Collège doctoral

N° attribué par la bibliothèque

|_|_|_|_|_|_|_|_|_|_|_|_|

T H È S E

pour obtenir le grade de
Docteur de l'École des Mines de Paris
Spécialité « Informatique Temps Réel, Robotique et Automatique »

présentée et soutenue publiquement par
Alexandre FREY

le 18 juin 2004

APPROCHE ALGÉBRIQUE DU TYPAGE D'UN LANGAGE À LA ML AVEC OBJETS, SOUS-TYPAGE ET MULTI-MÉTHODES

Directeur de thèse : Didier RÉMY

Jury

M. Giuseppe CASTAGNA	Rapporteur
M. François FAGES	Rapporteur
M. Dominique BOLIGNANO	Examinateur
M. Guy COUSINEAU	Examinateur
M. Thomas JENSEN	Examinateur

Version 2.0 (Soutenance) du 7 juin 2004

Résumé

Les langages à objets offrent une forme particulière de polymorphisme en permettant l'écriture de « méthodes » dont l'exécution dépend du type dynamique des arguments qu'on lui passe. Le plus souvent, cette opération de « dispatch dynamique » ne prend en compte qu'un argument unique (**`this`**, **`self`**,...). Cependant, certains langages permettent le dispatch simultané sur tous les arguments et on parle alors de « multi-méthodes ».

Cette thèse s'intéresse à la définition et au typage d'un langage dérivant de ML, purement fonctionnel, d'ordre supérieur et avec multi-méthodes. Les objets sont construits en supposant données des déclarations explicites de classes, de façon similaire aux types concrets de ML. Les multi-méthodes sont introduites comme une forme particulière de filtrage sur les objets.

La présentation du système de types utilise une approche algébrique. Plutôt que de figer l'ensemble des types, on en axiomatise les propriétés nécessaires pour la correction du système. Cela permet d'écrire des preuves génériques qui ne dépendent pas du choix de l'algèbre. On montre ainsi, sous des hypothèse très générales, comment réduire la vérification automatique du typage à la résolution de problèmes simples exprimés dans un langage logique du premier ordre (contraintes). La résolution des problèmes de contraintes peut alors réutiliser le corpus de résultats disponibles dans la littérature.

L'avantage de cette approche générique est qu'elle permet de traiter d'un coup toute une classe de langages possibles se distinguant par la nature de l'algèbre de types, du langage d'expression des contraintes et du modèle d'interprétation de ces contraintes. Elle offre également un outil intéressant pour étudier le typage dans un contexte où le monde d'interprétation est ouvert, c'est-à-dire quand on souhaite que le typage d'un module apporte une garantie pour toutes les utilisations possibles de ce module.

Remerciements

Je tiens à remercier tout particulièrement **Didier Rémy** d'avoir accepté de me guider pendant la phase de rédaction de cette thèse. Tout en me laissant une totale liberté sur le plan scientifique, il a su me donner des conseils efficaces pour recentrer ce mémoire de thèse sur un sujet cohérent et de taille raisonnable. De plus, sa relecture du manuscrit a été extrêmement précise.

Je remercie **Gérard Berry** et **François Bourdoncle** pour m'avoir encadré à l'École des Mines et **Jean Vuillemin** pour sa participation stimulante à la partie expérimentale de ce travail (langage Jazz). Ensemble, ils m'ont fait découvrir un vaste sujet, allant de la sémantique des circuits synchrones au typage des multi-méthodes, et dont une petite partie seulement est représentée dans ce mémoire.

Je suis reconnaissant à **Giuseppe Castagna** et **François Fages** d'avoir bien voulu être rapporteurs de cette thèse. Leurs remarques nombreuses et pénétrantes m'ont ouvert les yeux sur certaines subtilités du système de types présenté ici et m'ont permis d'en améliorer la présentation.

Je remercie **Guy Cousineau** et **Thomas Jensen** de me faire l'honneur de s'intéresser à ce travail et de participer à mon jury de thèse.

Je remercie particulièrement **Dominique Bolignano** pour ses encouragements constants et pour m'avoir aidé à concilier la rédaction de cette thèse et mon travail à Trusted Logic. Je suis heureux qu'il ait accepté de faire partie de mon jury.

Je remercie **Xavier Leroy**, **François Pottier** et **Eduardo Giménez** qui, à différents stades de la rédaction, ont relu le manuscrit, ce qui m'a permis de l'améliorer considérablement.

Je remercie tous les amis, collègues et membres de ma famille qui m'ont aidé, soutenu et encouragé pendant ce travail.

