d'information statique initialement vide, ce qui empêche d'utiliser la règle $(fetch)$, et que la règle $(succeed)$ ne s'applique pas non plus. Pour continuer la construction de la stratégie, il faut maintenant choisir entre appliquer la requête $R_1 = \{(q_1, \mathbb{1}, \emptyset)\}$ au sous-arbre gauche, et appliquer la requête $R_2 = \{(q_2, t_2 \vee t_3, \emptyset)\}$ au sous-arbre droit.

Regardons les deux possibilités successivement. On choisit de commencer la stratégie par $(1, R_1) \mapsto (\ldots)$. Il faut compléter les $\ldots$. Il y a deux résultats statiques $\mathtt{R}_1$ possibles pour R_1, qui correspondent au succès ou à l'échec de son unique requête atomique r_1. En cas d'échec : $\mathtt{R}_1 = \{r_1 \mapsto \Omega\}$. On a alors : $\lbag \mathtt{I}(1, \mathtt{R}_1) \rbag = (\neg t_1 | \neg \mathbb{1}) \times \mathbb{1} \simeq (\neg t_1) \times \mathbb{1}$. Cela permet d'utiliser directement la règle $(fail)$:

$$\dfrac{t_0 \wedge \lbag\, \mathtt{I}(1, \mathtt{R}_1) \,\rbag \wedge \lbag\, (q_1, q_2) \rbag \simeq t_0 \wedge ((\neg t_1) \times \mathbb{1}) \wedge (t_1 \times t_2) \simeq \mathbb{0}}{\mathtt{I}(1, \mathtt{R}_1) \vdash ((q_1, q_2), t_0, \emptyset)}\,(fail)$$

et l'on peut instrumenter cette dérivation (avec les règles de la Figure 8.2) pour obtenir l'expression cible Ω.

En cas de succès de r_1, c'est-à-dire si $\mathtt{R}_1 = \{r_1 \mapsto \checkmark\}$, on obtient $\lbag \mathtt{I}(1, \mathtt{R}_1) \rbag = (t_1 | \neg \mathbb{1}) \times \mathbb{1} \simeq t_1 \times \mathbb{1}$. On peut alors éliminer une instance de la règle $(daemon)$, mais la deuxième subsiste nécessairement :

$$\dfrac{\dfrac{t_1 \leq \lbag q_1 \rbag}{\mathtt{I}(1, \mathtt{R}_1) \vdash_1 (q_1, t_1, \emptyset)}\,(succeed) \quad \dfrac{}{\mathtt{I}(1, \mathtt{R}_1) \vdash_2 (q_2, t_2 \vee t_3, \emptyset)}\,(daemon)}{\mathtt{I}(1, \mathtt{R}_1) \vdash ((q_1, q_2), t_0, \emptyset)}\,(prod)$$

On applique alors la requête R_2 sur le sous-arbre droit, c'est-à-dire que l'on continue la stratégie par $(2, R_2) \mapsto (\ldots)$. Il faut de nouveau distinguer suivant le succès ou l'échec de l'unique requête atomique r_2 de R_2. Si $\mathtt{R}_2 = \{r_2 \mapsto \Omega\}$, on obtient $\lbag \mathtt{R}_2 \rbag = (\neg t_2) \vee \neg(t_2 \vee t_3) \simeq \neg t_2$, et donc, en posant $\mathtt{I}_2 = \mathtt{I}(1, \mathtt{R}_1) \cup \mathtt{I}(2, \mathtt{R}_2)$: $\lbag \mathtt{I}_2 \rbag = t_1 \times (\neg t_2)$, ce qui permet d'utiliser $(fail)$ directement :

$$\dfrac{t_0 \wedge \lbag\, \mathtt{I}_2 \,\rbag \wedge \lbag\, (q_1, q_2) \rbag \simeq t_0 \wedge (t_1 \times (\neg t_2)) \wedge (t_1 \times t_2) \simeq \mathbb{0}}{\mathtt{I}_2 \vdash ((q_1, q_2), t_0, \emptyset)}\,(fail)$$

Si $\mathtt{R}_2 = \{r_2 \mapsto \checkmark\}$, on obtient $\lbag \mathtt{R}_2 \rbag = t_2 \vee \neg(t_2 \vee t_3) \simeq \neg t_3$, et donc, en posant $\mathtt{I}_2 = \mathtt{I}(1, \mathtt{R}_1) \cup \mathtt{I}(2, \mathtt{R}_2) : \lbag \mathtt{I}_2 \rbag = t_1 \times (\neg t_3)$, ce qui permet de remplacer $(daemon)$ par $(succeed)$:

$$\dfrac{\dfrac{t_1 \leq \lbag q_1 \rbag}{\mathtt{I}_2 \vdash_1 (q_1, t_1, \emptyset)}\,(succeed) \quad \dfrac{t_2 \leq \lbag q_2 \rbag}{\mathtt{I}_2 \vdash_2 (q_2, t_2, \emptyset)}\,(succeed)}{\mathtt{I}_2 \vdash ((q_1, q_2), t_0, \emptyset)}\,(prod)$$

ce qui correspond à l'expression cible $\{\} \oplus \{\}$, qui est équivalente à $\{\}$.

Globalement, en adoptant cette approche gauche-droite, on a obtenu la stratégie :

$$\mathtt{s}_1 = (1, R_1) \mapsto \begin{cases} \{r_1 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_1 \mapsto \checkmark\} & \mapsto & (2, R_2) \mapsto \begin{cases} \{r_2 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_2 \mapsto \checkmark\} & \mapsto & \{\} \end{cases} \end{cases}$$

Considérons maintenant l'autre choix possible, à savoir de commencer par la sous-requête R_2. En cas d'échec, on obtient $\wr \mathtt{I}(2, \mathtt{R}_2) \int \simeq \mathbb{1} \times \neg t_2$, d'où directement :

$$\frac{t_0 \wedge \wr\, \mathtt{I}(2, \mathtt{R}_2)\, \int \wedge \wr\, (q_1, q_2) \int \simeq \mathbb{0}}{\mathtt{I}(2, \mathtt{R}_2) \vdash ((q_1, q_2), t_0, \emptyset)}\ (fail)$$

En cas de succès, si $\mathtt{R}_2 = \{r_2 \mapsto \checkmark\}$, on obtient $\wr \mathtt{R}_2 \int \simeq \neg t_3$, et donc, $t_0 \wedge \wr\, \mathtt{I}(2, \mathtt{R}_2) \int = t_1 \times t_2$, ce qui permet de remplacer les *deux* règles (*daemon*) par (*succeed*) :

$$\frac{\dfrac{t_1 \leq \wr q_1 \int}{\mathtt{I}(2, \mathtt{R}_2) \vdash_1 (q_1, t_1, \emptyset)}\ (succeed) \quad \dfrac{t_2 \leq \wr q_2 \int}{\mathtt{I}(2, \mathtt{R}_2) \vdash_2 (q_2, t_2, \emptyset)}\ (succeed)}{\mathtt{I}(2, \mathtt{R}_2) \vdash ((q_1, q_2), t_0, \emptyset)}\ (prod)$$

ce qui correspond de nouveau à l'expression cible $\{\} \oplus \{\}$, qui se simplifie en $\{\}$.

La stratégie que l'on vient de construire est :

$$\mathtt{s}_2 = (2, R_2) \mapsto \begin{cases} \{r_2 \mapsto \Omega\} & \mapsto & \Omega \\ \{r_2 \mapsto \checkmark\} & \mapsto & \{\} \end{cases}$$

On peut alors se demander quelle stratégie il vaut mieux choisir, entre $\mathtt{s}_1$ et $\mathtt{s}_2$. Il n'y a pas *a priori* de meilleur choix. La stratégie $\mathtt{s}_2$ semble plus simple, puisqu'elle permet d'ignorer dans tous les cas le sous-arbre gauche, et qu'elle doit de toute manière être utilisée en cas d'échec de r_1 si l'on utilise $\mathtt{s}_1$ ($\mathtt{s}_2$ apparaît comme une sous-stratégie de $\mathtt{s}_1$). Mais cela n'est pas suffisant pour dire que $\mathtt{s}_2$ est nécessairement meilleur.

Si la sous-requête $r_1 = (q_1, \mathbb{1}, \emptyset)$ est plus beaucoup facile à évaluer que r_2 (par exemple si t_1 est un type de base, et que pour distinguer entre les types t_2 et t_3 il faut aller chercher en profondeur dans la valeur), et qu'il arrive effectivement, à l'exécution, que r_1 échoue, alors il est préférable d'utiliser $\mathtt{s}_1$ qui va parfois éviter d'évaluer r_2 (ce qui compense le fait que lorsque r_1 réussit, on a fait un calcul inutile). Un autre cas dans lequel il vaut mieux utiliser $\mathtt{s}_1$ est lorsque r_1 est seulement un petit peu plus facile à évaluer que r_2, et qu'à l'exécution, le plus souvent, r_1 échoue. En effet, choisir $\mathtt{s}_1$ dans cette situation permet d'éviter le plus souvent r_2, et même si le gain est faible, il est souvent réalisé. Donnons un modèle probabiliste simple pour appuyer l'intuition. On suppose que le coût de r_1 est constant, soit C_1, et que la probabilité de réussit de r_1 est $0 \leq \alpha_1 \leq 1$. De même, on introduit C_2, le coût de r_2. En ne considérant que le coût des sous-requêtes, le coût moyen de $\mathtt{s}_1$ est alors $\alpha_1(C_1 + C_2) + (1 - \alpha_1)C_1 = C_1 + \alpha_1 C_2$ (si r_1 réussit, il faut aussi évaluer r_2). Le coût moyen de $\mathtt{s}_2$ est simplement C_2. Choisir $\mathtt{s}_1$ est donc préférable dès que :

$$C_1 \leq (1 - \alpha_1)C_2$$

ce qui permet de retrouver les cas de la discussion informelle ci-dessus. Évidemment, la valeur de α_1 est inconnue du compilateur (elle dépend des valeurs effectivement manipulées à l'exécution), et le fait d'avoir supposé le coût des sous-requêtes comme étant constant n'est pas fondé (il dépend également des valeurs effectivement rencontrées). Cette analyse suggère néanmoins qu'il n'est pas possible de se reposer sur une hypothétique notion d'optimalité pour choisir entre les diverses stratégies possibles : il s'agit bien d'un choix heuristique.

Remarque 8.26 *Levin et Pierce [LP04] suggèrent une heuristique de « facteur de branchement maximal », qui, reformulée dans notre formalisme, suggérerait de choisir* $\mathtt{s}_2$ *dans cet exemple (inspiré de l'exemple présenté à la Figure 3 de leur article).*

8.4 Factorisation des captures

Dans cette section, nous montrons comment calculer les prédicats $p \xrightarrow{t_0} x$ et $p \xrightarrow{t_0} (x := c)$ introduits à la Section 8.3.

Le prédicat $p \xrightarrow{t_0} (x := c)$ est le plus facile, car on peut simplement utiliser l'algorithme de typage du filtrage. En effet, en utilisant le Théorème 6.12, on constate :

$$p \xrightarrow{t_0} (x := c) \iff ((t \wedge \lfloor p \rfloor) / p)(x) \leq b_c$$

et l'on a vu comment calculer l'environnement $((t \wedge \lfloor p \rfloor) / p)$.

Passons au prédicat $p \xrightarrow{t_0} x$. Nous allons en fait axiomatiser de manière inductive sa négation. Considérons le jugement $\vdash (p, t_0, x)$ défini par le système de règles ci-dessous :

$$\frac{x \notin \mathrm{Var}(p)}{\vdash (p, t_0, x)}$$

$$\frac{\vdash (p_1, t_0 \wedge \lfloor p_1 \rfloor, x)}{\vdash (p_1 | p_2, t_0, x)} \quad \frac{\vdash (p_2, t_0 \setminus \lfloor p_1 \rfloor, x)}{\vdash (p_1 | p_2, t_0, x)}$$

$$\frac{\exists i.\ \vdash (\sigma(q_i), \pi_i[t_0], x)}{\vdash ((q_1, q_2), t_0, x)}\ (prod)$$

$$\frac{\forall i.\ \vdash (p_i, t_0, x)}{\vdash (p_1 \& p_2, t_0, x)}$$

$$\frac{t_0 \not\leq b_c}{\vdash ((x := c), t_0, x)}$$

Lemme 8.27 *Soit p un motif bien formé, $v \in \lfloor p \rfloor$, et $x \in Var(p)$. Alors la valeur $(v/p)(x)$, vue comme un arbre binaire, a au plus autant de nœuds que v.*

> *Preuve*: Par induction sur le couple (v, p), en suivant la définition de la sémantique du filtrage. □

Lemme 8.28 *Soit q_1, q_2 deux nœuds de motif, v une valeur, et $x \in Var(q_1) \cup Var(q_2)$. Alors :*

$$(v/(q_1, q_2))(x) = v \iff v = (v_1, v_2) \wedge \forall i.\ (x \in Var(q_i) \wedge v_i = (v_i/\sigma(q_i))(x))$$

Preuve: L'implication $\Leftarrow$ est triviale. Prouvons l'implication $\Rightarrow$. Tout d'abord, si v n'est pas un couple, alors $(v/(q_1, q_2))(x) = \Omega$. Supposons donc $v = (v_1, v_2)$. Si $x \in \mathrm{Var}(q_1) \cap \mathrm{Var}(q_2)$, on a $(v_1, v_2) = v = (v/(q_1, q_2))(x) = ((v_1/\sigma(q_1))(x), (v_1/\sigma(q_2))(x))$, et donc $v_i = (v_i/\sigma(q_i))(x)$. Montrons qu'il est impossible d'avoir $x \in \mathrm{Var}(q_1) \backslash \mathrm{Var}(q_2)$, le cas symétrique se traitant de même. Si $x \in \mathrm{Var}(q_1) \backslash \mathrm{Var}(q_2)$, alors $(v_1, v_2) = (v_1/\sigma(q_2))(x)$, ce qui contredit le Lemme 8.27. □

Theorème 8.29 *On a :* $p \xrightarrow{t_0} x \iff \neg(\vdash (p, t_0 \mathbin{\mathbf{\wedge}} \lfloor\!\lceil p \rceil\!\rfloor, x))$

Preuve: Quitte à remplacer t_0 par $t_0 \mathbin{\mathbf{\wedge}} \lfloor\!\lceil p \rceil\!\rfloor$, on peut évidemment supposer que $t_0 \leq \lfloor\!\lceil p \rceil\!\rfloor$.

On montre d'abord, par induction sur la dérivation, l'implication $(\vdash (p, t_0, x)) \Rightarrow \neg(p \xrightarrow{t_0} x)$, en supposant que $t_0 \leq \lfloor\!\lceil p \rceil\!\rfloor$. Plus précisément, on associe à une dérivation de $\vdash (p, t_0 \mathbin{\mathbf{\wedge}} \lfloor\!\lceil p \rceil\!\rfloor, x)$ une valeur $v \in [\![t_0]\!]$ telle que $v/p \neq v$. Tous les cas sont faciles. La règle (*prod*) se traite avec le Lemme 8.28.

Pour prouver l'implication $\neg(p \xrightarrow{t_0} x) \Rightarrow (\vdash (p, t_0, x))$, nous montrons en fait l'assertion suivante, par induction sur le couple (v, p) :

$$(v \in [\![t_0]\!] \wedge x \in \mathrm{Var}(p) \wedge (v/p)(x) \neq x) \Rightarrow (\vdash (p, t_0, x))$$

L'induction suit la sémantique du filtrage. Tous les cas sont faciles. La règle (*prod*) se traite avec le Lemme 8.28. □

Pour savoir si la variable x se factorise dans le motif p pour un type d'entrée t_0, il s'agit donc de calculer le prédicat inductif défini par les règles données plus haut. Cela rentre directement dans le formalisme de la Section 7.1, et l'on peut donc utiliser par exemple l'algorithme sans retour-arrière pour décider si le jugement $\vdash (p, t_0 \mathbin{\mathbf{\wedge}} \lfloor\!\lceil p \rceil\!\rfloor, x)$ est dérivable ou non.

Troisième partie

Le langage CDuce

Chapitre 9

Enregistrements

Dans ce chapitre, nous allons étendre le calcul du Chapitre 5 avec une nouvelle classe de types, valeurs et motifs : les enregistrements. Nous indiquons comment adapter les définitions et les résultats des chapitres précédents.

Soit $\mathcal{L}$ un ensemble infini d'étiquettes. Les valeurs enregistrement sont des fonctions d'un ensemble fini d'étiquettes dans les valeurs. Pour simplifier l'exposé, nous allons en fait supposer que les enregistrements sont définis sur tout l'ensemble $\mathcal{L}$ mais qu'ils sont constants sur presque toutes les étiquettes (c'est-à-dire sur un ensemble cofini). Nous verrons à la Section 9.5 comment coder des fonctions partielles.

Fonctions presque constantes Pour un ensemble Z quelconque, une fonction $r : \mathcal{L} \to Z$ est dite presque constante s'il existe un élément $z \in Z$ tel que l'ensemble $\{l \in \mathcal{L} \mid f(l) \neq z\}$ est fini. L'élément z est uniquement identifié. On le note $\texttt{def}(r)$. On pose : $\mathrm{Dom}(r) := \{l \mid f(l) \neq \texttt{def}(r)\}$. L'ensemble des fonctions presque constantes de $\mathcal{L}$ dans Z est noté $\mathcal{L} \xrightarrow{c} Z$.

La notation $\{l_1 = z_1; \ldots; l_n = z_n; \_ = z\}$ désigne la fonction $r \in \mathcal{L} \xrightarrow{c} Z$ définie par $r(l_i) = z_i$ pour $i = 1..n$ et $r(l) = z$ pour $l \notin \{l_1, \ldots, l_n\}$. Tout élément $r \in \mathcal{L} \xrightarrow{c} Z$ s'écrit sous cette forme, et l'on peut même choisir l'ensemble $\{l_1, \ldots, l_n\}$ arbitrairement grand, mais cette écriture n'est pas unique (elle le devient si on impose d'avoir $z_i \neq z$).

Si $(Z_l)_{l \in \mathcal{L}}$ est une famille de sous-ensembles de Z, on note $\prod^{c}_{l \in \mathcal{L}} Z_l$ le sous-ensemble de $\mathcal{L} \xrightarrow{c} Z$ constitué des fonctions r telles que $r(l) \in Z_l$ pour toute étiquette l.

9.1 Types

9.1.1 Algèbre de types

Atomes Nous introduisons d'abord les types enregistrements dans l'algèbre minimale définie à la Section 3.3. Le foncteur F donne la signature des constructeurs. Nous avions posé :

$$a \in FX ::= x \mathbin{\rightarrow} x \;\mid\; x \mathbin{\times} x \;\mid\; b$$

Nous prenons maintenant :

$$a \in FX ::= x \rightarrow x \;\mid\; x \times x \;\mid\; b \;\mid\; r$$

où r désigne un élément de $\mathcal{L} \xrightarrow{c} X$. Nous disposons d'un nouvel univers d'atomes **rec**, en plus de **fun**, **prod**, **basic**. L'ensemble des atomes dans cet univers est :

$$T_{\mathbf{rec}} = \mathcal{L} \xrightarrow{c} T = \{\{l_1 = \theta_1; \ldots; l_n = \theta_n; \_ = \theta_0\} \mid \theta_i \in T\}$$

Socle Nous étendons la définition d'un socle $\sqsupset$ en imposant que si $r \in T_{\mathbf{rec}}$ est un atome enregistrement qui apparaît dans $P \cup N$ (avec (P, N) dans un type de $\sqsupset$), alors $\tau(r(l)) \in \sqsupset$ pour tout $l \in \mathcal{L}$. Autrement dit :

$$\{l_1 = \theta_1; \ldots; l_n = \theta_n; \_ = \theta_0\} \in P \cup N \Rightarrow \forall i = 0..n.\ \tau(\theta_i) \in \sqsupset$$

9.1.2 Modèles

On reprend la définition de l'interprétation extensionnelle associée à une interprétation ensembliste $[\![\_]\!] : \widehat{T} \to \mathcal{P}(D)$ (Définition 4.2). On pose :

$$\mathbb{E}D := \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega) + \mathcal{L} \xrightarrow{c} D$$

et, pour $r \in \mathcal{L} \xrightarrow{c} T$:

$$\mathbb{E}[\![r]\!] = \prod_{l \in \mathcal{L}}^{c} [\![\tau(r(l))]\!] \subseteq \mathcal{L} \xrightarrow{c} D$$

On note $\mathbb{E}_{\mathbf{rec}} D = \mathcal{L} \xrightarrow{c} D$.

Dans la définition d'un modèle bien fondé (Définition 4.3), on ajoute la condition :

$$d' \in \mathcal{L} \xrightarrow{c} D \Rightarrow \forall l \in \mathcal{L}.\ d'(l) \lhd d$$

Dans la définition d'un modèle structurel (Définition 4.4), on demande que D soit solution initiale d'une équation de la forme :

$$D = \mathcal{C} + D \times D + \mathcal{P}(D \times D_\Omega) + \mathcal{L} \xrightarrow{c} D$$

et que, pour $r \in \mathcal{L} \xrightarrow{c} T$:

$$[\![r]\!] = \mathbb{E}[\![r]\!]$$

9.1.3 Sous-typage

Nous reprenons la démarche de la Section 4.3. L'analogue des Lemmes 4.6 et 4.9 est donné par :

Lemme 9.1 *Soient* $(X_i)_{i\in P}$ *et* $(X_i)_{i\in N}$ *deux familles d'éléments de* $\mathcal{L} \xrightarrow{c} \mathcal{P}(D)$. *Soit* $L \supseteq \bigcup_{i\in P\cup N} Dom(X_i)$. *Alors :*

$$\bigcap_{i\in P} \prod_{l\in\mathcal{L}}^{c} X_i(l) \subseteq \bigcup_{i\in N} \prod_{l\in\mathcal{L}}^{c} X_i(l)$$

$$\Longleftrightarrow$$

$$\forall \iota : N \to L \cup \{\_\}. \begin{cases} \exists l \in L.\ \bigcap_{i\in P} X_i(l) \subseteq \bigcup_{i\in N \mid \iota(i)=l} X_i(l) & (i) \\ \vee \\ \exists i_0 \in N.\ (\iota(i_0) = \_) \wedge \bigcap_{i\in P} \mathtt{def}(X_i) \subseteq \mathtt{def}(X_{i_0}) & (ii) \\ \vee \\ \bigcap_{i\in P} \mathtt{def}(X_i) = \emptyset & (iii) \end{cases}$$

(avec la convention $\bigcap_{i\in\emptyset} \prod_{l\in\mathcal{L}}^{c} X_i(l) = \mathcal{L} \xrightarrow{c} D$*)*

Preuve: Montrons d'abord la contraposée de l'implication $\Rightarrow$. Supposons donc l'existence d'une fonction ι qui ne vérifie aucune des trois conditions (i), (ii), (iii) du membre droit. Nous allons construire un élément ρ qui infirme l'inclusion ensembliste du membre gauche. Sur L, les valeurs de ρ sont données par (i). On choisit $\rho(l)$ dans l'ensemble :

$$\bigcap_{i\in P} X_i(l) \setminus \bigcup_{i\in N \mid \iota(i)=l} X_i(l)$$

ce qui garantit déjà que ρ n'est pas dans $\prod_{l\in\mathcal{L}}^{c} X_i(l)$ si $\iota(i) \in L$.