Il est évident que je n'aurais pas pu terminer cette thèse sans l'appui indéfectible de mon épouse **Maria Vonica-Frey**. Je l'en remercie ici du fond du cœur.

Table des matières

Introduction

La plupart des langages de programmation modernes définissent une notion de typage : l'association d'un type à chaque donnée manipulée par le programme permet de vérifier si une opération est bien définie sur les données auxquelles elle est appliquée. On évite ainsi toute une classe d'erreurs de programmation, comme par exemple l'addition d'une chaîne de caractères à un entier, ou bien la concaténation deux entiers. Ces opérations incorrectes peuvent compromettre des propriétés essentielles attendues du programme : correction, portabilité, stabilité ou sécurité. Le typage contribue donc à améliorer la qualité du logiciel.

Le typage est dit statique quand il est réalisé avant même l'exécution du programme, lors de la compilation, de l'édition de lien ou bien lors de l'activation du programme. Quand le typage est statique, les erreurs peuvent être détectées indépendamment du chemin d'exécution effectivement emprunté par le programme, ce qui offre un niveau de sûreté supérieur au typage dynamique. Cela permet aussi de réaliser des gains de performance en temps et en espace par rapport au typage dynamique où l'exécution de chaque opération doit vérifier le type des arguments et où la représentation en mémoire de chaque objet manipulé doit transporter une information de type.

En contrepartie de ces avantages, le typage statique est généralement complexe à mettre en œuvre : il faut en effet procéder à une analyse abstraite du programme suivant un ensemble de règles logiques appelé système de types qui aboutit à accepter ou rejeter un programme. L'une des propriétés essentielles d'un système de types est sa correction : aucun programme ne doit être accepté s'il peut mener à une opération invalide, c'est-à-dire non définie. Cependant, les contraintes d'implémentation (décidabilité, performances) amènent souvent à rejeter des programmes en fait corrects, mais trop complexes pour être analysés comme tel. Cette limitation conduit à rechercher des système de types toujours plus expressifs, c'est-à-dire qui rejettent de moins en moins de programmes qu'on juge utile de pouvoir exprimer.

Dans cette recherche d'expressivité, le polymorphisme paramétrique [Str67, Str00, Hin69, Mil78] a marqué une étape importante en autorisant l'écriture de fonctions génériques opérant sur des données de différents types. Une fonction polymorphe peut ainsi manipuler certaines données comme des boîtes noires seulement susceptibles d'être transférées d'une structure de données à une autre (tuple, liste, arbre) ou bien d'être passées en argument à d'autres fonctions. On peut alors donner à une telle fonction un schéma de types incluant des paramètres, chaque paramètres pouvant varier indépendamment dans l'ensemble de tous les types possibles. Par exemple, un algorithme de tri, écrit une fois pour toute, peut s'appliquer à une liste de n'importe quel type pourvu qu'une fonction de comparaison soit fournie. Le polymorphisme paramétrique s'adapte ainsi particulièrement bien aux langages où les fonctions sont des données comme les autres, c'est-à-dire les langages applicatifs d'ordre supérieur, notamment de la famille de ML.

Il existe cependant d'autres formes de polymorphisme. Le polymorphisme dit « *ad hoc* » permet à une même fonction d'être définie sur plusieurs types tout en ayant un comportement différent sur chaque type. Un exemple typique concerne les opérateurs arithmétiques. Selon la nature des opérandes (entiers, flottants), l'opérateur d'addition par exemple s'exécute ainsi : si ses deux arguments sont deux entiers, on calcule l'addition entière ; pour deux flottants, on calcule l'addition flottante ; enfin, si l'un des argument est entier et l'autre flottant, on doit convertir l'argument entier en flottant puis retourner la somme flottante avec l'autre argument.