Pour chaque i_0 tel que $\iota(i_0) = \_$, on choisit un $l_{i_0} \notin L$, tous distincts, et l'on prend pour $\rho(l_{i_0})$ un élément de

$$\bigcap_{i\in P} \mathtt{def}(X_i) \setminus \mathtt{def}(X_{i_0})$$

qui n'est pas vide, d'après (ii), et cela garantit que ρ n'est pas dans $\prod_{l\in\mathcal{L}}^{c} X_{i_0}(l)$ si $\iota(i_0) = \_$.

Il ne reste plus qu'à compléter ρ sur les étiquettes $l \in \mathcal{L}\setminus L$ qui ne sont pas de la forme l_{i_0} pour $\iota(i_0) = \_$. On choisit un élément d de

$$\bigcap_{i\in P} \mathtt{def}(X_i)$$

qui n'est pas vide d'après (iii) et l'on pose $\rho(l) = d$ pour toute ces étiquettes. On a ainsi construit une fonction ρ presque constante, et qui infirme l'inclusion du membre gauche.

Passons maintenant à la preuve de l'implication $\Leftarrow$, encore par contraposition. On prend une fonction ρ qui infirme l'inclusion du membre gauche. Pour tout $i_0 \in N$, on peut donc trouver une étiquette l_{i_0} telle que :

$$\rho(l_{i_0}) \in \bigcap_{i \in P} X_i(l_{i_0}) \backslash X_{i_0}(l_{i_0})$$

Nous définissons une fonction $\iota : N \to L \cup \{\_\}$ par $\iota(i_0) = l_{i_0}$ si $l_{i_0} \in L$ et $\iota(i_0) = \_$ sinon. Vérifions que cette fonction ι infirme les assertions (i),(ii) et (iii). Si (i) était vraie, nous pourrions trouver $l \in L$ et $i_0 \in N$ avec $\iota(i_0) = l$ et $\rho(l) \in X_{i_0}(l)$ (car $\rho(l) \in \bigcap_{i \in P} X_i(l)$). Mais $\iota(i_0) = l$ donne $l = l_0$, et $\rho(l_{i_0}) \notin X_{i_0}(l_{i_0})$. Passons à l'assertion (ii). Soit $i_0 \in N$ avec $\iota(i_0) = \_$, c'est-à-dire $l_{i_0} \notin L$. On a donc $\rho(l_{i_0}) = \mathtt{def}(\rho)$, et $X_i(l_{i_0}) = \mathtt{def}(X_i)$ pour $i \in P \cup N$, ce qui donne : $\mathtt{def}(\rho) \in \bigcap_{i \in P} \mathtt{def}(X_i) \backslash \mathtt{def}(X_{i_0})$, et suffit donc à infirmer (ii). Quant à (iii), il suffit de considérer $\mathtt{def}(\rho)$, qui est bien dans $\bigcap_{i \in P} \mathtt{def}(X_i)$. □

Remarque 9.2 *Le résultat provient de la constatation suivante. On peut identifier l'ensemble $\mathcal{L} \xrightarrow{c} Z$ avec :*

$$\prod_{l \in L} Z \times \left((\mathcal{L} \backslash L) \xrightarrow{c} Z \right)$$

On se ramène ainsi à un problème d'inclusion entre une intersection et une réunion de produits cartésiens finis, problème que l'on a déjà rencontré dans le deux de produits à deux éléments (Lemme 4.6) et qui se généralise facilement. Pour la dernière composante, on doit traiter le problème de l'inclusion entre une intersection et une réunion de produits infinis dont toutes les composantes sont identiques (en se restreignant aux familles presque constantes). Ce problème est assez proche de celui traité dans le Lemme 4.8, ce qui explique le point (ii), sauf qu'un produit infini d'ensembles vides est vide, alors que $\mathcal{P}(\emptyset)$ n'est pas vide (ce qui explique le point (iii)).

De ce résultat, on dérive la manière d'étendre la définition de l'ensemble $\mathbb{E}\mathcal{S}$ (Définition 4.11). On prend :

$$C_{\mathbf{rec}} ::= \forall \iota : N_{\mathbf{rec}} \to L \cup \{\_\}. \left\{ \begin{array}{ll} & \exists l \in L. \left(\bigwedge_{r \in P} \tau(r(l)) \backslash \bigvee_{r \in N_{\mathbf{rec}} \mid \iota(r) = l} \tau(r(l)) \right) \in \mathcal{S} \\ \vee & \exists r' \in N_{\mathbf{rec}}. (\iota(r') = \_) \wedge \left(\bigwedge_{r \in P} \tau(\mathtt{def}(r)) \backslash \tau(\mathtt{def}(r')) \right) \in \mathcal{S} \\ \vee & \left(\bigwedge_{r \in P} \tau(\mathtt{def}(r)) \right) \in \mathcal{S} \end{array} \right.$$

où $N_{\mathbf{rec}} = N \cap T_{\mathbf{rec}}$ et $L = \bigcup_{r \in P \cup N_{\mathbf{rec}}} \mathrm{Dom}(r)$ (ou un sur-ensemble).

La construction d'un modèle universel (Section 4.5) ne pose pas de problème. Un algorithme de calcul du sous-typage pour les modèles universels s'obtient par

une extension de l'algorithme du Chapitre 7. On peut également appliquer la technique de la Section 7.3 pour les enregistrements. Plutôt que de considérer des arbres binaires, on utilise un arbre dont le branchement est indexé par $L \cup \{\_\}$.

9.1.4 Décomposition, projection

Nous notons $\mathbb{1}_{\mathbf{rec}}$ le type de tous les enregistrements, c'est-à-dire $\{\_ = \mathbb{1}\}$.

On définit l'ensemble des étiquettes d'un type t par :

$$\mathcal{L}(t) = \bigcup_{(P,N)\in t} \bigcup_{r\in(P\cup N)\cap T_{\mathbf{rec}}} \mathrm{Dom}(r)$$

Soit $L = \{l_1; \ldots; l_n\}$ un ensemble fini d'étiquettes. Si $(t_l)_{l\in L}$ est une famille de types, E un ensemble fini de types, et t un type, nous posons :

$$\begin{aligned} R((t_l)_{l\in L}; t_0; E) := \quad & \{l_1 = t_{l_1}; \ldots; l_n = t_{l_n}; \_ = t_0\} \setminus \\ & \bigvee_{s\in E} \{l_1 = \mathbb{1}; \ldots; l_n = \mathbb{1}; \_ = \neg s\} \end{aligned}$$

Chaque type $s \in E$ ajoute une contrainte qui s'interprète en « il existe une étiquette hors de L dont la valeur est dans s ».

Nous pouvons énoncer l'équivalent des Lemmes 4.34 et 4.35 sur cette représentation.

Lemme 9.3

$$R((t_l)_{l\in L}; t_0; E) \wedge R((t'_l)_{l\in L}; t'_0; E') \simeq R((t_l \wedge t'_l)_{l\in L}; t_0 \wedge t'_0; E \cup E')$$

Lemme 9.4

$$R((t_l)_{l\in L}; t_0; E) \setminus R((t'_l)_{l\in L}; t'_0; \emptyset) \simeq R((t_l)_{l\in L}; t; E \cup \{\neg t'_0\}) \vee \bigvee_{l_0\in L} R((t''_{l,l_0})_{l\in L}; t_0; E)$$

où $t''_{l,l_0} = t_l \setminus t'_l$ *si* $l = l_0$ *et* $t''_{l,l_0} = t_l$ *sinon.*

En utilisant ces deux faits et une preuve similaire à celle du Théorème 4.36, on peut établir :

Theorème 9.5 *Soit t un type tel que $t \leq \mathbb{1}_{\mathbf{rec}}$ et $\mathcal{L}(t) \subseteq L$. Alors il existe un ensemble fini $\pi^L_{\mathbf{rec}}(t)$ de triplets $((t_l)_{l\in L}, t_0, E) \in \widehat{T}^L \times \widehat{T} \times \mathcal{P}_f(\widehat{T})$ tel que :*

- $t \simeq \bigvee_{((t_l),t_0,E)\in\pi^L_{\mathbf{rec}}(t)} R((t_l); t_0; E)$
- $\forall((t_l), t_0, E) \in \pi^L_{\mathbf{rec}}(t).\ R((t_l); t_0; E) \not\simeq \mathbb{0}$
- *si $\mathbb{\sqsupset}$ est un socle qui contient t, alors : $\pi^L_{\mathbf{rec}}(t) \subseteq \mathbb{\sqsupset}^L \times \mathbb{\sqsupset} \times \mathcal{P}_f(\mathbb{\sqsupset})$*

On voit $\pi^L_{\mathbf{rec}}$ comme une fonction partielle, qui est définie sur les types t tels que $t \leq \mathbb{1}_{\mathbf{rec}}$ et $\mathcal{L}(t) \subseteq L$.

Notons que la condition $R((t_l); t_0; E) \not\simeq \mathbb{0}$ est équivalente à :

$$\begin{cases} \forall l \in L.\ t_l \not\simeq \mathbb{0} \\ t_0 \not\simeq \mathbb{0} \\ \forall s \in E.\ t_0 \wedge s \not\simeq \mathbb{0} \end{cases}$$

Projection sur une étiquette Pour une étiquette l_0 et un sous-type t de $\mathbb{1}_{\mathbf{rec}}$, nous voulons définir un type $\pi_{l_0}[t]$ tel que, dans un modèle structurel :

$$[\![\pi_{l_0}[t]]\!] = \{\rho(l_0) \mid \rho \in [\![t]\!]\}$$

On peut le voir comme la plus petite solution s à l'inéquation :

$$t \leq \{l_0 = s; \_ = \mathbb{1}\}$$

Le Théorème 9.5 permet de répondre à la question, en se ramenant à des types de la forme $R((t_l)_{l\in L}; t_0; E)$ où l'on a pris $L \supseteq \mathcal{L}(t) \cup \{l_0\}$. L'ensemble des valeurs possibles pour l'étiquette $\rho(l_0)$ lorsque ρ décrit $[\![R((t_l)_{l\in L}; t_0; E)]\!]$ est $[\![t_{l_0}]\!]$. On peut donc poser :

$$\pi_{l_0}[t] = \bigvee_{((t_l)_{l\in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} t_{l_0}$$

En fait, on pourrait se contenter de $L \supseteq \mathcal{L}(t)$, en prenant, si $l_0 \notin L$:

$$\pi_{l_0}[t] = \bigvee_{((t_l)_{l\in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} t_0$$

En effet, les affirmations « il existe au moins une étiquette hors de L dont la valeur est dans s », pour $s \in E$, ne donnent aucune information (plus précise que t_0) sur l'ensemble des valeurs possibles pour l'étiquette l_0.

9.2 Motifs

Nous ajoutons les enregistrements à l'algèbre de motifs (Chapitre 6). Nous ajoutons un cas dans la définition de l'image GY d'un ensemble Y par le foncteur G :

$$\begin{array}{rcll} p \in GY & ::= & \ldots & \\ & \mid & r & r \in \mathcal{L} \xrightarrow{c} Y \end{array}$$

Dans la définition d'un ensemble stable de motifs (Définition 6.1), nous ajoutons la condition $(\sigma \circ r) \in \mathcal{L} \xrightarrow{c} S$, c'est-à-dire :

$$\forall r \in S.\ \forall l \in \mathcal{L}.\ \sigma(r(l)) \in S$$

Une version plus effective de cette condition est, pour un motif $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$:

$$\forall i = 0..n.\ \sigma(q_i) \in S$$

La condition de bonne formation associé à un motif enregistrement r est :

$$\forall l_1 \neq l_2.\ \mathrm{Var}(r(l_1)) \cap \mathrm{Var}(r(l_2)) = \emptyset$$

Une version plus effective de cette condition est, pour un motif $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$:

$$(\forall i \neq j.\ \mathrm{Var}(q_i) \cap \mathrm{Var}(q_j) = \emptyset) \wedge (\mathrm{Var}(q_0) = \emptyset)$$

Ainsi, le nœud de motif q répété infiniment souvent dans r ne peut se contenter que d'effectuer un test de type.

Sémantique La sémantique du filtrage (dans un modèle structurel de support D) est donnée par :

$$d/r = \begin{cases} \bigoplus_{l \in \mathcal{L}} d(l)/\sigma(r(l)) & \text{si } d \in \mathcal{L} \xrightarrow{c} D \\ \Omega & \text{sinon} \end{cases}$$

Une version plus effective est, pour un motif $p = \{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}$ et un élément $d = \{l_1 = d_1; \ldots; l_n = d_n; \_ = d_0\}$:

$$d/p = d_0/\sigma(q_0) \oplus \ldots \oplus d_n/\sigma(q_n)$$

Notons que $d_0/\sigma(q_0)$ ne peut être que Ω ou $\{\}$, car $\mathrm{Var}(\sigma(q_0)) = \emptyset$ (motif bien formé).

Typage On étend sans problème l'opérateur $\wr\_\int$. En particulier, pour un motif enregistrement r, nous avons :

$$\wr r \int = \wr\_\int \circ \sigma \circ r$$

où le membre droit est un atome enregistrement. De manière plus explicite :

$$\wr\{l_1 = q_1; \ldots; l_n = q_n; \_ = q_0\}\int = \{l_1 = \wr\sigma(q_1)\int; \ldots; l_n = \wr(\sigma(q_n)\int; \_ = \wr\sigma(q_0)\int\}$$

Pour le typage du filtrage, on peut étendre le Lemme 6.11 avec :

$$(t /\!\!/ r)(x) = (\pi_l[t] /\!\!/ \sigma(r(l)))(x)$$

où l'étiquette l est définie par : $x \in \mathrm{Var}(\sigma(r(l)))$.

Le Théorème 6.12 s'étend alors sans problème : il faut simplement introduire une extension de la notion d'un système $\mathsf{V}\times$ qui tient compte des enregistrements. Un tel système est une fonction $\phi : V \to \mathcal{P}_f(V + V \times V + (\mathcal{L} \xrightarrow{c} V) + \widehat{T})$.

L'élimination de l'intersection (Section 6.5.2) dans les motifs se fait en constatant que, tout comme le constructeur produit $\times$, le constructeur enregistrement commute avec l'intersection (l'intersection se distribue sur les composantes).

Compilation La technique d'implémentation du filtrage à base de stratégies pour le cas des valeurs couples (Chapitre 8) s'adapte au cas des valeurs enregistrements (c'est-à-dire d'éléments de $\mathcal{L} \xrightarrow{c} \mathcal{V}$). Une stratégie pour le cas des enregistrements peut effectuer deux genres de sous-requêtes. Comme dans le cas des produits, elle peut appliquer une requête R sur une sous-valeur $v(l)$; on note $(l, R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R}$ une telle stratégie. Elle peut aussi appliquer une requête R sur toutes les valeurs $v(l)$ pour l dans un ensemble cofini $\mathcal{L}\backslash L$ (L fini). En fait il n'y a qu'un nombre fini de valeurs $v(l)$ différentes, ce qui rend ce calcul possible. On note $(\mathcal{L}\backslash L, R) \mapsto (\mathtt{s}_{\mathtt{R}})_{\mathtt{R}:R}$ cette stratégie. Dans ce cas, toutes les requêtes atomiques (q, t_0, X) dans R sont telles que $\mathrm{Var}(q) = \emptyset$ (et donc $X = \emptyset$), et le résultat, pour chaque requête atomique est la conjonction logique de tout les résultats pour les sous-valeurs $v(l)$ ($l \in \mathcal{L}\backslash L$), en identifiant Ω avec faux et $\{\}$ avec vrai.

9.3 Calcul

Calcul Le calcul du Chapitre 5 s'étend naturellement avec un constructeur d'enregistrement dans les expressions :

$$\begin{array}{rcll} e \in \mathcal{E} & ::= & \ldots & \\ & | & \rho & \rho \in \mathcal{L} \xrightarrow{c} \mathcal{E} \end{array}$$

Système de types La règle de typage associée est :

$$\frac{r \in \mathcal{L} \xrightarrow{c} \widehat{T} \quad (\forall l)\ \Gamma \vdash \rho(l) : r(l)}{\Gamma \vdash \rho : r}$$

Une expression enregistrement dont toutes les composantes sont des valeurs est elle-même une valeur. Toutes les propriétés meta-théoriques du système de types restent valides.

Le lemme d'inversion (Lemme 5.12) s'étend avec :

$$[\![r]\!]_{\mathcal{V}} = \{\rho \in \mathcal{L} \xrightarrow{c} \mathcal{V} \mid \forall l.\ \vdash \rho(l) : r(l)\}$$

Sémantique Pour définir la sémantique, on rajoute dans la définition des contextes d'évaluation des contextes pour réduire une étiquette donnée $\{l_1 = []; l_2 = e_2; \ldots; l_n = e_n; \_ = e\}$ ou un ensemble cofini d'étiquettes avec la même expression $\{l_1 = e_1; \ldots; l_n = e_n; \_ = []\}$.

Schémas, inférence de types Pour mettre en place l'algorithme d'inférence de types, nous ajoutons les enregistrements dans la syntaxe des schémas. Un schéma enregistrement a la forme $R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)$ où L est un ensemble fini d'étiquettes et E est un ensemble fini de types. Sa sémantique est donnée par :

$$\{\!|R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)|\!\} = \{s \mid \exists (t_l) \in \prod_{l\in L} \{\!|\mathbb{t}_l|\!\}. \exists t_0 \in \{\!|\mathbb{t}_0|\!\}. R((t_L)_{l\in L}; t_0; E) \leq s\}$$

On peut étendre la preuve du Lemme 5.26 en s'inspirant des Lemmes 9.3 et 9.4. La propriété sur le nombre de constructeurs $\otimes$ imbriqué s'étend au nombre de constructeurs enregistrement imbriqués. Pour prouver le Lemme 5.27, on utilise :

$$\mathbb{0} \in \{\!|R((\mathbb{t}_l)_{l\in L}; \mathbb{t}_0; E)|\!\} \iff (\exists l. \mathbb{0} \in \{\!|\mathbb{t}_l|\!\}) \vee (\mathbb{0} \in \{\!|\mathbb{t}_0|\!\}) \vee (\exists s \in E. \mathbb{0} \in \{\!|s \between \mathbb{t}_0|\!\})$$

et une induction sur le nombre de constructeurs enregistrement imbriqués.

La règle pour les enregistrements dans l'algorithme de typage est donnée par :

$$\mathbb{\Gamma}[\rho] := R((\mathbb{\Gamma}[\rho(l)])_{l\in \mathrm{Dom}(\rho)}; \mathbb{\Gamma}[\mathtt{def}(\rho)]; \emptyset)$$

9.4 Opérateur de concaténation

Nous définissons un opérateur de concaténation (ou fusion) d'enregistrements. Il s'agit d'un opérateur binaire $\rho \oplus_t \rho'$ où le type t permet de sélectionner

lequel des deux arguments fournit la valeur pour chaque étiquette. On s'autorise à écrire l'expression $\rho \oplus_t \rho'$ au lieu de $\oplus_t((\rho, \rho'))$.

La sémantique est donnée par :

$$(\rho, \rho') \overset{\oplus_t}{\rightsquigarrow} \rho''$$

où $\rho''(l) = \rho(l)$ si $\rho(l) \notin [\![t]\!]_{\mathcal{V}}$ et $\rho''(l) = \rho'(l)$ si $\rho(l) \in [\![t]\!]_{\mathcal{V}}$. Autrement dit, les champs de ρ qui sont de type t sont remplacés par ceux correspondant dans ρ'. Les cas d'erreur sont donnés par :

$$v \overset{\oplus_t}{\rightsquigarrow} \Omega$$

lorsque v n'est pas un couple d'enregistrements.

Theorème 9.6 *Il n'existe pas en général de typage exact pour l'opérateur* $\oplus_t$.