Un autre genre de polymorphisme est présent dans les langages de programmation centrés sur la notion d'« objet », comme C++ [Str91], Eiffel [Mey92] ou Java [GJSB00]. Dans la tradition impérative de ces langages, on considère qu'un objet est une sorte d'automate, doté d'un état interne (les « champs ») et de fonctions de transition modifiant cet état (les « méthodes »). Un objet est construit sur le modèle d'une « classe » spécifiant le nom des champs et le corps des méthodes. L'une des grandes forces des langages à objets est qu'ils permettent aux classes d'être définies par héritage, c'est-à-dire en étendant d'autres classes écrites précédemment et en en redéfinissant certaines de leurs méthodes. On peut ainsi écrire du code générique qui manipule des objets sans connaître leur type précis, mais en sachant qu'ils dérivent d'une classe donnée et savent répondre aux méthodes de cette classe. Cette forme de polymorphisme est particulièrement utile dans les systèmes logiciels construits incrémentalement à partir d'un noyau générique et soumis à des changements fréquents. On pense par exemple aux interfaces graphiques où les différents objets graphiques (menus, fenêtres, boutons, etc.) peuvent tous dériver d'une classe de base définissant un petit nombre de méthodes (retourner la taille, se réafficher sur une portion de l'écran, etc.), ce qui permet de coder une fois pour toute la gestion de l'affichage tout en autorisant l'ajout ultérieur d'objets graphiques variés. Plus généralement, l'expérience montre que tout gros système logiciel subit nécessairement des changements constants et qu'il est important de pouvoir s'appuyer sur des classes génériques et réutilisables, ce qui explique que les objets se sont imposés comme un outil incontournable de génie logiciel.

Pour refléter les polymorphismes ad-hoc et d'héritage, un système de types statique doit disposer d'une notion de sous-typage. Partout où un objet d'un certain type est attendu, on autorise ainsi la substitution par un objet d'un sous-type. Comparés au typage des langages comme ML, les systèmes de types des langages à objets sont généralement assez frustres (pas d'inférence, ordre 1). Parfois, ils ne sont même pas corrects et supposent que des vérifications dynamiques ont lieu à l'exécution. Mais ils intègrent tous cette notion de sous-typage.

Combiner les polymorphismes paramétriques et ad-hoc, le sous-typage, les objets et les fonctions dans un système de types supérieurement expressif a motivé de nombreux travaux de recherche [Mit84, FM88, FM89, AC93, Rém94, Pot98]. La plupart de ces travaux étudient les objets en les codant sous la forme d'enregistrements extensibles, les méthodes étant contenues dans ces enregistrements sous la forme de fonction ordinaires prenant en argument l'objet lui-même. Les systèmes de types résultant présentent souvent une grande complexité technique (types récursifs ou d'ordre supérieur, contraintes de sous-typage, etc.). De plus, il ne parviennent qu'imparfaitement à traiter un problème parfois très important : les méthodes binaires [BCC+95].

Le problème des méthodes binaires, ou plus généralement des multi-méthodes, provient d'une limitation intrinsèque de la vision traditionnelle des objets et par conséquent de la modélisation par enregistrements extensibles : quand une méthode est appelée sur un objet, le code effectivement exécuté dépend de la classe de l'objet, ce qui permet la généricité et l'extensivité du programme. Ce qui est parfois trop restrictif est qu'il dépend *uniquement* de cette classe, ce qui ne permet pas, par exemple, d'écrire ses propres opérateurs arithmétiques dont l'exécution dépend *simultanément* des

types de tous ses arguments.

Certains langages autorisent le programmeur à écrire de telles « multi-méthodes » sur ses propres hiérarchies d'objets extensibles, ce qui démultiplie la souplesse et l'extensibilité du langage. Parmi ces langages, on peut citer CLOS [Ste90, Pae93], Cecil [Ct02, CL95] et Dylan [Sha96]. L'étude des systèmes de types statiques adaptés aux langages avec objets et multi-méthodes (parfois appelées « fonctions surchargées ») a donné lieu à moins de travaux que l'approche avec enregistrements extensibles [WB89, CGL92, BM96, BM97].

L'origine de cette thèse : circuits, nombres et multi-méthodes

Dans cette thèse, nous nous intéressons particulièrement au système $\mathrm{ML}_{\leqslant}$ décrit dans [BM96]. Voici quelques caractéristiques de ce système : $\mathrm{ML}_{\leqslant}$ est revendiqué comme une extension naturelle de ML avec objets et multi-méthodes car les déclarations de classes et les définitions de multi-méthodes sont vues comme une extension des types concrets et du filtrage de ML. Les schémas de types font apparaître des contraintes de sous-typage sur les paramètres de type et il n'y a pas besoin de types récursifs.

Notre intérêt pour les multi-méthodes et le système $\mathrm{ML}_{\leqslant}$ est issu de l'objectif initial de ce travail de thèse : la redéfinition du langage 2Z [BVB94]. Ce langage d'expérimentation et d'enseignement est spécialisé dans la description et la synthèse de circuits synchrones arithmétiques. Ses fonctionnalités sont similaires à celles de VHDL [VHD93] ou Verilog [Ver01]. Sa particularité est d'intégrer des types arithmétiques comme les nombres 2-adiques [Vui94]. Le polymorphisme ad-hoc des opérateurs arithmétiques sur ces types est considéré comme une caractéristique essentielle du langage car il permet l'écriture de fonctions génériques qui soit provoque la synthèse d'un circuit, soit la vérification statique de ce circuit, selon le type des arguments passés à la fonction.