Preuve: Soit c_0, c_1, c_2, c_3, c_4 cinq constantes différentes. Posons :

$$t_0 = R(\emptyset; b_{c_0} \vee b_{c_1}; \emptyset) \times R(\emptyset; b_{c_2} \vee b_{c_3} \vee b_{c_4}; \{b_{c_3}, b_{c_4}\})$$

Les valeurs de t_0 sont donc des couples d'enregistrements (ρ, ρ') où ρ n'a que des c_0 ou c_1 comme valeurs et ρ' n'a que des c_2,c_3 ou c_4, avec au moins un c_3 et au moins un c_4. Soit $t = b_{c_1}$. Si $\oplus_t$ avait un typage exact, nous pourrions trouver un type s_0 qui représente exactement tous les enregistrements que l'on peut obtenir en combinant avec $\oplus_t$ un couple (ρ, ρ') comme ci-dessus. Prenons l_0 et l_1 deux étiquettes hors de $\mathcal{L}(s_0)$. L'enregistrement $\rho_1'' = \{l_0 = c_0; l_1 = c_0; \_ = c_2\}$ est dans s_0 car on peut l'obtenir avec $\rho = \{l_0 = c_0; l_1 = c_0; \_ = c_1\}$ et $\rho' = \{l_0 = c_3; l_1 = c_4; \_ = c_2\}$. Par contre, l'enregistrement $\rho_2'' = \{l_0 = c_0; \_ = c_2\}$ n'est pas dans s_0. En effet, si on l'obtenait à partir de ρ et ρ', on aurait $\rho'(l) = c_2$ sauf peut-être pour $l = l_0$, ce qui empêche d'y trouver c_3 et c_4 comme attendu. Le fait que ρ_1'' est dans s_0 et ρ_2'' ne l'est pas donne une contradiction avec le lemme ci-dessous. □

Lemme 9.7 *Soit t un type, et $L \supseteq \mathcal{L}(t)$. Soient ρ, ρ' deux valeurs enregistrement qui coïncident sur L et qui ont la même image directe pour $\mathcal{L}\backslash L$ ($\rho(\mathcal{L}\backslash L) = \rho'(\mathcal{L}\backslash L)$). Alors : $\rho \in [\![t]\!] \iff \rho' \in [\![t]\!]$.*

Preuve: C'est trivial lorsque t est un type enregistrement atomique, et la propriété est stable par combinaisons booléennes. □

Nous devons renoncer à un typage exact, et introduire une approximation pour typer cet opérateur. Nous voyons dans la preuve du Théorème 9.6 que l'impossibilité d'avoir un typage exact provient des contraintes existentielles E dans les $R((t_l)_{l \in L}; t_0; E)$. Dans la mesure où ces contraintes ne donnent aucune information lorsque l'on extrait la valeur d'un champ, il semble raisonnable de les oublier lorsque l'on applique l'opérateur $\oplus_t$. Une idée naturelle pour définir cette approximation est d'associer à un type $t \leq \mathbb{1}_{\mathbf{rec}}$ le type $\widetilde{t}$ défini par :

$$\widetilde{t} = \bigvee_{((t_l)_{l \in L}, t_0, E) \in \pi^L_{\mathbf{rec}}(t)} R((t_l)_{l \in L}; t_0; \emptyset)$$

où $L = \mathcal{L}(t)$, c'est-à-dire que l'on supprime les contraintes existentielles E. En fait, chaque terme de cette réunion peut être vu comme un atome enregistrement. Nous notons $P(t)$ l'ensemble de ces atomes :

$$\widetilde{t} = \bigvee_{r \in P(t)} r$$

Cependant, une telle définition est délicate à manier. En particulier, il semble difficile de montrer directement dessus des propriétés simple de l'opérateur $t \mapsto \widetilde{t}$, comme sa monotonie (et même l'invariance par équivalence de types). Pour pouvoir raisonner de manière similaire au Théorème 4.42, nous allons adopter une approche sémantique en décrivant l'approximation sous la forme d'un opérateur de saturation sur la sémantique des types.

Definition 9.8 (Adhérence) *Soit X un ensemble de valeurs enregistrement. Nous définissons son* **adhérence** *$\widetilde{X}$ comme l'ensemble des valeurs enregistrement qui coïncident avec un certain élément de X sur un ensemble fini arbitraire d'étiquettes :*

$$\widetilde{X} = \{\rho \mid \forall L \in \mathcal{P}_f(\mathcal{L}).\exists \rho' \in X.\forall l \in L.\ \rho(l) = \rho'(l)\}$$

Lemme 9.9 *Si X et Y sont deux ensembles de valeurs enregistrement, alors :*
- $X \subseteq \widetilde{X}$
- $X \subseteq Y \Longrightarrow \widetilde{X} \subseteq \widetilde{Y}$
- $\widetilde{\widetilde{X}} = \widetilde{X}$
- $\widetilde{\emptyset} = \emptyset$
- $\widetilde{X \cup Y} = \widetilde{X} \cup \widetilde{Y}$

> *Preuve*: Le seul point qui n'est pas immédiat est l'inclusion $\subseteq$ dans la dernière égalité. Soit ρ dans $\widetilde{X \cup Y}$. Supposons que ρ n'est pas dans $\widetilde{X}$. On dispose donc de $L_0 \in \mathcal{P}_f(\mathcal{L})$ pour lequel il est impossible de trouver $\rho' \in X$ qui coïncide avec ρ sur L_0. Montrons alors que ρ est dans $\widetilde{Y}$. Soit $L \in \mathcal{P}_f(\mathcal{L})$. Comme ρ est dans $\widetilde{X \cup Y}$, on peut trouver $\rho' \in X \cup Y$ qui coïncide avec ρ sur $L \cup L_0$, et donc *a fortiori* sur L_0, ce qui garantit que ρ' n'est pas dans X, et donc il est dans Y. □

Lemme 9.10 *Soit $t = R((t_l)_{l \in L}; t_0; E)$ et $t' = R((t_l)_{l \in L}; t_0; \emptyset)$. Si $t \not\simeq \mathbb{0}$, alors :*

$$\widetilde{[\![t]\!]_{\mathcal{V}}} = [\![t']\!]_{\mathcal{V}}$$

> *Preuve*: Pour prouver l'inclusion $\subseteq$, compte-tenu de $t \leq t'$, il suffit de constater que $[\![t']\!]_{\mathcal{V}}$ est clos par adhérence. En effet, si ρ est dans l'adhérence de cet ensemble, alors, pour une étiquette l arbitraire, en prenant $L' = \{l\}$ dans la définition de l'adhérence, on voit que $\rho(l) \in [\![t_l]\!]$ (pour $l \in L$) ou $\rho(l) \in [\![t_0]\!]$ (pour $l \notin L$), ce qui prouve que ρ est dans t'.
>
> Prouvons maintenant que toute valeur enregistrement ρ de type t' est dans l'adhérence du type t. Soit L' un ensemble fini d'étiquette. On peut supposer que $L \subseteq L'$. Pour chaque type $s \in E$,

on choisit une étiquette $l_s \notin L'$ et une valeur v_s de type $s \wedge t_0$ (c'est possible car $t \not\simeq \mathbb{0}$). On suppose les étiquettes l_s deux-à-deux distinctes. On définit alors ρ' comme la valeur enregistrement qui coïncide avec ρ sauf sur les l_s, et $\rho'(l_s) = v_s$. Cet enregistrement est dans $[\![t]\!]$. □

Avec les Lemmes 9.9 et 9.10, on voit que la définition syntaxique de $\widetilde{t}$ coïncide, au niveau sémantique, avec l'opérateur d'adhérence.

Corollaire 9.11 *Soit $t \leq \mathbb{1}_{\mathbf{rec}}$. Alors $[\![\widetilde{t}]\!] = \widetilde{[\![t]\!]}$.*

Ce corollaire donne, avec le Lemme 9.9, des propriétés sur l'opérateur $t \mapsto \widetilde{t}$ telles que sa monotonie par rapport à la relation de sous-typage.

On peut voir la sémantique de l'opérateur $\oplus_t$ comme une fonction $\mathcal{V} \to \mathcal{V}_\Omega$. Pour un ensemble X de valeurs, notons $\oplus_t(X)$ son image directe par cette fonction, c'est-à-dire $\{v' \mid v' \in \mathcal{V}_\Omega, \exists v \in X.\ v \overset{\oplus_t}{\rightsquigarrow} v'\}$.

Étudions cette fonction lorsque X est un produit de types enregistrements atomiques. Pour deux types t_1, t_2, nous notons $f_t(t_1, t_2) = t_1$ si $t_1 \wedge t \simeq \mathbb{0}$, et $f_t(t_1, t_2) = (t_1 \setminus t) \vee t_2$ sinon. On étend cette fonction point-à-point aux atomes enregistrements $f_t(r_1, r_2) = (l \mapsto f_t(r_1(l), r_2(l)))$.

Lemme 9.12 *Soient t_1, t_2 deux types non vides et v une valeur de type $f_t(t_1, t_2)$. Alors il existe deux valeurs $v_1 \in [\![t_1]\!], v_2 \in [\![t_2]\!]$ telles que $v_1 \in [\![t]\!] \Rightarrow v = v_1$ et $v_1 \notin [\![t]\!] \Rightarrow v = v_2$.*

Lemme 9.13 *Si $r_1 \not\simeq \mathbb{0}, r_2 \not\simeq \mathbb{0}$, alors :*

$$\oplus_t([\![r_1]\!] \times [\![r_2]\!]) = [\![f_t(r_1, r_2)]\!]$$

Preuve: Montrons d'abord l'inclusion $\subseteq$. Soit v une valeur dans $[\![r_1]\!] \times [\![r_2]\!]$. C'est un couple (ρ_1, ρ_2), avec, pour tout l, $\rho_i(l)$ de type $r_i(l)$. Lorsque l'on applique l'opérateur $\oplus_t$, on obtient un enregistrement ρ, et il s'agit de vérifier que $\rho(l)$ est de type $r(l) = f_t(r_1(l), r_2(l))$. Si $\rho_1(l)$ est de type t, alors $r_1(l) \wedge t \not\simeq \mathbb{0}$, donc $r_2(l) \leq r(l)$, et l'on a de plus $\rho(l) = \rho_2(l)$, qui est donc bien de type $r(l)$. Si $\rho_1(l)$ n'est pas de type t, on a $\rho(l) = \rho_1(l)$, qui est de type $r_1(l)$, mais pas de type t, or $r_1(l) \setminus t$ est forcément sous-type de $r(l)$.

Passons à la preuve de l'inclusion $\supseteq$. Soit ρ un enregistrement de type $f_t(r_1, r_2)$. Le lemme ci-dessus permet de définir deux enregistrements ρ_1 et ρ_2, respectivement de type r_1 et r_2, tels que $(\rho_1, \rho_2) \overset{\oplus_t}{\rightsquigarrow} \rho$. □

Étudions maintenant l'ensemble $\oplus_t([\![t_1]\!] \times [\![t_2]\!])$ pour deux sous-types t_1,t_2 de $\mathbb{1}_{\mathbf{rec}}$. L'idée est d'obtenir un résultat exact modulo adhérence. Nous allons voir que l'on peut faire cette approximation avant ou après l'application de l'opérateur $\oplus_t$ sans changer le résultat.

Lemme 9.14 *Soient X_1, X_2 deux ensembles d'enregistrements. Alors :*

$$\oplus_t(\widetilde{X_1} \times \widetilde{X_2}) \subseteq \oplus_t(\widetilde{X_1 \times X_2})$$

Preuve: Soit $\rho = \oplus_t(\rho_1, \rho_2)$ un élément du membre gauche, avec $\rho_i \in \widetilde{X_i}$. Montrons qu'il est dans l'adhérence de $\oplus_t(X_1 \times X_2)$. Soit L un ensemble fini d'étiquettes. On peut trouver $\rho'_i \in X_i$ qui coïncide avec ρ_i sur L. On pose alors $\rho' = \oplus_t(\rho'_1, \rho'_2)$, qui coïncide avec ρ sur L. □

Lemme 9.15 *Soient t_1, t_2 deux sous-types de $\mathbb{1}_{\mathbf{rec}}$ et $X = \oplus_t(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)$. Alors $\widetilde{X} = \oplus_t(\widetilde{\llbracket t_1 \rrbracket} \times \widetilde{\llbracket t_2 \rrbracket})$.*

Preuve: Posons $X_i = \llbracket t_i \rrbracket$. L'inclusion $\supseteq$ est une conséquence du lemme ci-dessus. Dans la mesure où $X_1 \times X_2 \subseteq \widetilde{X_1} \times \widetilde{X_2}$, pour prouver l'inclusion $\subseteq$, il suffit de voir que le membre droit est clos par adhérence. Cette propriété étant stable par réunion sur t_1 ou sur t_2, on peut se ramener au cas où $t_1 = R((t_l^1)_{l\in L}; t_0^1; E^1)$ et $t_2 = R((t_l^2)_{l\in L}; t_0^2; E^2)$ pour un certain L (avec $t_i \not\simeq \mathbb{0}$) et alors l'adhérence de t_i est représentée par un atome enregistrement $r_i = R((t_l^i)_{l\in L}; t_0^i; \emptyset)$ (Lemme 9.10), et l'on a vu que $\oplus_t(\llbracket r_1 \rrbracket \times \llbracket r_2 \rrbracket)$ est lui-même représenté par un atome enregistrement (Lemme 9.13), et il est donc stable par adhérence. □

Nous pouvons maintenant proposer un typage (non exact) de l'opérateur $\oplus_t$. La relation $\oplus_t : \_\!\rightarrow\!\_$ est définie par le système inductif de la Figure 9.1.

$$\frac{}{\oplus_t : (r_1 \times r_2) \rightarrow f_t(r_1, r_2)}\ (\oplus_t\{\})$$

$$\frac{\oplus_t : t_1 \rightarrow s_1 \quad \oplus_t : t_2 \rightarrow s_2}{\oplus_t : (t_1 \wedge t_2) \rightarrow (s_1 \wedge s_2)}\ (\oplus_t\wedge)$$

$$\frac{\oplus_t : t_1 \rightarrow s_1 \quad \oplus_t : t_2 \rightarrow s_2}{\oplus_t : (t_1 \vee t_2) \rightarrow (s_1 \vee s_2)}\ (\oplus_t\vee)$$

$$\frac{\oplus_t : t' \rightarrow s' \quad t \leq t' \quad s' \leq s}{\oplus_t : t \rightarrow s}\ (\oplus_t \leq)$$

FIG. 9.1 – Axiomatisation du typage de l'opérateur $\oplus_t$

Lemme 9.16 *L'opérateur $\oplus_t$ est bien typé.*

Preuve: Comme pour la preuve du Lemme 5.35, il suffit de considérer la règle $(\oplus_t\{\})$. Ce cas est donné par l'inclusion $\subseteq$ du Lemme 9.13. □

Lemme 9.17 *Soient t_0 et s_0 deux types. Les assertions suivantes sont équivalentes :*

(i) $\oplus_t : t_0 \rightarrow s_0$

(ii) $\widetilde{\oplus_t(\llbracket t_0 \rrbracket)} \subseteq \llbracket s_0 \rrbracket$

(iii) $\begin{cases} t_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}} \\ \forall (t_1, t_2) \in \pi(t_0).\ \forall r_1 \in P(t_1), r_2 \in P(t_2).\ f_t(r_1, r_2) \leq s_0 \end{cases}$

Preuve: Prouvons d'abord $(i) \Rightarrow (ii)$ par induction sur la dérivation de $\oplus_t : t_0 \rightarrow s_0$. Pour la règle $(\oplus_t\{\})$, on obtient en fait une égalité avec le Lemme 9.13. Pour la règle $(\oplus_t\vee)$, on utilise le fait que le membre gauche de (ii) commute avec la réunion. Pour les deux autres règles, on utilise la monotonie du membre gauche.

Prouvons $(ii) \Rightarrow (iii)$. Si l'on n'avait pas $t_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}}$, alors $\oplus_t(\llbracket t_0 \rrbracket)$ contiendrait Ω, ce qui est exclus. Soit $(t_1, t_2) \in \pi(t_0)$. On a $t_1 \times t_2 \leq t_0$, et donc $\oplus_t(\widetilde{\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket}) \subseteq \llbracket s_0 \rrbracket$. D'après le Lemme 9.15 et le Corollaire 9.11, le membre gauche est $\oplus_t(\llbracket \widetilde{t_1} \times \widetilde{t_2} \rrbracket)$. Si $r_i \in P(t_i)$, alors $r_i \leq \widetilde{t_i}$, et donc $\oplus_t(\llbracket r_1 \rrbracket \times \llbracket r_2 \rrbracket) \leq \llbracket s_0 \rrbracket$ et le membre gauche est $\llbracket f_t(r_1, r_2) \rrbracket$, d'où l'on déduit bien $f_t(r_1, r_2) \leq s_0$.

Prouvons enfin $(iii) \Rightarrow (i)$. On écrit $t = \bigvee_{(t_1,t_2)\in\pi(t_0)} t_1 \times t_2$. En écrivant

$$t_i \leq \widetilde{t_i} = \bigvee_{r_i \in P(t_i)} r_i$$

on voit que :

$$t \leq \bigvee_{\substack{(t_1,t_2)\in\pi(t) \\ r_1\in P(t_1) \\ r_2\in P(t_2)}} r_1 \times r_2$$

Avec la règle $(\oplus_t\{\})$ et la règle $(\oplus_t \leq)$, on obtient $\oplus_t : (r_1 \times_2) \rightarrow s_0$, et l'on conclut avec la règles $(\oplus_t\vee)$. Le cas $t_0 \leq \mathbb{0}$ se traite à part, avec la règle $(\oplus_t\{\})$ appliquée à des atomes enregistrement vides. □

Comme pour π_1, l'assertion (iii) permet de déduire une fonction de typage pour $\oplus_t$ sur les schémas. On étend la fonction π aux schémas $\mathbb{t}$ tels que $\mathbb{t} \leq \mathbb{1}_{\mathbf{prod}}$ par

$$\begin{array}{lcl} \pi(\mathbb{t}_1 \vee \mathbb{t}_2) & = & \pi(\mathbb{t}_1) \cup \pi(\mathbb{t}_2) \\ \pi(\mathbb{t}_1 \otimes \mathbb{t}_2) & = & \{(\mathbb{t}_1, \mathbb{t}_2)\} \end{array}$$

La fonction P associe à tout type $t \leq \mathbb{1}_{\mathbf{rec}}$ un ensemble fini de types enregistrement atomiques, c'est-à-dire de fonctions presque constantes $\mathcal{L} \rightarrow \widehat{T}$. Nous l'étendons pour définir $P(\mathbb{t})$, ensemble fini de fonctions presque constantes $\mathbb{r}$ des étiquettes dans les schémas (on identifie une telle fonction à un $R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; \emptyset)$) :

$$\begin{array}{lcl} P(\mathbb{t}_1 \vee \mathbb{t}_2) & = & P(\mathbb{t}_1) \cup P(\mathbb{t}_2) \\ P(R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; E) & = & \{R((\mathbb{t}_l)_{l\in L'}; \mathbb{t}_0; \emptyset)\} \end{array}$$

Nous étendons aussi la fonction f_t aux couples de schémas enregistrement $(\mathbb{r}_1, \mathbb{r}_2)$, de manière naturelle. On peut alors donner une définition pour $\oplus_t[\mathbb{t}]$:

$$\oplus_t[\mathbb{t}] = \begin{cases} \bigvee_{\substack{(\mathbb{t}_1,\mathbb{t}_2)\in\pi(\mathbb{t}) \\ \mathbb{r}_1\in P(\mathbb{t}_1) \\ \mathbb{r}_2\in P(\mathbb{t}_2)}} f_t(\mathbb{r}_1, \mathbb{r}_2) & \text{si } \mathbb{t}_0 \leq \mathbb{1}_{\mathbf{rec}} \times \mathbb{1}_{\mathbf{rec}} \\ \Omega & \text{sinon} \end{cases}$$

9.5 Fonctions partielles

Le formalisme que nous avons adopté pour présenter les enregistrements est celui des fonctions presque constantes. Par rapport à celui des fonctions à domaine fini, il a l'avantage d'être uniforme et de simplifier les énoncés. Dans cette section, nous montrons comment encoder des fonctions à domaine fini dans les enregistrements.

Nous choisissons une constante $\mu \in \mathcal{C}$ qui représente la valeur des champs d'enregistrement qui sont en fait non définis. Pour les expressions, on écrit simplement $\{l_1 = e_1; \ldots; l_n = e_n\}$ au lieu de $\{l_1 = e_1; \ldots; l_n = e_n; \_ = \mu\}$. Pour les types, on écrit $\{l_1 = t_1; \ldots; l_n = t_n\}$ pour $\{l_1 = t_1; \ldots; l_n = t_n; \_ = \mathbb{1}\}$ (type enregistrement ouvert) et $\{|l_1 = t_1; \ldots; l_n = t_n|\}$ pour $\{l_1 = t_1; \ldots; l_n = t_n; \_ = b_\mu\}$ (type enregistrement fermé), et de même pour les motifs. À la place d'une spécification de champ dans un type enregistrement, on autorise également à écrire $l_i : t_i$ pour $l_i = t_i \backslash b_\mu$, et $l_i :? t_i$ pour $l_i = t_i \vee b_\mu$, et de même pour les motifs : $l_i : p_i$ signifie $l_i = p_i \& \neg b_\mu$. Ainsi, le motif $\{|l_1 : x;\ l_2 = ((y \& \neg b_\mu) | (y := c))|\}$ accepte tout enregistrement défini sur l_1, éventuellement sur l_2, et nulle part ailleurs, et capture la valeur pour l_1 dans x et celle pour l_2 dans y si ce champ est défini, et sinon associe la constante c à y.

Opérateurs dérivés On peut également définir un certain nombre d'opérateurs dérivés de $\oplus_t$. Pour une constante c, on note $\oplus_c$ au lieu de $\oplus_{b_c}$. On se donne une constante c_0 différente de μ (elle ne sert qu'à donner la sémantique des nouveaux opérateurs, et cette sémantique ne dépend pas du choix de c_0). Soit $L = \{l_1, \ldots, l_n\}$ un ensemble fini d'étiquettes. L'opérateur de restriction $_{|L}$ est défini par :

$$e_{|L} := \{l_1 = c_0; \ldots; l_n = c_0; \_ = \mu\} \oplus_{c_0} e$$

L'opérateur de suppression $_{\backslash L}$ est défini par :

$$e_{\backslash L} := \{l_1 = \mu; \ldots; l_n = \mu; \_ = c_0\} \oplus_{c_0} e$$

Pour un type t, l'opérateur de restriction sur t est défini par :

$$e_{|t} := r \oplus_{\neg t} \{\_ = \mu\}$$

Cet opérateur supprime les champs qui ne sont pas de type t. Enfin, on note simplement $r_1 \oplus r_2$ pour $r_2 \oplus_\mu r_1$. C'est un opérateur de concaténation, qui prend, en cas de conflit pour une étiquette (si les deux arguments sont définis sur cette étiquette), la valeur du deuxième argument.

Domaine fini Si l'on ne s'autorise à écrire que des expressions enregistrement de la forme $\{l_1 = e_1; \ldots; l_n = e_n\}$, c'est-à-dire des enregistrements à domaine fini (qui valent μ presque partout), alors la relation de sous-typage n'est plus complète par rapport au modèle des valeurs (formellement : l'ensemble des valeurs n'est plus un modèle). En effet, le type $\{\_ = t\}$ ne contient aucune valeur dès que $t \wedge b_\mu \simeq \mathbb{0}$. Il est très facile de modifier la définition d'un modèle pour corriger cela : il suffit de prendre, au lieu de $\mathcal{L} \xrightarrow{c} D$, l'ensemble des fonctions qui valent μ presque partout et cela change alors légèrement la relation de sous-typage (dans la définition de l'ensemble $\mathbb{E}\mathcal{S}$, on remplace $(\bigwedge_{r \in P} \tau(\mathtt{def}(r))) \in \mathcal{S}$ par $(b_\mu \wedge \bigwedge_{r \in P} \tau(\mathtt{def}(r))) \in \mathcal{S}$). Le type $\{l_1 = t_1; \ldots; l_n = t_n; \_ :? t\}$ signifie

alors qu'en dehors des étiquettes l_i, l'enregistrement peut être défini sur un certain nombre fini d'autres étiquettes, et que les valeurs de ces champs sont nécessairement dans t.

Chapitre 10

Présentation du langage

Dans ce chapitre, nous présentons ℂDuce, un langage de programmation construit sur les bases théoriques présentées dans les chapitres précédents. ℂDuce possède les traits d'un langage de programmation fonctionnel généraliste, mais il a été conçu plus particulièrement pour développer des applications qui manipulent de manière sûre des documents XML.

Cette thèse n'étant pas destinée à fournir un manuel de référence ou un manuel d'utilisateur pour ℂDuce, et puisque le langage continue d'évoluer, nous n'allons pas spécifier formellement sa syntaxe concrète ou sa sémantique, mais présenter ses caractéristiques via un codage dans le calcul du Chapitre 5, étendu avec le filtrage (Chapitre 6) et les enregistrements (Chapitre 9).

La syntaxe pour les fonctions en ℂDuce combine en fait abstraction et filtrage :

$$\texttt{fun}\ f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n)\ p_1 \rightarrow e_1 \mid \ldots \mid p_m \rightarrow e_m$$

L'identificateur f peut être omis si la fonction n'est pas récursive.

10.1 Types de base, constantes

Nous devons tout d'abord instancier le cadre générique en spécifiant les types de base et les constantes. Les constantes sont de trois genres : entiers, atomes, caractères.

Entiers Nous utilisons des entiers à précision arbitraire. L'ensemble des constantes entier est donc l'ensemble $\mathbb{Z}$ des entiers relatifs. Nous notons `Int` le type de base qui dénote tous ces entiers : $\mathbb{B}[\![\texttt{Int}]\!] = \mathbb{Z}$. Nous introduisons également des types de base intervalles fermés $i—j$ où $i, j \in \mathbb{Z}$, avec $\mathbb{B}[\![i—j]\!] = \{n \mid i \leq n \leq j\}$, ainsi que les intervalles ouverts $i—\infty$ avec $\mathbb{B}[\![i—\infty]\!] = \{n \mid i \leq n\}$ et $\infty—j$ avec $\mathbb{B}[\![\infty—j]\!] = \{n \mid n \leq j\}$. Un type singleton $i—i$ est noté simplement i.

Tout sous-type t de `Int` peut s'écrire sous la forme d'une réunion finie de types intervalles. Cette écriture n'est pas unique. Il existe cependant une décomposition canonique, que l'on peut caractériser de deux manières différentes :

- les intervalles dans la décomposition sont maximaux parmi les intervalles sous-types de t ;
- le nombre d'intervalle dans la décomposition est minimal.

En pratique, partant d'une décomposition, on obtient cette décomposition canonique en fusionnant (de manière itérative) deux intervalles qui ont une intersection non vide ou qui se « touchent » (comme i—j et $(j+1)$—k).

Le langage possède les opérateurs arithmétiques classiques. L'opérateur d'addition, par exemple, possède un typage exact, alors que l'opérateur de multiplication n'en a pas. Par exemple, la multiplication d'un entier de type `Int` et d'un entier de type 2 donne sémantiquement l'ensemble des entiers pairs, qui n'est pas représentable par un type. On peut évidemment donner le type `Int` systématiquement au résultat, ou prendre un type plus précis. Par exemple, on peut faire de l'arithmétique d'intervalles, en spécifiant que l'on prend la décomposition en intervalles maximaux.

Caractères Les constantes caractère représentent des éléments du jeu de caractères Unicode. Chaque constante représente un *code point* de cette spécification, et est donc identifiable à un entier. Nous notons `Char` le type de tous les caractères. Son interprétation est l'ensemble des *code points* Unicode. Nous introduisons également des types de base intervalles qui représentent chacun un segment du jeu Unicode. Le type singleton associé à une constante atome est un type intervalle. Les caractères sont notés entre des guillemets simples ($'\texttt{a}', '\texttt{b}', \ldots$).

Atomes Les atomes sont utilisés, entre autres, pour représenter les étiquettes (*tags*) des éléments XML. Un atome est couple $`ns : ln$ où $ns$ est un *namespace* XML (c'est-à-dire une chaîne de caractères Unicode éventuellement vide), et $ln$ est un nom local, au sens de la spécification XML Namespaces [Nam]. Lorsque $ns$ est le *namespace* vide, nous notons simplement $`ln$ l'atome $`ns : ln$. Nous notons `Atom` le type de tous les atomes, $`ns : *$ celui des atomes dans le *namespace* ns, et $`ns : ln$ celui qui dénote l'unique atome $`ns : ln$.

Dans les programmes ℂDuce, les *namespaces* ne sont pas écrits directement dans les constantes atome. Le langage utilise un mécanisme semblable à la spécification XML Namespaces pour permettre de définir des préfixes, qui dénotent de manière compacte des *namespaces*. Ce sont ces préfixes qui sont utilisés pour écrire concrètement les constantes atome dans les programmes ℂDuce.

10.2 Types, motifs

L'algèbre des types (Chapitre 3) et l'algèbre des motifs (Chapitre 6) ont de nombreux constructeurs similaires : réunion, intersection, produit, enregistrement. Un motif p sans variables de capture ($\mathrm{Var}(p)$) peut être naturellement identifié au type $⌈p⌋$, et tout type t peut être vu comme un motif. Il est naturel d'unifier syntaxiquement ces deux algèbres, et le langage utilise effectivement une syntaxe externe commune pour représenter ces deux algèbres. Les connecteurs réunion, intersection, différence sont notés respectivement `|`,`&` et `\`. Le constructeur produit est noté (_, _) (au lieu de _`×`_ pour les types). Les types $\mathbb{0}$ et $\mathbb{1}$ sont notés `Empty` et `Any`. Nous utilisons également la notation _ au lieu de `Any` (spécialement dans les motifs).

Un terme qui ne contient pas de variables de captures peut être utilisé comme un type. Pour utiliser un terme de cette algèbre comme un motif, il faut vérifier les conditions de bonne formation sur les variables de capture. Un sous-terme

de la forme $p_1 \rightarrow p_2$ ne doit pas contenir de variable, ce qui permet en fait de le voir comme un motif contrainte de type. De même, dans $p_1 \backslash p_2$, le terme p_2 ne doit pas contenir de variable.

Un terme comme (`Int`, `Int`), utilisé en position de motif, peut être interprété de deux manières différentes : soit comme un motif contrainte de type (le type étant un type produit), soit comme un motif produit dont chaque composante est un motif contrainte de type. Ce genre d'ambiguïté entre types et motifs ne pose aucun problème (les différentes interprétations ont la même sémantique, et donc le même typage).

Le langage CDuce utilise la relation de sous-typage $\leq$ induite par un modèle universel (Section 4.5). Cette relation de sous-typage peut être calculée par les algorithmes de Chapitre 7.

Enregistrements Nous utilisons le formalisme de la Section 9.5 [1] . Les seules expressions enregistrement autorisées sont de la forme $\{l_1 = e_1; \ldots; l_n = e_n\}$. Pour la constante μ, qui représente les champs non définis des enregistrements, nous prenons l'atome ‘`missing`. Pour l'ensemble des étiquettes $\mathcal{L}$, nous prenons l'ensemble des constantes atome (notées sans le ‘).

Nous notons $e.l$ l'accès au champ l. C'est du sucre syntaxique pour :

$$\texttt{match}\ e\ \texttt{with}\ \{l = x\} \rightarrow x$$

Éléments XML Outre les fonctions, couples, enregistrements, et constantes, le langage manipule une autre catégorie de valeurs : les éléments XML. Une expression qui dénote un telle valeur a la forme $< (e_1)\ e_2 > e_3$. Elle se comporte comme un simple produit cartésien à trois termes. On pourrait l'identifier au couple imbriqué $(e_1, (e_2, e_3))$, mais on préfère introduire une nouvelle catégorie (de valeurs, types, motifs), pour éviter les ambiguïtés. Lorsque e_1 est une constante atome ‘$ns : ln$, on peut l'écrire sans le ‘ initial : $< ns : ln\ e_2 > e_3$. Lorsque e_2 est une expression enregistrement $\{\ldots\}$, on peut omettre les { et } : $< (e_1)\ \ldots > e_3$, ainsi que les point-virgules qui séparent les différents champs. On dispose évidemment de types et motifs qui correspondent à cette nouvelle catégorie de valeurs et d'expressions : $< (p_1)\ p_2 > p_3$, avec les mêmes simplifications. Le sous-typage, le typage et le filtrage dans cette nouvelle catégorie de valeurs se déduit facilement de la théorie introduite pour les produits.

Récursion Les types et motifs récursifs peuvent être écrits sous la forme t_0 `where` $\texttt{X}_1 = t_1$ `and` $\ldots$ `and` $\texttt{X}_n = t_n$ où les $\texttt{X}_i$ sont des lieurs de récursion qui peuvent être utilisés à l'intérieur de t_0 et des t_i. Pour les types, le langage autorise également des déclarations globales mutuellement récursives :

`type` $\texttt{X}_1$ `=` t_1
$\ldots$
`type` $\texttt{X}_n$ `=` t_n

Les récursions doivent être bien formées (tout cycle de récursion doit traverser au moins un constructeur flèche, produit, XML, enregistrement).

[1] En fait l'implémentation est légèrement différente : l'absence des champs n'est pas représentée par une valeur du langage (les enregistrements sont des fonctions partielles).

10.3 Séquences, expressions régulières

Un certain nombre de constructions syntaxiques dérivées et d'opérations sont introduites pour faciliter la manipulation de séquences de valeurs.

10.3.1 Séquences

Les séquences sont encodées par des couples imbriqués et nous utilisons la constante atome `'nil` pour représenter la séquence vide. Une expression séquence $[e_1 \dots e_n]$ est ainsi traduite en l'expression $(e_1, \dots, (e_n, \text{`}\texttt{nil}) \dots)$.

10.3.2 Expressions régulières de types et de motifs

Pour décrire précisément le type des séquences, et pour pouvoir effectuer des extractions par filtrage dessus, nous introduisons des expressions régulières dans l'algèbre des types et des motifs. Ces termes sont notées $[R]$ où R est une expression régulière, c'est-à-dire un terme engendré par :

$$\begin{array}{rcl} R & := & p \\ & | & R_1 R_2 \\ & | & R_1 | R_2 \\ & | & R* \\ & | & R? \\ & | & R*? \\ & | & R?? \\ & | & x :: R \end{array}$$

Il ne s'agit que de sucre syntaxique, et nous allons montrer plus bas comment traduire un terme $[R]$ en un type ou un motif.

La notation $x :: R$ permet de capturer dans x toute la sous-séquence reconnue par R. Nous disons que x est une variable de capture de séquence. Lorsque la même variable de capture de séquence est utilisée plusieurs fois dans la même expression régulière, ou lorsqu'elle est utilisée sous un opérateur de répétition, la sémantique attendue est de concaténer toutes les sous-séquences correspondantes (les occurrences imbriquées ne sont pas autorisées).

Les deux constructeurs $*$ et $*?$ sont des étoiles de Kleene. La différence entre les deux est que le premier a un comportement glouton (*greedy*) - il essaie de faire autant d'itérations que possible - alors que le deuxième a au contraire un comportement paresseux (il essaie de faire aussi peu d'itérations que possible). De même, les deux constructeurs ? et ?? capturent de manière optionnelle leur argument : le premier essaie en priorité de la capturer, alors que le deuxième évite de le faire si c'est possible.

Remarque 10.1 *Vansummeren [Van04] a remarqué que la sémantique gloutonne pour l'étoile de Kleene ne simule pas une sémantique longest-match, comme on pourrait naïvement le croire. Considérons par exemple le motif* $p = [x :: (1|(1\ 2)) *\ 2?]$ *et la valeur* $v = [1\ 2]$. *Le résultat pour* x, *avec la sémantique gloutonne est* $[1]$, *car lors de la première itération dans l'étoile, la première branche de l'alternative* $(1|(1\ 2))$ *est essayée, réussie, et empêche de faire une deuxième itération, sans pour autant empêcher le filtrage de réussir globalement. Une sémantique longest-match pour l'étoile capturerait pour* x *toute la séquence*

[1 2]. *Vansummeren étudie de tels variantes de sémantique et les problèmes d'inférence de types associés.*

10.3.3 Traduction des expressions régulières

Pour décrire la traduction, nous allons généraliser la syntaxe $[R]$ et utiliser des termes de la forme $[R; p_1; p_2]$. La forme $[R]$ est traduite en $[R; \text{`}\texttt{nil}; \text{`}\texttt{nil}]$. Intuitivement, les motifs p_1 et p_2 représentent les « continuations », c'est-à-dire le motif à appliquer à ce qui reste de la séquence une fois que R a été reconnue sur un préfixe de la séquence. Le motif p_1 est utilisé lorsque R a consommé au moins un élément et le motif p_2 est utilisé lorsque R n'a reconnu aucun élément. Nous expliquerons plus bas la raison pour laquelle nous distinguons ces deux continuations.

La première étape de la traduction consiste à éliminer les variables de capture de séquence. Cela se fait en remplaçant $[R_0; p_1; p_2]$ par $[R'_0; p'_1; p'_2]$. L'expression régulière R'_0 est obtenue de la manière suivante. Si une sous-expression $x :: R$ apparaît dans R_0, elle est remplacée par R' où R' est obtenu à partir de R en remplaçant tous les motifs p qui apparaissent dedans par $(x\&p)$. Si $x_1, \ldots, x_n$ sont toutes les variables de capture de séquences qui apparaissent dans R_0, les continuations p'_1 et p'_2 sont définies par $p'_i = (x_1 := \text{`}\texttt{nil})\& \ldots \&(x_n := \text{`}\texttt{nil})\&p_i$.

Par exemple, la traduction de $[x :: (p_1 \quad y :: (p_2?)) \quad y :: p_3; \text{`}\texttt{nil}; \text{`}\texttt{nil}]$ est $[(x\&p_1)(x\&y\&p_2)?(y\&p_3); p'; p']$ avec $p' = (x := \text{`}\texttt{nil})\&(y := \text{`}\texttt{nil})\&\text{`}\texttt{nil}$ (on pourrait prendre $x\&y\&\text{`}\texttt{nil}$ qui est équivalent à p').

Voyons maintenant comment traduire $[R_0; p_1; p_2]$ lorsque R_0 ne contient pas de variable de capture de séquence. Nous définissons cette traduction sous la forme d'une fonction Ψ, par induction sur la structure de R_0 :

$$
\begin{array}{lcl}
\Psi[p; p_1; p_2] & = & (p, p_1) \\
\Psi[R_1 R_2; p_1; p_2] & = & \Psi[R_1; \Psi[R_2; p_1; p_1]; \Psi[R_2; p_1; p_2]] \\
\Psi[R_1 | R_2; p_1; p_2] & = & \Psi[R_1; p_1; p_2] | \Psi[R_2; p_1; p_2] \\
\Psi[R*; p_1; p_2] & = & \texttt{X} | p_2 \texttt{ where X} = \Psi[R; \texttt{X} | p_1; \texttt{Empty}] \\
\Psi[R*?; p_1; p_2] & = & p_2 | \texttt{X where X} = \Psi[R; p_1 | \texttt{X}; \texttt{Empty}] \\
\Psi[R?; p_1; p_2] & = & \Psi[R; p_1; p_2] | p_2 \\
\Psi[R??; p_1; p_2] & = & p_2 | \Psi[R; p_1; p_2]
\end{array}
$$

Dans ces définitions, le lieur X est frais. Compte-tenu de la politique *first-match* pour le filtrage, nous voyons que l'étoile $*$ a en effet un comportant glouton (elle capture autant d'éléments que possible avant de passer à la suite), et que $*?$ a un comportement paresseux. De même, l'expression régulière $R?$ essaie d'abord de reconnaître R avant d'y renoncer si c'est impossible, et l'expression régulière $R??$ essaie de passer à la suite avant de reconnaître R seulement si c'est nécessaire.

La construction est assez classique. La seule subtilité provient du fait que l'on doit produire des récursions bien formées, qui traversent un constructeur. C'est la raison pour laquelle nous distinguons les deux continuations p_1 et p_2. Si l'on n'avait qu'une seule continuation $[R; p]$, nous aurions posé :

$$\Psi[R*; p] = \texttt{X where X} = \Psi[R; X] | p$$

Or cela aurait donné lieu à une récursion mal formée lorsque R accepte la séquence vide. Par exemple, pour $R = [(p_1 * \; p_2*)*]$, nous aurions (en réarrangeant

un peu) :

$$\Psi[R;p] = \mathtt{X}_0 \texttt{ where } \begin{cases} \mathtt{X}_0 & = & \mathtt{X}_1 | p \\ \mathtt{X}_1 & = & (p_1, \mathtt{X}_1) | \mathtt{X}_2 \\ \mathtt{X}_2 & = & (p_2, \mathtt{X}_2) | \mathtt{X}_0 \end{cases}$$

ce qui donne un cycle de récursion qui ne traverse aucun constructeur. Le fait d'avoir deux continuations différentes permet d'imposer à chaque itération de R dans $[R*]$ de capturer au moins un élément (en terme d'automates, cela revient à éliminer les cycles d'ϵ-transition). Notre définition de Ψ n'introduit pas de récursion mal formée (car la continuation p_1 n'est utilisée que sous un constructeur). En reprenant l'exemple ci-dessous, nous obtenons (après de petits arrangements cosmétiques) :

$$\Psi[R;p;p] = \mathtt{X}_0 \texttt{ where } \begin{cases} \mathtt{X}_0 & = & \mathtt{X}_1 | \mathtt{X}_2 | p \\ \mathtt{X}_1 & = & (p_1, \mathtt{X}_0) \\ \mathtt{X}_2 & = & (p_2, \mathtt{X}_2 | \mathtt{X}_0) \end{cases}$$

Cette définition récursive est bien formée : les cycles de récursion croisent tous un constructeur.

Remarque 10.2 *Cette présentation de la traduction avec deux continuations est inspirée par une étude approfondie [FC04] du problème posé par les expressions régulières « dangereuses », c'est-à-dire avec un sous-terme de la forme $R*$, où R accepte la séquence vide.*

Autres constructeurs Nous introduisons également les constructeurs d'expression régulière dérivés $R+$ et $R+?$ définis par $R+ = R\ R*$ et $R+? = R\ R*?$. Nous pouvons également introduire un constructeur de prédicat $/p$, qui applique un certain motif sur la séquence courante sans en consommer d'élément. Sa traduction est donnée par :

$$\Psi[/p; p_1; p_2] = p \& p_2$$

On peut par exemple utiliser ce prédicat pour positionner dans une variable un indicateur qui indique le choix qui a été effectué dans une alternative :

$$[(p_1 * /(x := 1)) | (p_2 * /(x := 2))]$$

Dans cet exemple, la variable x sera liée à 1 (resp. 2) si c'est la première (resp. deuxième) branche qui a été choisie.

Exemple Nous allons illustrer par un exemple comment la sémantique choisie pour les motifs produit (q_1, q_2) lorsque la même variable apparaît dans les deux q_i permet de capturer des sous-séquences (non nécessairement consécutives).

Prenons $p = [((x :: t) | \_)*]$. Appliqué à une séquence, ce motif capture dans x la sous-séquence formée de tous les éléments qui sont de type t. La traduction de p est

$$\mathtt{X} \texttt{ where } \mathtt{X} = (x \& t, \mathtt{X}) | (\_, \mathtt{X}) | (x := \text{`}\mathtt{nil}) \& \text{`}\mathtt{nil}$$

Si on applique ce motif à une valeur $[v_1\ v_2\ v_3] = (v_1, (v_2, (v_3, \text{`}\mathtt{nil})))$ avec, par exemple $v_1, v_3$ de type $t$ et $v_2$ de type $\neg t$, on récupère pour $x$ le résultat $[v_1\ v_3] = (v_1, (v_3, \text{`}\mathtt{nil}))$.

Notons que l'on bénéficie automatiquement pour les motifs expression régulière de l'exactitude du typage du filtrage, y compris pour les variables de

capture de séquence, ainsi que de la compilation efficace du filtrage. Ainsi, si l'on applique le motif ci-dessus à une expression de type $[(\neg t)\ t*]$ et si l'on utilise la factorisation des variables de capture (Section 8.4) lors de la compilation, ce motif pourra être compilé comme $(\_, x)$ (c'est-à-dire la deuxième projection). Si on l'applique à une expression de type $[t_1\ (\neg t)\ t_2+]$, avec $t_1 \leq t$, $t_2 \leq t$, il pourra être compilé comme $(x, (\_, x))$, et le système de type saura que x a le type $[t_1\ t_2+]$.

10.3.4 Décompilation des types séquences

Definition 10.3 *Un* **automate** *est un quadruplet $\mathcal{A} = (Q, \delta, \mathfrak{q}_0, F)$ où Q est un ensemble fini d'états, $\delta \subseteq \in \mathcal{P}_f(Q \times \widehat{T} \times Q)$ est la relation de transition, $\mathfrak{q}_0$ est l'état initial et $F \subseteq Q$ est l'ensemble des états finaux.*

La sémantique $[\![\mathcal{A}]\!]_{\mathfrak{q}}$ d'un état $\mathfrak{q} \in Q$ est un ensemble de valeurs séquence, défini par :

$$\begin{array}{lcl} \text{`}\mathtt{nil} \in [\![\mathcal{A}]\!]_{\mathfrak{q}} & \iff & \mathfrak{q} \in F \\ (v_1, v_2) \in [\![\mathcal{A}]\!]_{\mathfrak{q}} & \iff & \exists (t, \mathfrak{q}').\ (\mathfrak{q}, t, \mathfrak{q}') \in \delta \wedge v_1 \in [\![t]\!] \wedge v_2 \in [\![\mathcal{A}]\!]_{\mathfrak{q}'} \end{array}$$

(Cette définition est bien fondée sur la structure des valeurs.) On note simplement $[\![\mathcal{A}]\!]$ pour $[\![\mathcal{A}]\!]_{\mathfrak{q}_0}$.