Dans sa version initiale, le langage 2Z disposait d'un système de types ad-hoc, mal spécifié et qui ne permettait pas la définition de types utilisateur. L'un des objectifs principaux de la redéfinition du langage était d'ajouter les objets et de disposer d'un système de types plus riche et mieux connu. Étant donnée l'importance du polymorphisme des opérateurs arithmétiques dans ce langage, on s'est naturellement tourné vers les multi-méthodes et le système $\mathrm{ML}_{\leqslant}$ a été développé à cet effet. Le résultat de ce travail de réimplémentation et d'intégration du nouveau système de types est le langage Jazz [FBB$^+$].

À l'occasion de l'implémentation de Jazz, l'intégration de $\mathrm{ML}_{\leqslant}$ dans un vrai langage a posé des problèmes pratiques et théoriques. En effet, le système de types s'est rapidement révélé trop rigide pour le besoin du langage car les seules contraintes exprimables dans le système sont des inégalités de sous-typage et leur conjonction, ce qui les force à opérer sur des hiérarchies de classes connexes. En pratique, il est très vite apparu nécessaire de pouvoir exprimer des fonctions polymorphes sur des hiérarchies de classes séparées. Par exemple, les opérateurs booléens devait pouvoir s'appliquer à tout élément d'une « algèbre booléenne », comme les nombres binaires ou les BDDs, alors que les types de ces deux genres d'algèbres ne sont pas en relation de sous-typage. Il fallait donc étendre les contraintes exprimables avec des prédicats plus généraux que le sous-typage.

Une autre faiblesse de $\mathrm{ML}_{\leqslant}$ est que son modèle d'interprétation des contraintes est trop limité. Il présente certes une simplicité et des caractéristiques désirables. Par exemple, les types sont des termes finis, contrairement aux approches des objets par enregistrements extensibles, ce qui se rapproche beaucoup du modèle initial de ML. Ensuite, la hiérarchie de classes spécifiée par le programmeur

n'est pas figée : si un programme est bien typé, alors il reste bien typé dans toutes les extensions de la hiérarchie courante, une propriété essentielle pour capturer l'extensibilité des systèmes à objets. En pratique cependant, le modèle de $ML_{\leqslant}$ a des rigidités gênantes. Le sous-typage est nécessairement structurel, c'est-à-dire que deux types ne peuvent être comparés que s'ils ont le même squelette. Par exemple, List⟨int⟩ est comparable à List⟨float⟩ (et en est un sous-type). Mais, List⟨int⟩ ne peut jamais avoir un sous-type non paramétré de la forme ListInt. Par ailleurs, l'extensibilité du monde introduite dans $ML_{\leqslant}$ est en fait trop générale. Elle considère que toute hiérarchie de classe qui contient la hiérarchie au moment du typage en est une extension, alors que le langage Jazz ne permet l'extension que par le bas (définition de sous-classes) et l'héritage simple. Du coup certains programmes sont rejeté car ils sont incorrects dans une extension du monde qui n'est pas exprimable dans le langage, ce qui est trop restrictif.

Certains de ces problèmes ont été traités dans Jazz et dans des travaux connexes [Bon02b]. L'intégration dans la présentation initiale de $ML_{\leqslant}$ s'est alors révélée difficile. En particulier, la preuve de correction donnée dans [BM96] repose sur une présentation non standard de la sémantique opérationnelle, qui est typée et couplée de façon compliquée avec le système de types. C'est gênant car l'exécutif de Jazz repose sur une machine virtuelle non typée, très similaire à ce qu'on peut faire en ML. Finalement, la complexité intrinsèque du typage des multi-méthodes en monde ouvert et le caractère fermé et rigide de $ML_{\leqslant}$ faisait qu'il était souvent délicat de se convaincre, même informellement, de la correction de certaines extensions apportées à Jazz.

Objectifs

Pour assurer une base saine au typage de Jazz, il nous apparaît nécessaire de donner une nouvelle présentation du système de types $ML_{\leqslant}$, ce qui est l'objet de cette thèse.