Nous notons $\mathbb{1}_{\mathbf{seq}}$ le type $[\mathtt{Any}*]$.

Theorème 10.4 *Soit X un ensemble de valeurs. Les assertions suivantes sont équivalentes :*

(i) Il existe un type $t \leq \mathbb{1}_{\mathbf{seq}}$ tel que $[\![t]\!] = X$.

(ii) Il existe un automate $\mathcal{A}$ tel que $[\![\mathcal{A}]\!] = X$.

(iii) Il existe une expression régulière R, engendrée par les productions ci-dessous, telle que $[\![[R]]\!] = X$.

$$\begin{array}{rcl} R & := & t \\ & | & R_1 R_2 \\ & | & R_1 | R_2 \\ & | & R* \end{array}$$

Preuve: L'implication $(iii) \Rightarrow (i)$ est triviale. L'implication $(ii) \Rightarrow (iii)$ est un résultat classique sur les automates de mots (pour prouver formellement l'implication, il faudrait donner une sémantique directe aux types de la forme $[R]$, et prouver que la traduction préserve cette sémantique). Notons que le langage vide est obtenu avec $R = \mathtt{Empty}$, et le langage réduit à la séquence vide est obtenu avec $R = \mathtt{Empty}*$.

Prouvons l'implication $(i) \Rightarrow (ii)$. Soit $\sqsupset$ un socle qui contient t (et $b_{\text{`}\mathtt{nil}}$). On considère l'automate (Q, δ, t, F), où :

$$\begin{array}{rcl} Q & = & \{t_0 \in \sqsupset \mid t_0 \leq \mathbb{1}_{\mathbf{seq}}\} \\ \delta & = & \{(t_0, t_1, t_2) \in Q \times \sqsupset \times Q \mid (t_1, t_2) \in \pi(t_0 \backslash b_{\text{`}\mathtt{nil}})\} \\ F & = & \{t_0 \in \sqsupset \mid b_{\text{`}\mathtt{nil}} \leq t_0\} \end{array}$$

On montre alors facilement que $[\![\mathcal{A}]\!] = [\![t]\!]$. Notons que si $t_0 \leq \mathbb{1}_{\mathbf{seq}}$ et si $(t_1, t_2) \in \pi(t_0 \backslash b_{\text{`}\mathtt{nil}})$, alors $t_2 \leq \mathbb{1}_{\mathbf{seq}}$. □

10.3.5 Opérateurs

Concaténation Nous introduisons un opérateur de concaténation de séquences, noté **@**. La sémantique de cet opérateur est donnée par :

$$([v_1 \ \ldots \ v_n], [v_{n+1} \ \ldots \ v_m]) \overset{\texttt{@}}{\rightsquigarrow} [v_1 \ \ldots \ v_m]$$

et

$$v \overset{\texttt{@}}{\rightsquigarrow} \Omega$$

lorsque v n'est pas un couple de séquences.

Compte-tenu du Théorème 10.4, nous obtenons un typage exact en prenant **@** : $[R_1]\times[R_2]\rightarrow[R_1\ R_2]$ et des règles d'inférences analogues à celles qui définissent π_1 : $\_\rightarrow\_$ (Figure 5.4). Pour calculer le type résultat étant donné le type d'entrée, on peut choisir des représentants arbitraires pour R_1 et R_2. En pratique, partant de t_1 et t_2, on peut calculer le type résultat pour $t_1\times t_2$ sans construire explicitement deux expressions régulières R_1,R_2 avec $t_i \simeq [R_i]$. L'idée est de construire l'automate associé à t_1, et de produire un système d'équations pour définir le type résultat. Ce système d'équations se déduit facilement de l'automate (on « recolle » t_2 dans les états finaux). Cet algorithme se généralise pour définir la fonction de typage de **@** sur les schémas.

Applatissement Nous introduisons également un opérateur d'applatissement de séquences `flatten`.

La sémantique de cet opérateur est donnée par :

$$[[v_1 \ \ldots \ v_n] \ \ldots \ [v_m \ \ldots \ v_l]] \overset{\texttt{flatten}}{\rightsquigarrow} [v_1 \ \ldots \ v_l]$$

et

$$v \overset{\texttt{flatten}}{\rightsquigarrow} \Omega$$

lorsque v n'est pas une séquence de séquences.

Un typage exact est obtenu à partir de la règle `flatten` : $[R]\rightarrow[\overline{R}]$ où $\overline{R}$ est défini en remplaçant dans R (expression régulière au sens du Théorème 10.4) un type t par R' où R' est choisi de sorte que $t \simeq [R']$.

Remarquons que l'opérateur **@** pourrait être défini à partir de `flatten` en posant $\texttt{@}((v_1, v_2)) = \texttt{flatten}([v_1\ v_2])$.

Map L'opérateur `map` permet d'appliquer une fonction à chaque élément d'une séquence. La sémantique est donnée par :

$$(f, [v_1 \ \ldots \ v_m]) \overset{\texttt{map}}{\rightsquigarrow} [(f\ v_1) \ \ldots \ (f\ v_m)]$$

et

$$v \overset{\texttt{map}}{\rightsquigarrow} \Omega$$

lorsque v n'est pas un couple formé d'une abstraction et d'une séquence.

Le typage est obtenu à partir de la règle `map` : $t_f\times[R]\rightarrow[R']$ où R' est obtenu à partir de R en remplaçant chaque type t par un type s tel que **app** : $t_f\times t\rightarrow s$. Comme pour l'opérateur de concaténation, on peut choisir un représentant R arbitraire sans changer le résultat, et même éviter de le calculer, en déroulant les types récursifs.

Ce typage est exact, à condition d'autoriser d'une manière ou d'une autre une fonction à ne pas toujours retourner la même valeur lorsqu'elle est appliquée au même argument (soit avec un opérateur de choix non déterministe, un générateur aléatoire, des effets de bords, ...). Pour voir pourquoi cela est nécessaire, considérons le type $(b_c \to \mathbb{1}) \times [b_c*]$ où b_c est le type singleton associé à la constante c. Le type calculé pour le résultat de `map` appliqué à une valeur de ce type est $[\mathbb{1}*]$. Or si les fonctions renvoient toujours le même résultat pour le même argument, on ne peut pas atteindre toutes les valeurs de ce type avec un argument de `map` de type $(b_c \to \mathbb{1}) \times [b_c*]$. On voit facilement comment le faire avec un choix non déterministe ou un compteur (incrementé à chaque appel à la fonction), par exemple.

Remarque 10.5 *Le fait de voir les fonctions comme des fonctions non déterministe est naturel dans le cadre d'un langage de programmation, et l'on a d'ailleurs fait cette hypothèse dans la définition d'un modèle (voir les commentaires qui suivent la Définition 4.2).*

Dans un système de types avec polymorphisme paramétrique à la ML, l'opérateur `map` peut en général être défini comme une fonction récursive. La raison pour laquelle ℂDuce possède un opérateur spécifique est non seulement l'absence de polymorphisme paramétrique dans le système de types, mais aussi le manque de précision d'un schéma de type ML pour `map`. En effet, le typage spécifique pour l'opérateur `map` donne par exemple :

$$\texttt{map} : ((t_1 \to s_1) \wedge (t_2 \to s_2)) \times [t_1* \quad t_2?] \to [s_1* \quad s_2?]$$

Autrement dit, lorsque l'on applique une fonction surchargée à tous les éléments d'une séquence de type non uniforme, et qu'elle agit différemment sur les différents types présents dans la séquence, le type du résultat reflète l'ordre (et l'arité) des éléments de la séquence d'entrée, alors qu'un typage à la ML avec un schéma de type polymorphe du genre $\forall\alpha.\forall\beta.(\alpha \to \beta) \times [\alpha*] \to [\beta*]$ imposerait d'approximer le type d'entrée par un type de séquence uniforme ($[(t_1|t_2)*]$ dans l'exemple ci-dessus), et donc de perdre en précision pour le résultat ($[(s_1|s_2)*]$). Le gain de précision obtenu avec le typage exact est apparu utile en pratique pour l'écriture de transformations XML.

Remarque 10.6 *En fait, le langage ℂDuce possède une construction spécifique pour* `map` *:*

$$\texttt{map}\ e\ \texttt{with}\ B$$

où B est un ensemble de branches de filtrage $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$. Cela évite de devoir écrire explicitement une fonction (et un filtrage), en particulier son interface. Pour typer cette construction, après avoir trouvé un type t pour e, on choisit une expression régulière de types R telle que $t \simeq [R]$ et on applique le typage du filtrage de B à chaque type qui apparaît sans R pour produire une nouvelle expression régulière R'.

Ce typage dépend du choix de R, et aucun ne permet en général d'obtenir un type le plus précis (et aucun schéma ne capture tous les types possibles). Par exemple, supposons que e a le type $[0\text{—}\infty]$, et que $B = x \to x + x$. On peut prendre $R = 0\text{—}\infty$, mais aussi des expressions régulières plus fines, telles que $R = (0|1|\ldots|n|((n+1)\text{—}\infty))$ (l'alternative $|$ est le constructeur d'expression régulière). Avec une telle décomposition, la branche $x \to x+x$ sera typée $n+2$ fois,

et le typage exact de l'addition donne un type résultat $0|2|\ldots|2n|((2n+2)\text{—}\infty)$. *On voit que l'on peut obtenir des types arbitrairement précis, mais l'ensemble exact des valeurs que l'on peut obtenir n'est pas représentable. Il faut pouvoir* spécifier *une décomposition en expression régulière qui garantisse les propriétés attendues pour le typage (élimination de la subsomption et admissibilité de la règle intersection). Une manière de le faire est d'utiliser, pour produire l'automate associé à un sous-type de* $\mathbb{1}_{\mathbf{seq}}$*, une fonction* π *qui associe à un type* $t \leq \mathbb{1}_{\mathbf{prod}}$ *l'ensemble des types* (t_1, t_2) *tels que le produit cartésien* $t_1 \times t_2$ *est maximal parmi les sous-types de* t *(c'est-à-dire que si* $t_1 \times t_2 \leq t'_1 \times t'_2 \leq t$*, alors* $t_1 \simeq t'_1$ *et* $t_2 \simeq t'_2$*), en choisissant un représentant pour chaque produit maximal. Cette construction a été mentionnée à la suite du Théorème 4.36.*

Le langage possède d'autres itérateurs que nous n'allons par présenter ici.

10.3.6 Chaînes de caractères

ℂDuce n'a pas de type de base spécifique pour représenter les chaînes de caractères. Celles-ci sont simplement des séquences de caractères. Le type `String` est défini comme étant [`Char`$*$]. On définit aussi `PCDATA` comme étant l'expression régulière `Char`$*$.

La chaîne de caractères [$'$`a`$'$ $'$`b`$'$ $'$`c`$'$] peut être écrite [$'$`abc`$'$], ou encore "`abc`". Une chaîne littérale peut aussi être vue comme un type singleton.

Le choix de voir les chaînes de caractères comme des séquences permet de réutiliser les opérateurs sur les séquences (concaténation, itération), ainsi que les types et motifs expression régulières pour spécifier finement des sous-types de `String` et pour extraire des informations des chaînes par filtrage.

10.4 XML

ℂDuce permet de représenter les documents XML comme des valeurs du langage. Un élément XML a la forme :

```
<tag attr1="..."  ... attrn="...">
...
</tag>
```

Le contenu, entre la balise $<$ `tag`$\ldots>$ et la balise $<$ /`tag` $>$ est une séquence qui mélange caractères et éléments XML. Un tel élément est codé en ℂDuce par une valeur de la forme :

```
<(`tag) {attr1="...";  ...; attrn="..."}>[ ... ]
```

ou plus simplement, avec les conventions déjà mentionnées :

```
<tag attr1="..."  ... attrn="..."}>[ ... ]
```

Par exemple, le document XML

```
<animal kind="insect">
 <name>Bumble bee</name>
 <comment>There are about <strong>19</strong> different species
          of bumblebee.</comment>
</animal>
```

est codé, en supprimant certains caractères espace, par :

```
<animal kind="insect">[
 <name>['Bumble bee']
 <comment>['There are about ' <strong>['19'] ' different species of bumblebee.']
]
```

Le type de tous les documents XML peut être défini par :

```
type XML = <(Atom) { _ = String }>[ (Char | XML)* ]
```

On peut écrire en ℂDuce des sous-types qui capturent un fragment important des contraintes imposées sur les documents par une spécification comme DTD ou XML-Schema, par exemple :

```
type Animals = <animals {||}>[ Animal* ]
type Animal = <animal {|king=String|}>[ <name>String <comment>Text ]
type Text = [ TextItem* ]
type TextItem = Char | <strong>Text
```

(Les accolades $\{|\ldots|\}$ interdisent la présence d'autres attributs que ceux mentionnés.)

Un utilitaire `dtd2cduce` permet de convertir automatiquement une DTD en un tel ensemble de déclarations de type. On peut également importer des spécifications XML-Schema ; le type des valeurs produites par la validation contre une telle spécification n'est pas forcément un sous-type de `XML`. Par exemple, si la spécification indique qu'un certain attribut est un entier (type `xs :int`), la valeur du champ correspondant, dans le document obtenu après validation, sera un entier ℂDuce (type `Int`) et non une chaîne de caractère (type `String`), et cela est reflété dans le type statique associé au XML-Schema.

Des fonctions prédéfinies (en fait, des opérateurs), de type `String→XML` et `XML→String` permettent de lire un document XML contenu dans une chaîne de caractères, et réciproquement. En cas d'erreur de *parsing*, si une chaîne ne représente pas un document XML bien formé, une exception (voir plus bas) est levée pour signaler l'erreur. En fait, pour améliorer les messages d'erreurs, les documents XML peuvent être chargés directement à partir d'un fichier (ou d'une URL) externe.

10.5 Constructions dérivées

Liaison locale La construction `let` $p = e_1$ `in` e_2 est du sucre syntaxique pour :

$$\texttt{match}\ e_1\ \texttt{with}\ p \to e_2$$

Contrainte de type L'opérateur de contrainte de type $(\_ : t)$, pour un type t, peut être défini comme du sucre syntaxique pour l'application de la fonction

$$\texttt{fun}\ (t{\rightarrow}t)\ x \to x$$

Si l'on préfère le voir comme un opérateur o_t, la relation de typage $o_t : \_{\rightarrow}\_$ est donnée par $o_t : t_1{\rightarrow}t_2 \iff t_1 \leq t \leq t_2$. La sémantique est l'identité. Le

typage n'est évidemment pas exact, et c'est le but recherché : cet opérateur permet d'oublier certaines informations de type, ce qui peut être utile pour documenter du code, pour mettre au point les programmes (en améliorant la localisation des messages d'erreurs), ou pour simplifier la tâche du typeur lorsque les types manipulés deviennent trop complexes et inutilement précis.

Abstractions curryfiées La syntaxe ci-dessous facilite l'écriture de fonctions curryfiées :

$$\texttt{fun}\ f(x_1 : t_1) \ldots (x_n : t_n) : s = e$$

L'identificateur f est optionnel. Pour $n = 1$, cette expression est traduite en :

$$\texttt{fun}\ f(t_1 \rightarrow s)\ x_1 \rightarrow s$$

Pour $n \geq 2$, la traduction est donnée par :

$$\texttt{fun}\ f(t_1 \rightarrow (\ldots (t_n \rightarrow s) \ldots))\ x_1 \rightarrow (\texttt{fun}\ (x_2 : t_2) \ldots (x_n : t_n) : s = e)$$

Par exemple, la traduction complète de $\texttt{fun}\ f(x_1 : t_1)(x_2 : t_2) : s = e$ est :

$$\texttt{fun}\ f(t_1 \rightarrow (t_2 \rightarrow s))\ x_1 \rightarrow (\texttt{fun}\ (t_2 \rightarrow s)\ x_2 \rightarrow e)$$

Booléens Les deux constantes atome `'true` et `'false` sont utilisées pour représenter les booléens. Le type `Bool` est prédéfini comme `'true | 'false`. La construction `if` e `then` e_1 `else` e_2 est traduite en :

$$\texttt{match}\ e\ \texttt{with}\ \texttt{'true} \rightarrow e_1 \mid \texttt{'false} \rightarrow e_2$$

Les connecteurs logiques sont définis à partir de cette construction, ce qui leur donne une sémantique paresseuse par rapport à leur deuxième argument (pour les connecteurs binaires). Par exemple, la conjonction logique est définie par :

$$e_1 \&\& e_2 \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{'false}$$

10.6 Traits impératifs

Le langage ℂDuce étend le noyau purement fonctionnel du Chapitre 5 avec certains traits impératifs issus de ML. La manière d'étendre la sémantique opérationnelle pour rendre compte de ces constructions est classique, et nous n'allons pas la décrire.

La valeur `[]` (c'est-à-dire `'nil`) et le type singleton associé sont utilisés comme le type `unit` en ML, pour représenter l'argument ou le résultat des fonctions qui opèrent par effet de bord.

L'opérateur de séquencage $e_1; e_2$ évalue d'abord e_1 (dont le type doit être `[]`) puis e_2 et renvoie le résultat de e_2. La sémantique à petits pas, définie à la Section 5.5 ne spécifie pas complètement l'ordre d'évaluation du langage ; par exemple, l'implémentation peut évaluer dans un ordre arbitraire les deux sous-expressions e_1 et e_2 dans une expression (e_1, e_2). Pour une expression enregistrement $\{l_1 = e_1; \ldots; l_n = e_n; \_ = e_0\}$, l'expression e_0 peut être évaluée un nombre arbitraire (fini) de fois.

Nous n'allons pas décrire les opérations classiques d'entrée-sortie, d'accès au système d'exploitation, ...

Références Les références sont des cellules mémoire, typées, dont on peut extraire et modifier le contenu. Pour éviter d'introduire une nouvelle catégorie de constructeurs, nous définissons le type des références de type t, noté `ref` t, comme étant :

$$\{|\ \texttt{get} = [] \to t;\ \texttt{set} = t \to []\ |\}$$

Autrement dit, une référence est vue comme un enregistrement à deux champs (un objet à deux méthodes, dans la terminologie des langages orientés objets), qui correspondent aux deux opérations que l'on peut effectuer sur les références (lecture du contenu et affectation).

On introduit un opérateur $\texttt{ref}_t$ qui prend une valeur v de type t, crée une cellule mémoire initialisée avec la valeur v, et renvoie un enregistrement de type `ref` t dont les méthodes sont des fonctions « magiques » (non définissables dans le langage) qui accèdent effectivement à la cellule mémoire.

L'affectation $e_1 := e_2$ est du sucre syntaxique pour e_1`.set` e_2. L'accès $!e$ est du sucre syntaxique pour e`.get[]`.

Il faut noter que les deux fonctions `get` et `set` d'une valeur de type `ref` t ne sont pas forcément issues (directement) d'un appel au constructeur de référence $\texttt{ref}_t$. Il est possible, par exemple, de créer des valeurs de type `ref` t qui instrumentent une vraie référence, en imprimant une trace des accès effectués. On peut aussi définir des références non initialisées, de la manière suivante (par exemple, pour le type `Int`) :

```
type Int_option = Int | `none
type Int_ref = ref Int_option
let new_ref(_ : []) : Int_ref =
 let r = ref Int_option `none in
 { get =
     (fun (_ : []) : Int = match r.get [] with
           | `none -> raise "Empty reference !"
           | x -> x);
   set = r.set
 }
```

Si l'on essaie d'accéder au contenu d'une référence non initialisée, une exception est levée (voir paragraphe suivant). Notons que dans la définition de la méthode `set`, on utilise la subsomption autorisée par le sous-typage :

$$\texttt{Int\_option}\to[] \leq \texttt{Int}\to[]$$

Exceptions La sémantique des exceptions de ℂDuce est similaire à celle de ML. Le contenu d'une exception peut être n'importe quelle valeur. On lève une exception v par la construction `raise` v (dont le type est $\mathbb{0}$) et on capture une exception par un filtrage :

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1\ |\ \ldots\ |\ p_n \to e_n$$

Les branches du filtrage sont typées avec un type d'entrée $\mathbb{1}$ (car e peut lever *a priori* n'importe quelle exception). Une branche implicite $x \to$ `raise` x ajoutée à la fin du filtrage lève de nouveau l'exception si aucune branche ne la capture.

10.7 Un exemple

La Figure 10.1 donne un exemple de programme ℂDuce, qui définit une unique fonction de transformation XML, appelée `split`.

```
type Person   = FPerson | MPerson
type FPerson  = <person gender = "F">[ Name Children ]
type MPerson  = <person gender = "M">[ Name Children ]
type Children = <children>[ Person* ]
type Name     = <name>[ PCDATA ]

type Man       = <man name=String>[ Sons Daughters ]
type Woman     = <woman name=String>[ Sons Daughters ]
type Sons      = <sons>[ Man* ]
type Daughters = <daughters>[ Woman* ]

let fun split (MPerson -> Man ; FPerson -> Woman)
 <person gender=g>[
   <name>n
   <children>[(mc::MPerson | fc::FPerson)*]
 ] ->
     let tag = match g with "F" -> `woman | "M" -> `man in
     let s = map (split,mc) in
     let d = map (split,fc) in
     <(tag) name=n>[ <sons>s  <daughters>d ]
```

FIG. 10.1 – Un programme ℂDuce

Le programme commence par des déclarations de types. Le type `Person` représente l'arbre de descendance généalogique d'une personne. Le sexe d'une personne est donné par la valeur d'un attribut. La fonction `split` transforme une valeur de ce type en un arbre dans lequel, à chaque niveau, les fils et les filles sont regroupés. Le sexe des personnes est indiqué par une étiquette d'élément. De plus, le nom des personnes, qui était contenu dans un élément XML, se retrouve dans un attribut.

La fonction est constituée d'un filtrage avec une unique branche. Le motif permet d'extraire, d'un seul coup, le nom de la personne (dans la variable `n`), son sexe (variable `g`), et de séparer la liste de ses enfants en deux : la variable `mc` capture tous les fils, et la variable `fc` capture toutes les filles. Dans le corps de la branche, on commence par calculer l'étiquette qui correspond au sexe de la personne. On applique ensuite récursivement `split` à tous les fils et à toutes les filles, au moyen de l'itérateur `map`. Enfin, on construit l'élément XML à renvoyer.

Pouvoir expressif du filtrage L'exemple montre le pouvoir expressif du filtrage, qui permet d'effectuer une extraction d'information assez complexe avec un seul motif. On pourrait même aller plus loin, et calculer l'étiquette `tag` directement dans le motif, en utilisant des motifs constante :

```
<person gender=("F" & (tag := ‘woman)) | ("M" & (tag := ‘man))>[
  <name>n
  <children>[(mc::MPerson | fc::FPerson)*]
]
```

ou alors, en mettant à part ce calcul :

```
<person>[
  <name>n
  <children>[(mc::MPerson | fc::FPerson)*]
]
& ((FPerson & (tag := ‘woman)) | (MPerson & (tag := ‘man)))
```

Typage L'interface de la fonction indique deux types flèche, ce qui donne à la fonction un type plus précis que `Person -> Man | Woman`. Cette plus grande précision est nécessaire pour pouvoir typer correctement le corps de la fonction. En effet, lorsque l'opérateur `map` applique la fonction récursivement à tous les éléments de `mc`, qui sont tous de type `MPerson`, il renvoie une séquence d'éléments de type `Man`. De même, partant de `fc`, on obtient des éléments de type `Woman`. Cela permet de voir que le contenu des éléments `<sons>` et `<daughter>` sont bien, respectivement, de type `[ Man* ]` et `[ Woman* ]`, ce qui est nécessaire pour obtenir une valeur de type acceptable en sortie. Si l'on ne disposait que du type `Person -> Man | Woman` pour la fonction `split`, le typeur donnerait le type `[ (Man | Woman)* ]` pour les variables `s` et `d`, ce qui ferait échouer le typage.

Le corps de la fonction est typé deux fois : une fois sous l'hypothèse que l'argument est de type `MPerson`, une fois sous l'hypothèse qu'il est de type `FPerson`. Le typeur doit vérifier que l'étiquette de l'élément renvoyé est bien `‘man` ou `‘woman` selon le cas. Cela est fait de la manière suivante. Tout d'abord, le typage exact du filtrage donne un type singleton pour la variable `g`. Ensuite, dans le filtrage utilisé pour calculer `tag`, seule une des deux branches contribue au type du résultat (la règle de typage du filtrage permet d'ignorer la branche qui n'est pas utilisée). Cela permet de retrouver un type singleton pour `tag`, et donc de vérifier que l'étiquette de l'élément renvoyé est bien celle attendue.

Compilation du filtrage Le motif utilise un style déclaratif : il indique comment séparer la liste des enfants en deux, en utilisant les types `MPerson` et `FPerson`. Le compilateur se rend compte que les deux types peuvent en fait être distingués simplement en regardant la valeur de l'attribut `gender`. Si l'attribut vaut `"F"`, l'élément est de type `FPerson`, sinon il est de type `MPerson` (car il est forcément de l'un des deux types, étant donné le type des arguments de la fonction). Le code produit procède effectivement ainsi pour distinguer entre les deux types. Un schéma de compilation naïf devrait parcourir tout l'arbre XML pour vérifier si l'élément est de type `FPerson` ou s'il est de type `MPerson`.

De plus, les tests sur les étiquettes des éléments XML `person`, `name`, `children` ne sont là que pour simplifier la relecture du code. Le compilateur sait statiquement que ces tests sont vérifiés, et il ne génère donc aucun code pour eux.

On voit sur cet exemple que la compilation efficace du typage, qui utilise les types statiques, permet au programmeur d'utiliser un style plus déclaratif (utilisation « abstraite » des types `MPerson`, `FPerson`, au lieu d'une méthode

concrète pour les distinguer), et plus informatif (tests d'étiquettes redondants), sans nuire à l'efficacité du code produit.

Chapitre 11

Techniques d'implémentation

Dans ce chapitre, nous décrivons de manière informelle certaines techniques mises en œuvre dans l'implémentation de ℂDuce.

11.1 Typeur

Restriction de la règle (*abstr*) Nous avons introduit (Section 5.7) une notion de schémas pour représenter symboliquement l'ensemble des types d'une expression. Une solution alternative pour obtenir un algorithme de typage consiste à restreindre la règle de typage (*abstr*) en interdisant les types flèche négatifs (c'est-à-dire en imposant $m = 0$). Dans le système de types obtenu, on peut calculer facilement le meilleur type pour une expression e pour un environnement de typage Γ donné (sous certaines hypothèses sur le typage des opérateurs). Évidemment, ce système permet de typer moins de programmes que l'algorithme à base de schémas. La situation caractéristique est celle où l'on filtre une abstraction avec un type flèche négatif :

$$\texttt{match}\ (\texttt{fun}\ (\texttt{Int}\rightarrow\texttt{Int})\ldots)\ \ \texttt{with}\ \neg(\texttt{Char}\rightarrow\texttt{Char}) \rightarrow 0$$

Pour voir que cette expression est bien typée, il est nécessaire d'autoriser les types flèche négatifs (ce qui permet de donner le type $\neg(\texttt{Char}\rightarrow\texttt{Char})$ à l'abstraction). Nous n'avons jamais rencontré en pratique une telle situation, et nous avons donc choisi d'implémenter le système sans types flèche négatifs. Évidemment, il ne s'agit que du système de types statique : la sémantique dynamique, qui peut effectuer des tests de type dynamique sur les *valeurs*, n'est pas affectée. En particulier, l'expression suivante, qui est bien typée dans le système implémenté, s'évalue en 0 comme attendu :

$$\texttt{match}\ (\texttt{fun}\ (\texttt{Int}\rightarrow\texttt{Int})\ldots)\ \ \texttt{with}\ \neg(\texttt{Char}\rightarrow\texttt{Char}) \rightarrow 0 \mid \_ \rightarrow 1$$

Autrement dit, l'algorithme de typage sans types flèche négatifs est correct par rapport au système de types (et donc il garantit l'absence d'erreur de type à l'exécution), mais il n'est pas complet.

Localisation des erreurs L'implémentation naïve du typeur sous la forme d'un algorithme ascendant (qui calcule le type d'une expression des feuilles vers la racine) ne permet pas de localiser précisément les erreurs de type. Considérons par exemple l'expression ci-dessous :

$$\texttt{fun}\ (t{\rightarrow}s)\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$$

Un algorithme de typage ascendant va commencer par calculer un type s_i pour chaque branche e_i, puis le type pour le filtrage (la réunion des types obtenus pour chaque branche), avant de vérifier que ce type est bien sous-type de s. Si cela n'est pas le cas, la règle (*abstr*) ne peut pas s'appliquer, et une erreur est alors signalée au niveau de l'abstraction. Or l'erreur se situe en fait dans l'une (au moins) des branches, à savoir celle(s) dont le type du résultat n'est pas sous-type de s. Si l'expression e_i est incriminée, on peut éventuellement identifier encore plus précisément une sous-expression qui engendre l'erreur de typage. Par exemple si $s = \texttt{Int}{\times}\texttt{Char}$ et $e_i = (1, 2)$, on voit que l'erreur provient de la constante 2, et non pas de l'expression e_i globalement.

Pour pouvoir localiser précisément les erreurs, nous utilisons un algorithme qui synthétise les types de manière ascendante, mais propage également une contrainte de manière descendante. On peut formaliser cet algorithme sous la forme d'un jugement $\Gamma \vdash e : t \leq t_0$, où Γ, e, t_0 sont des entrées, et t est la sortie de l'algorithme (le type le plus précis pour e, modulo la restriction sur la règle (*abstr*)). Si ce jugement ne peut être dérivé, l'algorithme signale la sous-expression fautive, et sinon, il renvoie un type t qui est nécessairement plus précis que t_0. Décrivons les principaux cas de l'algorithme, suivant la forme de e.

Si $e = (e_1, e_2)$, alors on commence par vérifier que $t'_0 = t_0 \wedge \mathbb{1}_{\mathbf{prod}}$ n'est pas vide. S'il est vide, ce n'est pas forcément une erreur de typage, mais il faut alors vérifier que l'une des deux expressions e_1 ou e_2 a le type $\mathbb{0}$ et que l'autre est bien typée. Si les deux expressions sont bien typées mais qu'aucune n'a le type $\mathbb{0}$, on signale une erreur pour e, qui est une expression couple, alors que le type attendu ne contient aucun couple. Si t'_0 n'est pas vide, on calcule t_1, le type le plus précis pour e_1, avec la contrainte $\pi_1[t'_0]$. Pour e_2, on utilise comme contrainte le plus gros type t_2 tel que $t_1{\times}t_2 \leq t'_0$ (un calcul facile montre que l'on peut prendre $t_2 = \neg\pi_2[(t_1{\times}\mathbb{1})\backslash t_0]$). Le cas des enregistrements se traite d'une manière similaire.

Si e est une abstraction $\mu f(t_1{\rightarrow}s_1; \ldots t_n{\rightarrow}s_n).\lambda x.e$, on commence par vérifier que l'intersection des types flèche de l'interface est bien sous-type de t_0. On calcule ensuite, pour chacun de ces types $t_{i_0}{\rightarrow}s_{i_0}$, le type du corps e de l'abstraction, avec la contrainte s_{i_0} et avec l'environnement étendu par les hypothèses $(x : t_{i_0})$ et $(f : \bigwedge t_i{\rightarrow}s_i)$. Le type du résultat peut être ignoré.

Pour le filtrage, on calcule le type de l'expression filtrée, sous la contrainte t'_0 qui est la réunion $\bigvee \lfloor p_i \rfloor$ des types acceptés par les motifs des différentes branches. Une approche alternative est de calculer ce type sans contrainte et de vérifier après coup que le type obtenu est sous-type de cette réunion, ce qui peut éventuellement donner un message d'erreur plus explicite (« filtrage non exhaustif »). Dans les deux cas, on calcule ensuite un type pour chaque branche (sauf celles qui ne peuvent pas réussir), avec la même contrainte t_0 que pour le filtrage.

Pour chaque opérateur, on adopte une stratégie spécifique pour calculer la contrainte à utiliser pour typer l'argument. On peut éventuellement choi-

sir de ne pas utiliser les contraintes les plus précises, pour ne pas introduire des contraintes trop complexes (qui donneraient des messages d'erreur peu clairs). Par exemple, pour typer la concaténation $e_1@e_2$ avec une contrainte t_0, on peut commencer par calculer le plus petit type t'_0 tel que $t_0 \leq [t'_0*]$ et typer les deux arguments avec la contrainte $[t'_0*]$. Pour typer une application e_1e_2, on commence par calculer un type t_1 pour e_1 sous la contrainte $\mathbb{0}\to\mathbb{1}$; on utilise alors le type $\mathrm{Dom}(t_1)$ pour typer e_2 (en fait, on pourrait faire mieux et calculer le plus gros type t_2 tel que $\mathbf{app} : t_1\times t_2 \leq t_0$).

Messages d'erreur Lorsqu'une erreur de type a été détectée, il s'agit en général d'une contrainte $t \leq t_0$ qui n'est pas vérifiée, où t est le type calculé pour une sous-expression e et t_0 est la contrainte qui a été utilisée pour typer e. Dans le rapport d'erreur, on signale que le meilleur type que l'on peut calculer pour l'expression est t, alors que le type attendu est t_0, et pour expliquer pourquoi t n'est pas sous-type de t_0, on peut exhiber une valeur de type $t\setminus t_0$ (ce type n'étant pas vide, il existe une telle valeur, et on peut en fait en calculer une explicitement).

L'algorithme de typage permet de détecter les expressions mal typées. Il permet également de détecter d'autres types d'erreurs potentielles, qui peuvent donner lieu à des avertissements. Voici quelques exemples.

- Déclaration de type vide. Considérons le programme ci-dessous :
  ```
  type T = <a>[ T+ ]
  let fun f(x : T) : Int = ...
  ```
  Ce programme est bien typé, indépendamment du corps de la fonction. En effet, le type `T` est vide (car les valeurs sont des objets finis, et `T` dénoterait plutôt des arbres nécessairement infinis), et la règle de typage du filtrage indique qu'il n'est pas nécessaire de typer les branches qui ne peuvent pas être utilisées (ce qui est le cas, pour une fonction, lorsque le type de l'argument est vide). On peut supposer que le programmeur s'est trompé en définissant le type `T`, et qu'il voulait plutôt écrire :
  ```
  type T = <a>[ T* ]
  ```
  Pour détecter ce genre d'erreurs, le typeur signale, sous la forme d'un avertissement, les déclarations de type qui définissent un type vide.
- Branche ou motif inutilisé. Dans un filtrage, une branche peut ne pas être utilisée (lorsque le type d'entrée du filtrage est disjoint du type accepté par le motif), et elle n'est alors pas typée. Cela est utile pour typer une fonction surchargée, où certaines branches doivent être ignorées pour vérifier certaines des contraintes données dans l'interface (rappelons que le corps d'une fonction surchargée est typée une fois pour chaque type flèche de l'interface). Néanmoins, une branche qui ne serait jamais typée correspond probablement à une erreur de programmation. Cette situation est détectée et signalée au programmeur. On peut raffiner cette analyse, et détecter les sous-motifs inutilisés ; pour cela, on peut instrumenter l'algorithme de typage des motifs, pour détecter ceux qui reçoivent systématiquement un type vide en entrée.
- Variable de capture inutilisée. Les variables et les noms de types ne sont pas distingables syntaxiquement. Une ambiguïté apparaît donc au moment d'interpréter un identificateur x dans un motif : il peut s'agir d'une variable de capture ou d'un motif contrainte de type, si x est déclaré plus haut dans

le programme (ou prédéfini). L'heuristique suivante est utilisée pour lever l'ambiguïté : si le type x est défini, l'identificateur est interprété comme un motif contrainte de type ; sinon, il est interprété comme une variable de capture. Cette heuristique peut cacher des erreurs de frappe ; considérons par exemple le programme ci-dessous :

```
fun ( (Int | Char, Int | Char) -> Int)
| (Inte,Int) -> 0
| _ -> 1
```

Dans la première branche, l'identificateur `Int` est interprété comme un type, et l'identificateur `Inte` comme une variable de capture. La fonction est bien typée (aucun avertissement n'est donnée pour la seconde branche, car elle n'est pas inutile). Cependant, on peut supposer que l'identificateur `Inte` est une erreur de frappe, et que le programmeur voulait taper `Int`. Pour détecter ce genre d'erreur, il suffit de donner un avertissement lorsqu'une variable de capture n'apparaît pas dans le corps de la branche (c'est le cas pour la variable `Inte` ci-dessus).

11.2 Représentation des valeurs

Pour des raisons de performances, dans l'implémentation, l'algèbre des valeurs est étendue avec un certain nombre de formes dérivées.

Chaînes de caractères En ℂDuce, les chaînes de caractères sont conceptuellement des séquences. Ainsi, la chaîne `"abc"` est en fait la valeur $[\texttt{'a}\ \texttt{'b}\ \texttt{'c}]$, c'est-à-dire $(\texttt{'a}, (\texttt{'b}, (\texttt{'c}, \texttt{`nil}())))$. Un tel codage est très coûteux en place mémoire. L'implémentation manipule à l'exécution des valeurs de la forme $\texttt{string}(s, i, j, v)$ où s est un tableau de caractères, i et j sont deux index dans s, et v est une valeur séquence. Cette valeur représente en fait la séquence constituée des éléments de s compris entre les index i et j, suivis de la séquence v. On a donc l'équivalence :

$$\texttt{string}(s, i, j, v) \equiv \begin{cases} (s[i], \texttt{string}(s, i+1, j, v)) & \text{si } i < j \\ v & \text{si } i = j \end{cases}$$

Cette forme dérivée permet de représenter de manière compacte en mémoire des chaînes de caractères, ou plus généralement des sous-séquences consécutives formées de caractères. Notons que le tableau s n'est pas recopié dans l'équivalence ci-dessus : dérouler la représentation compacte pour en extraire les caractères un par un se fait en temps constant (il suffit d'incrémenter un entier). Cette représentation permet également de « sauter » par dessus une suite de caractères dans une séquence. Ainsi, si l'on applique le motif $[((x :: t)|\_)*]$ (qui capture tous les éléments de type t d'une séquence) avec $t \wedge \texttt{Char} \simeq \mathbb{0}$, et que l'on tombe sur $\texttt{string}(s, i, j, v)$, on peut directement sauter à v, sans considérer tous les caractères de s entre i et j. De même, le résultat pour une variable de capture de séquence, lorsque la valeur filtrée est de la forme $\texttt{string}(s, i, j, v)$, peut être représentée en utilisant des représentations compactes de la forme $\texttt{string}(s, i', j', v')$. On peut utiliser l'équivalence suivante pour regrouper des fragments consécutifs d'un même tableau s :

$$\texttt{string}(s, i, j, \texttt{string}(s, j, k, v)) \equiv \texttt{string}(s, i, k, v)$$

Concaténation paresseuse L'opérateur de concaténation @, implémenté naïvement, a un temps d'exécution linéaire en la longueur de son premier argument. En effet, la concaténation de la séquence $[v_1 \dots v_n]$ et d'une séquence v' est $(v_1, (v_2, \dots, (v_n, v') \dots))$, et il faut donc construire n couples imbriqués. Un programme qui construit une séquence en concaténant les éléments un par un à la fin a ainsi une complexité quadratique (seulement pour les concaténations) en la longueur du résultat.

Nous introduisons une nouvelle forme de valeur manipulée à l'exécution, notée $v_1 @ v_2$, qui représente symboliquement la concaténation de v_1 et de v_2 (deux séquences). Cette valeur est construite en temps constant à partir de v_1 et de v_2. Pour itérer sur une valeur $v_1 @ v_2$, il suffit d'itérer d'abord sur v_1, puis sur v_2. Par contre, lorsque l'on évalue un filtrage, il est parfois nécessaire de normaliser $v_1 @ v_2$; pour cela, on utilise les équivalences :

$$\begin{array}{lcl} \texttt{`nil} @ v & \equiv & v \\ (v_1, v_2) @ v & \equiv & (v_1, v_2 @ v) \end{array}$$

Ces équivalences permettent d'exposer le premier élément d'une séquence, c'est-à-dire de la mettre sous la forme (v_1, v_2). Il s'agit d'une normalisation incrémentale. Pour normaliser $v' @ v$, lorsque v' est elle même une concaténation symbolique, on commence par normaliser v' avant d'appliquer les règles ci-dessus.

En fait, on peut voir toute valeur comme un arbre binaire dont les nœuds sont des @, et les feuilles des valeurs qui ne sont pas de la forme $v_1 @ v_2$. On peut normaliser un tel arbre en temps linéaire par rapport à la taille du résultat. Il s'agit alors d'une normalisation globale, qui évite de construire des structures intermédiaires inutiles.

Pour éviter d'évaluer plusieurs fois les mêmes concaténations, on peut modifier en place une valeur $v_1 @ v_2$ par sa forme normalisée lorsqu'on la calcule (soit de manière incrémentale, soit globalement).

Représentation des atomes Les atomes sont des couples de symboles (constitués d'un *namespace* XML et d'un nom local). Pour diminuer la taille mémoire occupée par les documents en mémoire, et pour accélérer les opérations sur les atomes (égalité et comparaison pour implémenter des arbres de décision lors du filtrage), ceux-ci sont représentés en interne avec partage optimal (obtenu par *hash-consing*) et identifiant numérique unique.

Comparaison avec d'autres travaux Le projet Xtatic a proposé [GLPS04] indépendamment le même genre de techniques pour évaluer de manière paresseuse la concaténation de séquences, et pour représenter de manière compacte des séquences de caractères.

Une approche complètement différente pour représenter les valeurs est proposée dans le cadre du projet XHaskell [LS04a, LS04b]. XDuce, ℂDuce et Xtatic utilisent une représentation uniforme pour l'algèbre de valeurs ; cela signifie qu'une valeur est représentée indépendamment de son type statique. La règle de subsomption est alors transparente à l'exécution : on peut effectivement voir une valeur de type t comme une valeur de type s lorsque $t \leq s$, sans lui appliquer aucune transformation. XHaskell propose d'associer à chaque type expression régulière une représentation concrète des valeurs de ce type. Ainsi, à chaque

type expression régulière est associé un type Haskell, de manière compositionnelle. Par exemple, l'étoile de Kleene des expressions régulière se traduit en le constructeur de liste de Haskell, et l'opérateur de choix | se traduit en un type somme à deux constructeurs. La représentation concrète du résultat d'une expression du langage dépend alors de son type statique. La règle de subsomption nécessite donc de changer la représentation des valeurs, et pour cela, il faut instrumenter l'algorithme de sous-typage pour pouvoir extraire de la preuve d'une instance $t \leq s$ une fonction de coercion entre la représentation de t et celle de s (en XHaskell, cela est fait en encodant l'algorithme de sous-typage dans un système étendu de classes de types Haskell).

Le langage Xen/Cω [MS03] est une extension de C$^\#$ avec des types structurels qui permettent de représenter des documents XML. Là encore, à chaque type XML du langage est associé un type dans l'architecture d'implémentation (la machine virtuelle CLR de Microsoft .NET), et il faut introduire des coercions pour rendre compte du sous-typage. Contrairement à XHaskell, cependant, Xen n'essaie pas d'implémenter un sous-typage d'inspiration ensembliste entre types expression régulière.

Le choix entre une représentation uniforme comme en XDuce, $\mathbb{C}$Duce, Xtatic et une représentation spécialisé par les types, comme en XHaskell est délicat. Outre sa simplicité, la représentation uniforme permet d'implémenter « gratuitement » la subsomption entre sous-types alors que la représentation spécialisés peut nécessiter une recopie complète des valeurs. La représentation spécialisée permet par contre des accès plus rapides aux données à l'intérieur de structures complexes (adressage direct) ; elle est aussi plus compacte, car une partie de la structure des valeurs est stockée implicitement dans leur type statique, et permet une meilleure intégration avec des langages généralistes.

11.3 Représentation des combinaisons booléennes

De nombreux algorithmes présentés dans cette thèse travaillent dans un socle $\sqsupseteq$ donné. La complexité de ces algorithmes, même si nous ne l'avons pas étudiée formellement, dépend de manière cruciale de la taille du socle, c'est-à-dire du nombre de types différents que l'on obtient en partant des types qui sont les données du problème, et en saturant par combinaison booléenne et par décomposition des atomes qui interviennent dans les types. Pour garantir que ce nombre est toujours fini, nous avons introduit à la Section 3.1 une certaine représentation des combinaisons booléennes (forme normale disjonctive).