À cet effet, nous nous fixons les objectifs suivants :

1. donner une présentation standard d'un calcul avec objets et multi-méthodes proche du langage Jazz afin de faciliter la compréhension du système et la comparaison avec des calculs existants ;
2. donner une formalisation souple et ouverte du système de types capable d'intégrer avec un minimum d'effort des extensions variées. En particulier, nous voulons que la preuve de correction, partie la plus critique de la formalisation, puisse être réutilisée facilement.

Contribution de cette thèse

Cette thèse apporte trois contributions principales.

Premièrement, nous définissons une extension de ML avec objets et multi-méthodes et nous lui donnons une sémantique opérationnelle non typée et par réduction, selon une approche éprouvée. Dans ce langage, nous approfondissons l'analogie entre multi-méthode et filtrage : grâce à des motifs autorisant le filtrage d'objets, les deux notions sont en fait pratiquement unifiées.

Deuxièmement, nous introduisons un système de types générique pour ce langage. Ce système utilise des algèbres de types dont nous donnons une définition axiomatique. Cela nous permet de montrer la correction du système par une preuve elle-même générique et qui ne dépend que des axiomes posés sur les algèbres de types. La formalisation du système de types met en œuvre un traitement original

du polymorphisme, basé sur une approche extensionnelle, les types polymorphes étant de simples ensembles de types monomorphes. La preuve elle-même utilise cependant des techniques standard (réduction du sujet).

Troisièmement, nous montrons comment réduire un système de vérification de types à la résolution de problèmes simples (des contraintes) exprimés dans un langage logique du premier ordre. La syntaxe et l'interprétation de cette syntaxe sont laissées ouvertes et doivent simplement respecter quelques propriétés de base. Nous utilisons le cadre standard de la théorie des modèles pour réaliser cette réduction.

Dans cette thèse, nous n'abordons pas la résolution des problèmes élémentaires de typage, étape indispensable pour obtenir un système en pratique. Nous nous focalisons simplement sur la correction du typage et la réduction du typage d'un programme à la résolution de problèmes élémentaires logiques. Le traitement des contraintes (résolution, implication, simplification) est en effet un thème déjà largement abordé ailleurs [FM88, FM89, Tiu92, AC93, TS96, Fre97, Pot98, Sim03, KR03] alors que le typage des multi-méthodes est un sujet moins développé et qui présente des difficultés spécifiques. Nous soutenons qu'une approche algébrique peut permettre de réutiliser les résultats existants concernant les contraintes.

Finalement, l'approche algébrique du typage des multi-méthodes à la $\mathrm{ML}_{\leqslant}$ en présence d'un monde ouvert suggère également de nouvelles formulations de problèmes de contraintes qui mériteraient d'être approfondies.

Plan de l'ouvrage

Le reste de cette thèse est organisée comme suit. Dans le chapitre 1, nous donnons la syntaxe du langage. Dans le chapitre 2, nous en donnons la sémantique opérationnelle. Dans le chapitre 3, nous introduisons les algèbres de types et les règles d'un système de types générique avec approche extensionnelle du polymorphisme. Dans le chapitre 4, nous montrons la correction de ce système de types algébrique vis-à-vis de la sémantique opérationnelle. Remarquant que ce système algébrique n'est pas directement utilisable en pratique sans annotation du programmeur, nous introduisons à cet effet au chapitre 5 une notion très générale de langage de types, ce qui nous permet d'écrire les annotations de type. Nous définissons ensuite un modèle de typage comme une interprétation d'un langage de types dans une algèbre, ce qui nous permet de donner une spécification d'un système de vérification de types opérant sur des expressions annotées par le programmeur. La correction du système est prouvée par effacement des annotations et en utilisant le système du chapitre 3. Dans un dernier développement formel au chapitre 6, nous montrons comment la vérification de types peut être réduite indépendamment du modèle choisie en un ensemble de problèmes élémentaires (implications de contraintes, tests de couvertures). Au chapitre 7, nous discutons plus informellement des problèmes et perspectives soulevés par ce travail. Nous abordons en particulier l'hypothèse d'un monde ouvert ainsi que l'inférence de type. Dans ce chapitre, nous comparons également cette thèse avec d'autres travaux apparentés.

Notations

Voici quelques notations utilisées tout au long du document.

$E \# E'$ signifie que les ensembles E et E' sont disjoints, c'est-à-dire que $E \cap E' = \varnothing$. La notation $E_1 \# \cdots \# E_n$ signifie que les ensembles E_i sont disjoints deux-à-deux.

Le domaine de définition d'une fonction f est noté $\mathrm{dom}(f)$.

La notation $a \mapsto b$ désigne la fonction f de domaine $\{a\}$ telle que $f(a) = b$.