Dans cette section, nous proposons des techniques alternatives pour optimiser cette représentation, en vue de diminuer le nombre de types différents considérés dans les algorithmes, et donc leur complexité.

11.3.1 Simplification par tautologies booléennes

On peut optimiser la représentation des combinaisons booléennes en prenant en compte un certain nombre de tautologies booléennes. Par exemple, la représentation par des formes normales disjonctive tient déjà compte de la commutativité et de l'associativité de la réunion et de l'intersection. On peut aller plus loin et imposer des contraintes de normalisation plus fortes. Formellement, on considère une relation de réécriture $\rightsquigarrow$ sur les $\mathcal{B}X$ telle que que pour toute

interprétation ensembliste de $\mathcal{B}X$ dans un ensemble D, si $t \rightsquigarrow t'$, alors $[\![t]\!] = [\![t']\!]$ (l'interprétation ne change pas lorsque l'on simplifie la combinaison booléenne). Les simplifications considérées diminuent la taille des combinaisons booléennes, ce qui assure leur terminaison. On peut normaliser toutes les combinaisons booléennes manipulées, ce qui réduit leur taille, et aussi leur nombre, en permettant d'identifier plus de combinaisons. Nous donnons quelques exemples de simplifications possibles.

Dans une forme normale disjonctive, toute « ligne » qui contient un même atome sous forme positive et sous forme négative en même temps peut être supprimée. Cela se traduit par la règle de simplification :

$$\frac{P \cap N \neq \emptyset}{\{(P,N)\} \cup t \rightsquigarrow t}$$

Une autre simplification consiste à remarquer que si l'on peut passer d'une ligne à l'autre dans une forme normale disjonctive en ajoutant des littéraux, alors la ligne obtenue est inutile (car son interprétation ensembliste est incluse dans la ligne de départ) :

$$\frac{P' \subseteq P \quad N' \subseteq N}{\{(P,N),(P',N')\} \cup t \rightsquigarrow \{(P',N')\} \cup t}$$

On peut fusionner deux lignes égales sauf pour un atome qui apparaît sous forme positive dans l'une et négative dans l'autre. Cela correspond à la tautologie : $(x \wedge A) \vee (\neg x \wedge A) \simeq A$. On pose :

$$\frac{}{\{(P \cup \{x\}, N),(P, N \cup \{x\})\} \cup t \rightsquigarrow \{(P,N)\} \cup t}$$

Un dernier exemple de simplification consiste à supprimer le littéral $\neg x$ dans $(x \wedge B) \vee (\neg x \wedge A \wedge B)$:

$$\frac{P' \subseteq P \quad N' \subseteq N}{\{(P, N \cup \{x\}),(P' \cup \{x\}, N')\} \cup t \rightsquigarrow \{(P,N),(P' \cup \{x\}, N')\} \cup t}$$

et symétriquement :

$$\frac{P \subseteq P' \quad N \subseteq N'}{\{(P, N \cup \{x\}),(P' \cup \{x\}, N')\} \cup t \rightsquigarrow \{(P, N \cup \{x\}),(P', N')\} \cup t}$$

Plus on détecte de simplifications possibles, plus on diminue la taille et le nombre des combinaisons booléennes manipulées. Les algorithmes qui manipulent les types sont alors plus efficaces. Mais la détection de ces simplifications peut s'avérer coûteuse en elle-même. Il y a donc un compromis à faire dans l'implémentation.

11.3.2 Atomes disjoints

Supposons donnée une relation d'orthogonalité entre atomes de types, notée $\perp$, telle que deux atomes orthogonaux sont nécessairement disjoints ($x \perp y \Rightarrow x \mathbin{\mathbf{\wedge}} y \simeq \mathbb{0}$). On peut alors considérer des simplifications supplémentaires. Par exemple, on peut supprimer une ligne d'une forme normale disjonctive si elle contient deux atomes orthogonaux en position positive :

$$\frac{x \perp y}{\{(\{x, y\} \cup P, N)\} \cup t \rightsquigarrow t}$$

De même, on peut supprimer d'une ligne les littéraux négatifs dont l'atome est orthogonal à un atome positif de la même ligne :

$$\frac{x \perp y}{\{(\{x\} \cup P, \{y\} \cup N)\} \cup t \rightsquigarrow \{(\{x\} \cup P, N)\} \cup t}$$

Il reste à définir la relation binaire $\perp$. On peut déjà évidemment décréter que deux atomes de genre différent (par exemple un atome flèche et un atome produit) sont orthogonaux, et de même pour deux types de base disjoints (pour la représentation des combinaisons de types de base, voir la Section 11.3.4). On peut aussi dérouler les types produit. Par exemple, on sait que $t_1 \times t_2$ et $t'_1 \times t'_2$ seront disjoints dès que t_1 et t'_1 (ou t_2 et t'_2) sont disjoints. On peut aussi utiliser l'algorithme de sous-typage pour détecter des types disjoints : évidemment, cela a un coût et une circularité apparaît (parce que l'algorithme de sous-typage doit pouvoir calculer des combinaisons booléennes, et il faut donc faire attention à ne pas boucler).

11.3.3 Représentation alternative : arbres de décision

Arbres de décision binaires Plutôt que de chercher à simplifier les combinaisons booléennes représentées par des formes normales disjonctives, on peut utiliser directement d'autres représentations qui sont parfois plus compactes.

Une technique classique consiste à représenter les combinaisons booléennes par des arbres de décision binaires. Formellement, il s'agit des objets engendrés par la grammaire suivante :

$$t \quad := \quad 0 \mid 1 \mid (a?t_1 : t_2)$$

On modifie la définition d'une interprétation ensembliste ; la condition intéressante est :

$$[\![(a?t_1 : t_2)]\!] = ([\![a]\!] \cap [\![t_1]\!]) \cup (D \backslash [\![a]\!] \cap [\![t_2]\!])$$

En général, on demande à ce qu'un même atome apparaisse au plus une fois sur chaque branche, ce qui peut s'obtenir en pratique en choisissant un ordre total $\preceq$ sur les atomes et en imposant aux atomes sur chaque branche d'apparaître par ordre croissant strict. Cela garantit que si le nombre d'atomes est fini, alors le nombre de combinaisons booléennes l'est aussi. On peut définir les opérateurs de réunion, intersection, complémentaire directement sur cette représentation. Par exemple, si $t = (a?t_1 : t_2)$ et $t' = (a'?t'_1 : t'_2)$, on définit $t \vee t'$ par :

$$\begin{cases} (a?t_1 \vee t'_1 : t_2 \vee t'_2) & \text{si } a = a' \\ (a?t_1 \vee t' : t_2 \vee t') & \text{si } a \preceq a' \\ (a'?t \vee t'_1 : t \vee t'_2) & \text{si } a' \preceq a \end{cases}$$

Si l'atome à la racine de t' est plus grand que tous les atomes de t, par exemple, on voit que t' est « recopié » à chaque feuille de t ; ainsi, la taille de la représentation du résultat de l'opérateur $\vee$ n'est plus linéaire en la taille de ses arguments.

Voici les règles pour calculer $t \wedge t'$:

$$\begin{cases} (a?t_1 \wedge t'_1 : t_2 \wedge t'_2) & \text{si } a = a' \\ (a?t_1 \wedge t' : t_2 \wedge t') & \text{si } a \preceq a' \\ (a'?t \wedge t'_1 : t \wedge t'_2) & \text{si } a' \preceq a \end{cases}$$

On peut introduire diverses simplifications sur la représentation par arbres de décision binaire, par exemple : $(a?t : t) \rightsquigarrow t$.

Il est facile de revenir à une représentation en forme normale disjonctive à partir d'un arbre de décision binaire (chaque branche qui se termine sur une feuille 1 donne lieu à une « ligne »). Cela permet d'utiliser cette représentation alternative pour le stockage et les calculs sur les combinaisons booléennes en interne, tout en gardant une interface commune pour observer ces objets.

Arbres de décision ternaires Un problème avec les arbres de décisions binaire est que la taille des arbres peut exploser exponentiellement lorsque l'on calcule l'opérateur booléen réunion. Une solution est d'utiliser des arbres de décision ternaires :

$$t \quad := \quad 0 \mid 1 \mid (a?t_1 : t_2 : t_3)$$

La condition intéressante dans la définition d'une interprétation ensembliste est alors :

$$[\![(a?t_1 : t_2 : t_3)]\!] = ([\![a]\!] \cap [\![t_1]\!]) \cup [\![t_2]\!] \cup (D \backslash [\![a]\!] \cap [\![t_3]\!])$$

Il est alors possible de définir les opérateurs booléens sur cette représentation, et la taille de l'arbre qui représente la réunion de deux arbres est linéaire en la somme des tailles de ces arbres. Si $t = (a?t_1 : t_2 : t_3)$ et $t' = (a'?t'_1 : t'_2 : t'_3)$, on définit en effet $t \vee t'$ par :

$$\begin{cases} (a?t_1 \vee t'_1 : t_2 \vee t'_2 : t_3 \vee t'_3) & \text{si } a = a' \\ (a?t_1 : t_2 \vee t' : t_3) & \text{si } a \preceq a' \\ (a'?t'_1 : t \vee t'_2 : t'_3) & \text{si } a' \preceq a \end{cases}$$

Voici les règles pour calculer $t \wedge t'$:

$$\begin{cases} (a?(t_1 \vee t_2) \wedge (t'_1 \vee t_2) : 0 : (t_3 \vee t_2) \wedge (t'_3 \vee t_2)) & \text{si } a = a' \\ (a?t_1 \wedge t' : t_2 \wedge t' : t_3 \wedge t') & \text{si } a \preceq a' \\ (a'?t \wedge t'_1 : t \wedge t'_2 : t \wedge t'_3) & \text{si } a' \preceq a \end{cases}$$

(La terminaison est assurée car la représentation de $t_1 \vee t_2$ est plus petite que celle de t.) On voit qu'il est nécessaire, dans un cas, de repasser dans une représentation binaire au niveau supérieur (c'est-à-dire d'avoir 0 comme composante neutre) ; cela se fait en utilisant l'équivalence $(a?t_1 : t_2 : t_3) \simeq (a?t_1 \vee t_2 : 0 : t_3 \vee t_2)$. De même, pour calculer le complémentaire $\neg t$ où $t = (a?t_1 : t_2 : t_3)$, on prend

$$(a?\neg(t_3 \vee t_2) : 0 : \neg(t_1 \vee t_2))$$

On peut considérer plusieurs simplifications, telles que $(a?t_1 : t_2 : t_3) \rightsquigarrow t_1 | t_2$ si $t_1 = t_3$, ou $(a?t_1 : 1 : t_3) \rightsquigarrow 1$.

11.3.4 Représentation des types de base

Nous utilisons des représentations spécifiques pour les combinaisons booléennes de types de base. Ainsi, tout combinaison booléenne de types entier ou caractère (intervalles) peut être représentée de manière unique sous la forme d'une réunion d'intervalles maximaux (donc disjoints). Les combinaisons booléennes de types atome sont représentées de manière unique sous l'une des formes $a_1 \vee \ldots \vee a_n$ ou $\neg(a_1 \vee \ldots \vee a_n)$, les a_i étant chacun de la forme $ns : ln$ ou $ns : *$ et tous deux à deux disjoints (c'est-à-dire que l'on ne peut pas avoir $ns : ln$ et $ns : *$ simultanément pour le même namespace ns).

On représente ces réunions finies comme des ensembles (et non des séquences), ce qui donne l'unicité de la représentation (on minimise ainsi le nombre de types considérés). On peut calculer les opérations booléennes directement sur ces représentations, et le test de vide est trivial.

11.4 Interface avec Objective Caml

L'implémentation de ℂDuce dispose d'un système d'interface avec le langage Objective Caml. Cette interface permet :

- de réutiliser depuis les programmes ℂDuce la plupart des bibliothèques OCaml existantes ;
- de développer des projets mixtes, dans lesquels des modules OCaml et des modules ℂDuce cohabitent.

Le choix du langage Objective Caml est naturel dans la mesure où il s'agit du langage d'implémentation de ℂDuce (en particulier, les programmes ℂDuce utilisent le système de gestion mémoire d'OCaml), et où les deux langages possèdent un certains nombre de traits communs : langages fonctionnels d'ordre supérieur, avec le même genre de sémantique - stricte, avec valeurs modifiables en place et exceptions. De nombreuses bibliothèques OCaml encapsulent des bibliothèques de plus bas niveau (généralement écrites en C), et leur offrent une interface suffisamment abstraite pour ℂDuce (elles s'occupent par exemple de la gestion mémoire, du rapport d'erreur par exception, et présentent suffisamment d'information dans les types, par opposition aux en-têtes C). L'interface avec OCaml permet indirectement d'utiliser ces bibliothèques (par exemple pour accéder à des bases de données, des bibliothèques de calcul numérique, des bibliothèques systèmes, ...).

La gestion mémoire étant par nature de l'implémentation de ℂDuce commune entre ℂDuce et OCaml, il n'y a aucun problème de ce coté-là ; par exemple, il n'y a pas à faire interagir deux ramasse-miettes (*garbage collectors*) différents.

Un des défis majeurs lorsque l'on veut interfacer deux langages fortement typés est de mettre en relation les deux systèmes de types. Considérons de manière abstraite deux langages L_1 et L_2, chacun muni d'une relation de sous-typage implicite et transparente pour la sémantique. Considérons également une traduction $t \mapsto \tilde{t}$ des types de L_1 dans les types de L_2. L'idée est qu'une valeur du langage L_1, de type t, peut être traduite automatiquement en une valeur de type $\tilde{t}$ pour être utilisée dans L_2. Cette relation doit être compatible avec les relations de sous-typage : si $t_1 \leq t_2$, alors toute valeur de type t_1 peut être vue comme une valeur de type t_2, et donc traduite automatiquement en une valeur de type $\tilde{t_2}$. Cette valeur peut aussi être traduite directement en une valeur de

type $\tilde{t_1}$. Pour préserver la transparence sémantique de la subsomption, nous devons donc avoir $\tilde{t_1} \leq \tilde{t_2}$ dès que $t_1 \leq t_2$. Par conséquence, puisque OCaml n'a pas de sous-typage implicite (outre l'identité), une fonction de traduction des types ℂDuce vers les types OCaml est nécessairement constante. Une telle fonction est une traduction uniforme qui oublie toute information de type, c'est-à-dire que vues depuis OCaml, toutes les valeurs ℂDuce ont le même type, disons `CDuce.value`. En fait, l'implémentation de ℂDuce utilise naturellement un tel type OCaml pour représenter les valeurs à l'exécution, et cette représentation est visible depuis les programmes OCaml. Cependant, nous pouvons difficilement parler d'une interface typée, puisque toute information de type est perdue dans cette représentation. Nous allons définir une traduction dans l'autre sens, des types OCaml vers les types ℂDuce. Puisqu'il n'y a pas de sous-typage implicite en OCaml, la contrainte mentionnée plus haut est vide, et nous avons une grande latitude pour définir cette traduction.

La traduction des types de base ne pose pas de problème : par exemple, le type OCaml `int` est traduit en un type intervalle i—j où i (resp. j) est le plus petit (resp. le plus grand) entier représentable en OCaml. De même, le type OCaml `char` est traduit dans le sous-type de `Char` qui correspond au jeu de caractère iso-8859-1 (le segment initial d'Unicode constitué de 256 caractères). Les types OCaml structurés (flèches, produits, enregistrements) sont traduits avec les constructions similaires dans ℂDuce. Un type somme OCaml déclaré par :

`type t =` $\mathtt{A}_1$ `of` t_1 `|` ... `|` $\mathtt{A}_n$ `of` t_n

est traduit en le type ℂDuce

(‘$\mathtt{A}_1$`,`$\tilde{t_1}$`)` `|` ... `|` (‘$\mathtt{A}_n$`,`$\tilde{t_1}$`)`

Les types récursifs OCaml sont évidemment traduits en des types récursifs ℂDuce. Un traitement particulier est réservé aux types abstraits OCaml : ceux-ci sont ajoutés à l'algèbre de type de ℂDuce. Si t est un type abstrait OCaml, une valeur de type $\tilde{t}$ est, du point de vue de ℂDuce, une boite noire qui porte l'étiquette t, son contenu effectif (une valeur OCaml de type t) restant inaccessible. Garder l'étiquette t sur la valeur est nécessaire pour pouvoir effectuer un filtrage sur les types en ℂDuce. Les types abstraits paramétrés ne sont pas gérés.

Si le type OCaml t est associé au type ℂDuce $\tilde{t}$, avec les règles ci-dessus, nous avons des fonctions de traduction naturelles des valeurs OCaml de type t dans les valeurs ℂDuce de type $\tilde{t}$ et réciproquement. Ces fonctions de traduction nous servent à mettre en place le système d'interface entre les deux langages.

OCaml vers ℂDuce Le programmeur ℂDuce peut utiliser une valeur OCaml `M.v` (`M` est le nom du module OCaml dans lequel est défini la valeur `v`) dans un programme ℂDuce. Le compilateur ℂDuce cherche dans l'interface compilée de `M` le type OCaml de `M.v`, soit t, et produit le code qui la traduit en une valeur ℂDuce de type $\tilde{t}$, qui est le type donné à l'expression `M.v` dans le programme ℂDuce.

Si la valeur possède un schéma de type OCaml polymorphe (une fonction polymorphe, généralement), le programmeur doit instancier explicitement les

variables de types utilisées avec des types ℂDuce. Par exemple, le schéma de type de `List.map` dans la bibliothèque standard d'OCaml est :

$$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$$

Pour utiliser cette valeur en ℂDuce, en instanciant α avec le type `Int` et β avec le type [`Int`$*$], le programmeur doit écrire

$$\texttt{List.map}\{\texttt{Int}; [\texttt{Int}*]\}$$

(Les schémas étant définis modulo alpha-renommage, on donne l'instanciation pour les variables dans l'ordre dans lequel elles apparaissent dans le schéma de type, pour une lecture de gauche à droite.) Les fonctions de traduction au niveau des valeurs pour les variables de types OCaml instanciées avec des types ℂDuce sont des identités.

ℂDuce vers OCaml Une valeur ℂDuce v peut être vue depuis OCaml comme une valeur de type OCaml t, dès que v est bien de type $\tilde{t}$. Il n'y a pas de choix canonique pour t. Par exemple, si v a le type $(\texttt{`A}, \texttt{0—10})|(\texttt{`B}, \texttt{0—20})$, on peut la voir comme une valeur du type OCaml `t` défini par :

```
type t = A of int | B of int
```

Mais on ne veut pas *déclarer* un tel type (rappelons que deux types concrets OCaml, même s'ils ont la même définition textuelle, sont incompatibles). Par exemple, si l'application OCaml utilise déjà un type `s` défini par :

```
type s = A of int | B of int | C of string
```

on peut vouloir voir v comme une valeur de ce type là.

Le système d'interface demande au programmeur de donner explicitement un type OCaml pour les valeurs ℂDuce qu'il veut utiliser depuis des modules ℂDuce. Cela se fait en associant à une unité de compilation ℂDuce une interface de module OCaml. Le compilateur ℂDuce vérifie que les valeurs ℂDuce exportées par cette unité de compilation sont bien compatibles avec les types OCaml que le fichier d'interface leur donne, et il produit un module OCaml qui effectue la traduction des valeurs ℂDuce vers les valeurs OCaml.

11.5 Performances

Nous n'avons pas étudié la complexité des algorithmes présentés dans cette thèse. L'algorithme de sous-typage résout en particulier le problème de l'inclusion de langages réguliers d'arbres (donnés sous forme d'automates), qui est EXPTIME-complet. Dans la mesure où les types représentent en fait des automates d'arbre alternants avec complémentaire, il faut attendre une borne inférieure théorique plus grande. L'étude précise de la complexité de l'algorithme dépend évidemment de l'algorithme de calcul de prédicat inductif choisi (avec ou sans retour-arrière, voir le Chapitre 7), et de la manière de représenter et calculer les connecteurs booléens (voir le Chapitre 3 et la Section 11.3).

En pratique, et sur des exemples de programmes ℂDuce réalistes (même si relativement petits), l'ensemble des algorithmes mis en jeu dans la compilation des programmes ℂDuce semblent se comporter de manière satisfaisante.

Par exemple, le programme utilisé pour générer le site web de ℂDuce à partir d'une description XML prend moins de deux dixièmes de seconde à compiler, sur un Pentium 4 à 2.7 Ghz. La moitié de ce temps environ est passé dans l'algorithme de sous-typage, qui est appelé 19956 fois (pour un total de 70999 itérations internes). Le programme fait environ 450 lignes de code ℂDuce, plus environ 300 lignes qui correspondent à la DTD XHTML 1.0 Strict représentée sous forme de types ℂDuce. Environ 3500 nœuds de type internes sont introduits. D'autres exemples confirment le fait, constaté par Hosoya [Hos01] que malgré la grande complexité théorique de l'algorithme de sous-typage, les diverses techniques d'implémentation permettent d'obtenir un algorithme efficace au moins pour les types XML rencontrés en pratique.

Les performances à l'exécution des programmes ℂDuce semblent tout à fait satisfaisantes, en particulier grâce à l'algorithme de compilation efficace du filtrage, ainsi qu'aux techniques de représentation des valeurs à l'exécution présentées dans ce chapitre.

Nous avons délibérément choisi de ne pas présenter de mesures d'efficacité (ni, *a fortiori*, des comparaisons avec d'autres langages « concurrents »). De telles mesures évaluent tout autant la qualité de l'implémentation et celle du langage d'implémentation que l'efficacité des algorithmes. Par exemple, nous avons observé des gains ou des pertes d'un facteur 2 en efficacité en jouant simplement sur les paramètres du ramasse-miette (*garbage collector*) d'Objective Caml. De même, la suppression de code mort a entraîné une perte d'efficacité de l'ordre de 20% (probablement dûe à des questions d'alignement de code). Nous estimons que des mesures de performances ou des comparatifs ne présentent que peu d'intérêt scientifique. Le lecteur néanmoins intéressé pourra se reporter à une publication antérieure [BCF03] ou au site web de ℂDuce pour ce genre de mesures.