La restriction d'une fonction f à un ensemble E est notée $f \restriction E$.

Pour deux fonctions f et g, $f \oplus g$ est la fonction de domaine $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ telle que :

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{si } x \in \mathrm{dom}(g) \\ f(x) & \text{si } x \notin \mathrm{dom}(g) \text{ et } x \in \mathrm{dom}(f) \end{cases}$$

Les autres notations et définitions seront introduites au cours du texte en PETITES CAPITALES. On en trouvera ci-dessous un index. Pour certains chapitres, constitués d'une présentation progressive des notions suivie d'un récapitulatif formel des définitions, le renvoi à la définition formelle est marqué en gras dans l'index.

Chapitre 1

Syntaxe

Ce chapitre a pour objectif de présenter la syntaxe du langage étudié et sa sémantique opérationnelle intuitive.

Le langage est formé d'EXPRESSIONS engendrées par la grammaire suivante :

Expressions	e	$::= x$	Identificateur
		$\mid e\,e$	Application
		$\mid \mathbf{fun}\ x\ \Rightarrow e$	Fonction
		$\mid \mathbf{let}\ x = e\ \mathbf{in}\ e$	Définition locale
		$\mid \mathbf{fix}\ x\ \Rightarrow e$	Définition récursive
		$\mid a$	Constante
		$\mid f$	Opération primitive
		$\mid C\,\{\ell = e; \ldots; \ell = e\}$	Objet
		$\mid C \cdot \ell$	Accesseur de champs
		$\mid \mathbf{meth}\ \{\pi \Rightarrow e; \ldots; \pi \Rightarrow e\}$	Multi-méthode définie cas par cas
Motifs	π	$::= \_$	Motif universel
		$\mid \varpi$	Motif primitif
		$\mid C\,\{\ell = \pi; \ldots; \ell = \pi\}$	Motif de sous-classe
		$\mid \#C\,\{\ell = \pi; \ldots; \ell = \pi\}$	Motif de classe exacte
		$\mid \pi\ \mathbf{as}\ x$	Motif lieur

1.1 Noyau ML

Dans cette grammaire, x désigne un IDENTIFICATEUR de variable, pris dans un ensemble Vars supposé infini. Les quatre premières productions (identificateur, application, **fun** et **let**) définissent le noyau ML habituel. La stratégie d'évaluation n'est pas fixée car les résultats de cette thèse en sont indépendants. Elle peut être en appel par nom ou en appel par valeur.

L'expression $\mathbf{fix}\ x\ \Rightarrow e$ définit récursivement une valeur égale à l'expression e dans laquelle l'identificateur x est considéré lié à la valeur en train d'être définie. Par exemple, la fonction factorielle

peut s'écrire ainsi :

$$\textbf{fix}\ \mathit{Fact} \Rightarrow \textbf{fun}\ n \Rightarrow \textbf{if}\ (n \leqslant 0)\ \textbf{then}\ 1\ \textbf{else}\ (n \times (\mathit{Fact}\ (n-1)))$$

Par souci de simplification, nous avons séparé la définition d'une valeur récursive de son utilisation. La construction **let rec** est donc une simple abréviation :

$$\textbf{let rec}\ x = e_1\ \textbf{in}\ e_2 \quad \equiv \quad \textbf{let}\ x = (\textbf{fix}\ x \Rightarrow e_1)\ \textbf{in}\ e_2$$

Si la sémantique est en appel par nom, la construction **fix** n'est pas limitée à la définition de fonctions récursives. On peut aussi s'en servir pour construire des structures infinies récursives. Par exemple, la liste infinie composée du seul élément 1 peut s'écrire comme suit :

$$\textbf{fix}\ l \Rightarrow 1\ \texttt{::}\ l$$

De même, la liste infinie des nombres entiers peut se coder ainsi :

$$(\textbf{fix}\ \mathit{seq} \Rightarrow \textbf{fun}\ i \Rightarrow i\ \texttt{::}\ \mathit{seq}\ (i+1))\ 0$$

1.2 Classes et objets

L'expression $C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ représente un OBJET construit avec la classe C.

L'ensemble des CLASSES doit être spécifié par le programmeur d'une façon non précisée ici. Nous supposerons simplement donné l'ensemble des classes, noté $\mathsf{Classes}$. Cet ensemble est organisé en hiérarchie, c'est-à-dire qu'il est muni d'un ordre partiel $\sqsubseteq$ de SOUS-CLASSEMENT[1]. Quand $C' \sqsubseteq C$, on dit que C' est une SOUS-CLASSE de C ou bien que C est une SUPER-CLASSE de C'.

Chaque classe C contient un ensemble fini éventuellement vide de CHAMPS. Ceux-ci sont identifiés par leur nom, tiré d'un ensemble infini d'étiquettes fixé une fois pour toutes et noté $\mathsf{Étiquettes}$. L'ensemble des champs d'une classe C est noté Champs_C. Les classes ne comprennent que des champs, à l'exclusion des méthodes qui sont définies en dehors des classes (voir section 1.4).

Quand on construit un objet, la valeur de tous les champs de cette classe doit être donnée. Pour que l'expression $C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ soit bien formée, on demande donc que les labels $\ell_1, \ldots, \ell_n$ représentent exactement l'ensemble Champs_C et qu'ils soient distincts deux à deux. L'ordre d'apparition des champs dans un objet n'est pas important. On considérera par exemple que les deux expressions suivantes sont identiques :

$$\mathsf{Point}\,\{x = 1; y = 2\} \quad \text{et} \quad \mathsf{Point}\,\{y = 2; x = 1\}$$

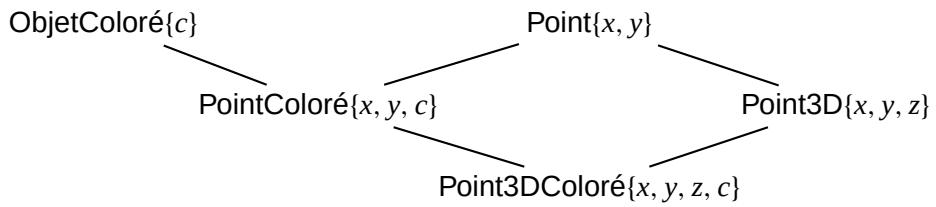
L'accès aux champs d'un objet se fait avec l'expression fonctionnelle $C{\cdot}\ell$ qui prend en argument un objet construit avec une sous-classe quelconque C' de C et qui renvoie la valeur du champ de cet objet nommé ℓ. On voit donc qu'on peut accéder à un objet sans connaître sa classe précise, à condition d'en connaître une super-classe. C'est une propriété clé des langages à objets qui permet l'écriture de fonction générique sur les objets. Pour que cette propriété soit bien fondée, il faut bien sûr

[1] Tous les résultats de cette thèse restent corrects si la relation de sous-classement est simplement un préordre, c'est-à-dire n'est pas supposée antisymétrique.

que les champs soit hérités d'une classe à une sous-classe. Nous posons donc en axiome la propriété suivante :

$$C' \sqsubseteq C \Rightarrow \mathsf{Champs}_C \subseteq \mathsf{Champs}_{C'}$$

Exemple. Voici un exemple de hiérarchie de classes qui illustre l'héritage des champs (ceux-ci sont représentés à côté des classes) :



■

Remarque. Contrairement à la plupart des formalisations de langages à objets, notons bien que l'accès à un champ ne peut se faire sans mentionner une super-classe de l'objet attendu. En d'autres termes, il n'est pas possible d'accéder à un champ simplement en écrivant $e \cdot \ell$, indépendamment de la classe de e.

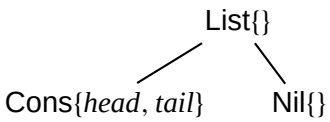
Ce choix, hérité de $\mathrm{ML}_{\leqslant}$, se justifie d'abord par des considérations sur la conception d'un langage de programmation. Imaginons par exemple qu'en plus de la hiérarchie ci-dessus, le programmeur ait défini la classe Cercle contenant les champs c et r, désignant respectivement le centre du cercle (un Point) et son rayon. Si elle était autorisée, l'expression $e \cdot c$ présenterait alors un polymorphisme trop général pour ne pas être gênant. En effet, on ne voudra certainement jamais accéder par une expression unique à des choses aussi différentes que la couleur d'un objet ou le centre d'un cercle. La syntaxe que nous présentons ici limite délibérément l'expressivité du langage pour éviter ce genre de situations et force le programmeur à préciser explicitement son intention.