Conclusion

Chapitre 12

Conclusion et perspectives

Le travail présenté dans cette thèse établit les bases théoriques du langage ℂDuce. J'ai développé, en parallèle à ces travaux théoriques, une implémentation du langage ℂDuce. Un premier prototype a été réalisé dès le début de la thèse, et n'a pas été diffusé. Il a permis de tester la faisabilité des algorithmes mis en jeu. Il a été complètement réécrit, et ℂDuce 0.1 a été rendu public en juin 2003, et a permis de faire connaître le langage auprès d'une petite communauté d'utilisateurs.

L'objectif du travail de thèse, à savoir proposer un langage de programmation fonctionnel adapté à XML, avec des traits généralistes et une implémentation utilisable me semble atteint. J'ai proposé des solutions à plusieurs des questions ouvertes exposées dans la thèse d'Hosoya [Hos01] (*Records, Higher-Order Functions, Improvement of the Type Inference Algorithm, Non-linear patterns, Pattern Optimization*), ainsi que d'autres extensions à XDuce.

Il y a eu de très fortes interactions entre le travail d'implémentation et les directions prises par les travaux théoriques. Parfois, l'implémentation a même précédé la formalisation, et la formalisation *a posteriori* a permis de mieux comprendre et d'améliorer en retour l'implémentation. L'implémentation a eu un rôle essentiel dans le formalisme de l'algèbre de types (Chapitre 3) et ce sont des considérations d'assez bas-niveau (partage des structures internes) qui ont mené à la mise en place de la théorie du Chapitre 2. Évidemment, tous les résultats liés aux techniques d'implémentation et aux travaux algorithmiques ont été motivés par l'implémentation et développés en parallèle. Cette thèse est donc le fruit d'un aller-retour incessant entre théorie et mise en œuvre pratique.

L'implémentation a été réalisée dans le langage Objective Caml. Ce langage s'est révélé d'une grande aide dans la mise au point de ℂDuce et a tenu à merveille ses promesses lorsque le code évoluait de manière conséquente et nécessitait donc des modifications généralisées. Évidemment, la philosophie d'OCaml n'a pu qu'imprégner la conception de ℂDuce ; on notera des similarités syntaxiques et sémantiques importantes, ainsi que la présence d'une interface (Section 11.4) pour faire interopérer les deux langages. L'implémentation de ℂDuce compte, pour la version 0.2 diffusée en juillet 2004, environ 20 000 lignes de code. Elle repose sur un certain nombre de bibliothèques développées par la communauté des utilisateurs OCaml.

Cette thèse ne présente le langage ℂDuce lui-même que de manière succincte et assez informelle (Chapitre 10). Elle n'a pas en effet vocation à se substituer à

un manuel de référence ou à un tutoriel pour le langage. De plus, celui-ci continue à évoluer. Le lecteur intéressé pourra se reporter à d'autres sources d'information concernant ℂDuce [BCF03, Ftct04b, Ftct04a] et son implémentation.

12.1 Perspectives

Polymorphisme La question du polymorphisme, mentionnée dans la thèse d'Hosoya, est toujours d'actualité. Il s'agit d'intégrer le genre de polymorphisme paramétrique que l'on trouve dans ML au formalisme des types de XDuce ou de ℂDuce. Une des difficultés techniques est d'étendre la relation de sous-typage à des types polymorphes, en préservant d'une part une approche ensembliste, et d'autre part un algorithme efficace en pratique. Une autre difficulté est celle de l'inférence pour les variables de type instanciées lors de l'appel d'une fonction polymorphe. Enfin, il y a un choix sémantique à faire, à savoir décider si les variables de type peuvent être utilisées dans un motif (auquel cas la sémantique d'une fonction polymorphe peut dépendre de l'instanciation des variables de type, ce qui pose des problèmes pour l'inférence de ces instanciations, et ce qui peut également avoir un impact sur la représentation des valeurs à l'exécution). Un travail préliminaire [HFC05] étudiant l'ajout de polymorphisme paramétrique à XDuce a été mené, avec Hosoya et Castagna.

Rapprochement avec ML L'approche présentée dans cette thèse et poursuivie par le projet ℂDuce a été de définir un langage adapté à la manipulation de documents XML, avec des constructions spécifiques, mais qui possède néanmoins des caractéristiques généralistes issues de langages existants, comme les fonctions de première classe. Une autre direction possible est d'intégrer à un langage généraliste des caractéristiques spécifiques pour XML (types et motifs expression régulière, en premier lieu). Une « cible » naturelle serait un langage de la famille ML. Il s'agirait alors de voir comment rapprocher le système de types de XDuce/ℂDuce (propagation de types qui représentent des automates), et celui de ML (inférence par unification). Un point de départ possible serait le système de types HM(X) [OSW99], qui étend celui de ML avec des contraintes. Une autre approche serait d'introduire un unique type XML à un langage ML, et de « raffiner » ce type (par un type expression régulière), dans un formalisme du style Dependent ML [Xi98, Xi99].

Inférence En ℂDuce, les fonctions doivent être explicitement annotées par leur type. Comme ces annotations peuvent en fait changer la sémantique des programmes (car les motifs permettent de tester le type déclaré d'une fonction), il n'est pas envisageable de les supprimer complètement. On peut néanmoins chercher à les inférer dans certains cas, si l'on arrive à détecter que ces fonctions ne seront pas filtrée par un test de type, ou si l'on accepte que l'algorithme d'inférence influe sur la sémantique des programmes (en faisant un certain choix). En particulier, il serait agréable pour le programmeur de ne pas devoir spécifier le type des abstractions « anonymes », passées en argument à des fonctionnelles d'ordre supérieur. Même sans considérer le fait que les annotations changent la sémantique des fonctions, il semble inconcevable d'avoir une inférence « totale ». Une des raisons est qu'une fonction peut avoir un nombre infini de types flèches différents et incomparables (de plus, en présence de types singletons, l'ensemble

des types d'une fonction caractérisent souvent complètement la sémantique de la fonction, par exemple $\lambda x.x + 1$, ou une fonction du genre `map` sur les séquences). Les techniques d'inférence locale [PT98, PT00, OZZ01] constituent un point de départ raisonnable. En fait, la structure de l'algorithme de typage implémenté pour ℂDuce et décrit au paragraphe « Localisation des erreurs » de la Section 11.1 est proche de la propagation bidirectionnelle des types dans l'inférence locale : il propage une contrainte de manière descendante dans l'arbre de syntaxe. Dans sa version actuelle, il se contente d'utiliser cette contrainte (une borne supérieure pour le type de l'expression) pour mieux localiser les erreurs de type (et en l'absence d'erreur, il renvoie un type plus fin). Il pourrait facilement être modifié pour utiliser cette contrainte pour « remplir » les annotations à inférer pour les abstractions. Ainsi, on pourrait souvent éviter de devoir spécifier le type des abstractions anonymes utilisées comme argument de fonctionnelles d'ordre supérieur.

Représentations des valeurs dirigée par le type et le contexte L'implémentation de ℂDuce utilise une représentation uniforme pour toutes les valeurs manipulées à l'exécution. Cela signifie que la représentation en machine de la valeur permet de la connaître complètement sans disposer d'aucune information sur son type statique. XHaskell [LS04b], au contraire, traduit les types XML statiques des expressions en des types du langage hôte (Haskell, en l'occurrence). Chaque type XML possède sa propre représentation concrète, et pour pouvoir utiliser une valeur d'un certain type là où une valeur d'un super-type est attendue, il est nécessaire d'appliquer une coercion sur la valeur. Cela peut avoir un coût important, mais en contrepartie, la représentation est plus compacte, et elle permet des accès directs au milieu des séquences, là où avec la représentation uniforme, il serait nécessaire de parcourir les éléments un par un. Une approche qu'il serait intéressant d'étudier consiste à faire dépendre la représentation non seulement du type statique, mais aussi du contexte d'utilisation de la valeur. Ainsi, si l'on peut prévoir qu'une coercion devra être appliquée à la valeur, on peut l'éviter en utilisant directement la représentation attendue. Si l'on ne peut rien prévoir sur la valeur, on peut utiliser comme défaut une représentation uniforme. Évidemment, cette approche nécessite une analyse de flot d'information assez précise (on peut espérer réutiliser les travaux de Benzaken, Burelle et Castagna [BBC03] sur l'analyse de flot d'information pour la sécurité dans ℂDuce).

Partage optimal des structures cycliques La formalisme du Chapitre 2 permet de développer le reste de la théorie sans se préoccuper des questions de partage entre types structurellement « isomorphes ». Évidemment, les algorithme sur les types (en particulier l'algorithme de sous-typage) sont d'autant plus efficaces que le nombre de types différents manipulés est réduit. On a donc intérêt à partager le plus de types possibles, dans la mesure où ce partage est moins coûteux que le gain obtenu. Dans l'implémentation, les types sont partagés au moment de la traduction entre les types de la syntaxe externe et l'algèbre de types interne, modulo α-renommage. Les types X `where` $X = (X, X)$`|Int` et X' `where` $X' = (X', X')$`|Int` sont ainsi traduit en des types égaux (physiquement) dans l'implémentation, alors que les trois types récursifs X `where` $X = (X, X)$`|Int`, Y `where` $Y =$ `Int|`(Y, Y) et Z `where` $Z = ((Z, Z)$`|Int`$, (Z, Z)$`|Int)|Int`

produisent trois types différents en interne même s'il serait acceptable de les partager. Les techniques de partage optimal de termes récursifs [Mau99, Mau00, Con00] permettrait d'identifier X et Z (qui ne diffèrent que par déroulement), mais pas X et Y, qui peuvent être rendus égaux en utilisant une propriété de commutativité dans l'algèbre interne. Il serait intéressant d'adapter ces travaux pour prendre en compte ce genre de propriétés (associativité, commutativité, idempotence).

On peut facilement décrire le prédicat qui détecte l'équivalence structurelle de deux types dans l'algèbre interne sous une forme coinductive ; on s'appuie sur le fait que deux ensembles A et B sont égaux si et seulement :

$$(\forall x \in A.\ \exists y \in B.\ x = y) \wedge (\forall y \in B.\ \exists x \in A.\ x = y)$$

La négation de ce prédicat d'équivalence structurelle, est comme pour le sous-typage, une propriété inductive, et l'on peut donc utiliser les algorithmes du Chapitre 7 pour l'implémenter efficacement. Il faudrait formaliser cette technique de partage et évaluer son impact concret sur les performances du typeur.

Algèbre de combinateurs : itérateurs, chemins XML, filtres ℂDuce possède des itérateurs prédéfinis, pour travailler sur des séquences (Section 10.3.5) ou des arbres XML. Le typage de ces opérateurs est beaucoup plus précis que ce que l'on obtient avec un polymorphisme paramétrique à la ML. Il est décevant de ne pas pouvoir *définir* ce genre d'opérateurs dans le langage. Hosoya [Hos04] a introduit une opération qui généralise en même temps le filtrage à base de motifs expression régulière et l'itération sur des séquences. Il serait intéressant d'étendre cette approche pour permettre au programmeur d'intégrer également des chemins XML (à la XPath[XPA]), et de spécifier des itérateurs qui ne préservent pas nécessairement l'ordre des éléments des séquences. Une piste serait d'introduire une algèbre de combinateurs qui permet d'exprimer dans un cadre uniforme toutes ces constructions, d'étudier dessus les questions de typage et d'évaluation, et de proposer enfin des syntaxes agréables pour définir ces combinateurs. Le typage des combinateurs se ferait au moment de leur application, lorsque le type de l'expression d'entrée est complètement connu (ce qui permet d'itérer dessus). Les questions d'implémentation efficace pourraient donner lieu à des développements similaires à ceux du Chapitre 8, pour prendre en compte les types statiques.

Heuristiques et modèles de coût pour le filtrage Le formalisme du Chapitre 8 permet de considérer plusieurs stratégies pour la compilation du filtrage. Nous avons décrit une stratégie séquentielle gauche-droite, et une stratégie itérative. L'implémentation actuelle de ℂDuce utilise une stratégie séquentielle gauche-droite.

Un schéma de compilation pour produire du code globalement séquentiel pour un filtrage doit choisir, pour chaque requête, entre une stratégie gauche-droite ou une stratégie droite-gauche. Levin et Pierce [LP04] suggèrent une heuristique de « facteur de branchement maximal ». Celle-ci peut s'adapter à notre formalisme, en suggérant, pour chaque requête, de choisir soit une stratégie gauche-droite, soit une stratégie droite-gauche. Voici comment procéder. On utilise la construction de la Section 8.3.8 pour calculer la sous-requête à effectuer à gauche (dans une stratégie gauche-droite), et de même, la sous-requête à

effectuer à droite si l'on utilisait une stratégie droite-gauche. Le facteur de branchement d'une requête est la taille maximale d'une partition disjointe de cette requête, c'est-à-dire d'une partition où deux requêtes atomiques qui acceptent une valeur en commun sont nécessairement dans la même classe (la partition correspondante est donnée par la clôture transitive de la relation « accepter une valeur en commun »). L'heuristique préscrit de commencer par le coté qui maximise ce facteur de branchement. Il serait intéressant de mesurer l'impact d'une telle heuristique.

On peut également chercher à étendre de genre d'heuristique à des stratégies non nécessairement itérative. Une manière de procéder, inspirée par l'optimisation de requêtes dans les bases de données, est de définir un modèle de coût, c'est-à-dire d'associer un coût numérique à toute stratégie. On peut alors chercher à optimiser ce coût, soit de manière exacte (éventuellement en listant toutes les stratégies possibles, il n'y en a qu'un nombre fini qui n'effectuent pas plusieurs fois la même sous-requête atomique), soit de manière approchée. Définir le coût des stratégies n'est pas chose aisée : il faut prendre en compte le coût d'une sous-requête, qui elle même peut dépendre du coût du coût de la requête courante que l'on est en train de calculer, en cas de récursion.

12.2 Autour de ℂDuce

En parallèle à ma thèse, d'autres travaux ont été initiés autour du langage ℂDuce.

Analyse de sécurité pour ℂDuce La thèse de Marwan Burelle, dirigée par Véronique Benzaken et Giuseppe Castagna, porte sur des analyses de sécurité, statiques et dynamiques, pour le langage ℂDuce. Elles s'appuient sur une analyse de flot d'information, en instrumentant la sémantique du langage avec des étiquettes (qui représentent des niveaux de sécurité). Des résultats préliminaires [BBC03] ont été publiés.

Langage de requête pour ℂDuce La thèse de Cédric Miachon, dirigée par Giuseppe Castagna et Véronique Benzaken, étudie un langage de requête, appelé CQL, construit au dessus du langage ℂDuce. CQL utilise le grand pouvoir expressif des motifs et du filtrage de ℂDuce. Ces travaux ont débouché sur l'intégration dans ℂDuce d'une construction `select .. from ... where ...`. Une autre ligne de recherche, menée par Véronique Benzaken et Ioana Manolescu, cherche à interfacer ℂDuce et CQL avec des systèmes de stockage physique de documents XML ; il s'agit d'éviter de charger intégralement des documents potentiellement gigantesques en mémoire centrale, et d'utiliser des algorithmes issus du monde des bases de données pour représenter efficacement les données en mémoire de masse, avec accès aléatoire, et pour permettre l'évaluation rapide des requêtes.

Combinateurs de filtrage pour ℂDuce Kim Nguyen étudie dans sa thèse, dirigée par Véronique Benzaken, une algèbre de combinateurs qui permet de définir des opérateurs fortement polymorphes (voir plus haut) : itérateurs, chemins, transformateurs dirigés par la structure. Des résultats préliminaires ont

été obtenus lors de son stage de DEA [Ngu04] que j'ai co-encadré avec Giuseppe Castagna et Véronique Benzaken.

Canaux et sous-typage sémantique Castagna, De Nicola et Varacca [CNV04] définissent, en étendant l'approche ensembliste présentée dans cette thèse, une relation de sous-typage pour des canaux, destinée à typer un π-calcul.

Bibliographie

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4) :575–631, September 1993. (cité page 32)

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures.* M.I.T. Press, 1991. (cité page 39)

[BBC03] Véronique Benzaken, Marwarn Burelle, and Giuseppe Castagna. Security analysis for xml transformations. In *Eighth Asian Computing Science Conference*, 2003. (cité pages 223, 225)

[BCF02] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce : a white paper. In *Programming Languages Technologies for XML (PLAN-X)*, 2002. (cité page 5)

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce : An XML-centric general-purpose language. In *ACM International Conference on Functional Programming (ICFP)*, 2003. (cité pages 5, 217, 222)

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997. (cité page 32)

[CF04] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Second workshop on Programmable Structured Documents*, 2004. (cité page 5)

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1) :115–135, February 1995. (cité page 34)

[CNV04] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the π-calculs, 2004. Unpublished. (cité page 226)

[Con00] Jeffrey Considine. Efficient hash-consing of recursive types. Technical Report 2000-006, Boston University, January 2000. (cité pages 53, 224)

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 687–706. Springer-Verlag, 1994. (cité page 30)

[DCFGM02] Mariangiola Dezani-Ciancaglini, Alain Frisch, Elio Giovannetti, and Yoko Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems (ITRS)*. Electronic Notes in Theoretical Computer Science, 2002. (cité page 5)

[DG84] William Dowling and Jean Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulas. *Journal of Logic Programming*, 3 :267–284, 1984. (cité page 33)

[DOM] Document Object Model (DOM) ; `http://www.w3.org/DOM/`. (cité page 22)

[FC04] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop. (cité pages 6, 194)

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 137–146. IEEE Computer Society Press, 2002. (cité page 6)

[Fri01] Alain Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées : application au typage de XML, 2001. Rapport de DEA (Université Paris 7). (cité pages 5, 70)

[Fri04] Alain Frisch. Regular tree language recognition with static information. In *3rd IFIP International Conference on Theoretical Computer Science (TCS)*, 2004. A preliminary version appeared in the PLAN-X 2004 workshop. (cité pages 5, 32)

[Ftct04a] Alain Frisch and the ℂDuce team. ℂDuce : Tutorial, 2004. `http://www.cduce.org/tutorial.html`. (cité page 222)

[Ftct04b] Alain Frisch and the ℂDuce team. ℂDuce : User's manual, 2004. `http://www.cduce.org/manual.html`. (cité page 222)

[GLP00] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. In *ACM SIGPLAN Notices*, volume 35(9), pages 221–231, 2000. (cité page 32)

[GLPS04] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. Xml goes native : Run-time representations for xtatic, 2004. (cité page 209)

[GP03] Vladimir Gapeyev and Benjamin Pierce. Regular object types. In *European Conference on Object-Oriented Programming*, 2003. (cité page 30)

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *Proceedings of the 32st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005. (cité pages 6, 222)

[HM02] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, 2002. (cité page 33)

[HM03] Haruo Hosoya and Makoto Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, 2003. (cité page 33)

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001. (cité pages 19, 23, 31, 71, 134, 217, 221)

[Hos03] Haruo Hosoya. Regular expression pattern matching - a simpler design, 2003. (cité page 31)

[Hos04] Haruo Hosoya. Regular expression filters for XML. 2004. (cité page 224)

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce : A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databas es (WebDB2000)*, 2000. (cité page 23)

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001. (cité page 23)

[HP02] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *Journal of Functional Programming*, volume 13(4). 2002. (cité page 23)

[HP03] Haruo Hosoya and Benjamin C. Pierce. A typed XML processing language. In *ACM Transactions on Internet Technology*, volume 3(2), pages 117–148. 2003. (cité page 23)

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000. (cité page 23)

[KPMI95] Dexter Kozen, Jens Palsberg, and Schwartzbach Michael I. Efficient recursive subtyping. In *Mathematical Structures in Computer Science*, volume 5(1), pages 113–125, 1995. (cité page 32)

[Lev03] Michael Y. Levin. Matching automata for regular patterns. In *International Conference on Functional Programming (ICFP)*, 2003. (cité page 32)

[LP04] Michael Y. Levin and Benjamin C. Pierce. Type-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004. (cité pages 32, 168, 224)

[LS04a] Kenny Zhuo Ming Lu and Martin Sulzmann. An implementation of subtyping among regular expression types. In *ASIAN Symposium on Programming Languages and Systems*, 2004. (cité page 209)

[LS04b] Kenny Zhuo Ming Lu and Martin Sulzmann. XHaskell : Regular expression types for haskell, 2004. (cité pages 30, 209, 223)

[Mau99] Laurent Mauborgne. *Representation of Sets of Trees for Abstract Interpretation*. PhD thesis, École Polytechnique, 1999. (cité pages 53, 224)

[Mau00] Laurent Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4) :290–311, 2000. (cité pages 53, 224)

[MS03] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *Declarative Programming in the Context of OO-Languages (DP-COOL)*, 2003. (cité page 210)

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Symposium on Principles of Database Systems*, pages 11–22, 2000. (cité page 24)

[Nam] Namespaces in XML ; `http://www.w3.org/TR/REC-xml-names`. (cité page 190)

[Ngu04] Kim Nguyễn. Une algèbre de filtrage pour le langage $\mathbb{C}$Duce, 2004. Rapport de DEA (Université Paris 7). (cité page 226)

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999. (cité page 222)

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, 2001. (cité page 223)

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1998. (cité page 223)

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), 2000. (cité page 223)

[REL] RELAX NG Specification ; `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`. (cité page 21)

[SCH] XML Schema Part 1 : Structures ; `http://www.w3.org/TR/xmlschema-1`. (cité page 21)

[SH04] Tadahiro Suda and Haruo Hosoya. A non-backtracking top-down algorithm for checking tree automata containment, 2004. Unpublished. (cité page 33)

[Van04] Stijn Vansummeren. Type inference for unique pattern matching, 2004. (cité page 192)

[VM04] Jerome Vouillon and Paul-André Melliès. Semantic types : a fresh look at the ideal model for types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 52–63. ACM Press, 2004. (cité page 31)

[Wat94] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Technical Report Computing Science Note 93/43, Eindhoven University of Technology, May 1994. (cité page 53)

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. (cité page 222)

[Xi99] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999. (cité page 222)

[XML] Extensible Markup Language Recommendation (XML) 1.0 ; `http://www.w3.org/TR/REC-xml`. (cité pages 19, 20)

[XPA] XML Path Language (XPath) ; `http://www.w3.org/TR/xpath`. (cité page 224)