Notons aussi que cette restriction n'est pas définitive. Nous pensons qu'il est souvent possible de considérer le langage de ce chapitre comme un langage primitif dans lequel on réécrit un langage source donné disposant de l'accès inconditionnel aux champs. Par exemple, considérons un langage source où le programmeur ne déclare aucune classe mais repose sur un système d'enregistrements extensibles. On peut implicitement déclarer une classe pour chaque ensemble possible de champs les hiérarchiser en suivant l'ordre d'extension des enregistrements ($C' \sqsubseteq C$ ssi $C \subseteq C'$). L'accès inconditionnel au champ ℓ peut alors s'écrire en utilisant l'accesseur explicite $\{\ell\} \cdot \ell$. On voit donc que le langage source peut dans ce cas facilement se coder dans notre langage. Dans d'autres cas, on peut également utiliser les méthodes (voir section 1.4) qui fournissent entre autres un moyen de mixer arbitrairement dans une même fonction des accès à des champs quelconques. ■

Il est utile d'interdire l'utilisation de certaines classes pour la construction d'objets et de limiter leur utilisation pour l'accès aux champs. Une telle classe est dite ABSTRAITE et une classe effectivement autorisée à construire un objet est dite CONCRÈTE. L'ensemble de toutes les classes concrètes est noté $\mathsf{ClassesConcrètes} \subseteq \mathsf{Classes}$ et sa définition est laissé à l'initiative du programmeur qui pourra par exemple utiliser le mot clé **`abstract`** pour désigner une classe abstraite lors de sa déclaration.

Exemple. Pour définir le type concret des listes chaînées, le programmeur peut typiquement introduire

la hiérarchie suivante, où List est une classe abstraite et Cons et Nil sont deux classes concrètes :



La liste `[1; 2; 3]` s'écrit alors de la façon suivante :

$$\mathsf{Cons}\left\{head = 1;\ tail = \mathsf{Cons}\left\{head = 2;\ tail = \mathsf{Cons}\left\{head = 3;\ tail = \mathsf{Nil}\,\{\}\right\}\right\}\right\}$$

Pour alléger les notations, on se permettra les abréviations classiques suivantes :

$$\begin{array}{rcl} e_1 \texttt{ :: } e_2 & \equiv & \mathsf{Cons}\,\{head = e_1;\ tail = e_2\} \\ \texttt{[]} & \equiv & \mathsf{Nil}\,\{\} \\ \texttt{[}e_1\texttt{; }\ldots\texttt{; }e_n\texttt{]} & \equiv & e_1 \texttt{ :: } (e_2 \texttt{ :: } (\ldots \texttt{ :: } (e_n \texttt{ :: [])}\ldots)) \end{array}$$ ■

Pour tout entier $n \geqslant 0$, on suppose qu'il existe une classe concrète $\texttt{Tuple}_n$ dont l'ensemble des champs est $\{1, \ldots, n\}$. Cette classe permet de manipuler des tuples d'expressions en utilisant l'abréviation suivante :

$$\langle e_1, \ldots, e_n\rangle \quad \equiv \quad \texttt{Tuple}_n\,\{1 = e_1; \ldots; n = e_n\}$$

Les classes de tuples sont utiles pour pouvoir écrire des définitions multiples ou des fonctions ou méthodes à plusieurs arguments. Par exemple, voici une façon d'écrire des définitions mutuellement récursives :

$$\left.\begin{array}{l} \texttt{let rec} \\ \quad x_1 = e_1 \texttt{ and} \\ \quad\quad \vdots \\ \quad x_n = e_n \\ \texttt{in} \\ \quad e \end{array}\right\} \equiv \left\{\begin{array}{l} \texttt{let } x = \\ \quad \texttt{fix } x \Rightarrow \\ \quad\quad \texttt{let } x_1 = (\texttt{Tuple}_n \cdot 1\, x) \texttt{ in} \\ \quad\quad\quad \vdots \\ \quad\quad \texttt{let } x_n = (\texttt{Tuple}_n \cdot n\, x) \texttt{ in} \\ \quad\quad\quad \langle e_1, \ldots, e_n\rangle \\ \texttt{in} \\ \quad \texttt{let } x_1 = (\texttt{Tuple}_n \cdot 1\, x) \texttt{ in} \\ \quad\quad \vdots \\ \quad \texttt{let } x_n = (\texttt{Tuple}_n \cdot n\, x) \texttt{ in} \\ \quad\quad e \end{array}\right.$$

De même, les fonctions à plusieurs arguments seront notées de préférence :

$$\texttt{fun } \langle x_1, \ldots, x_n\rangle \Rightarrow e \quad \equiv \quad \texttt{fun } x \Rightarrow \left(\begin{array}{l} \texttt{let } x_1 = (\texttt{Tuple}_n \cdot 1\, x) \texttt{ in} \\ \quad\quad \vdots \\ \texttt{let } x_n = (\texttt{Tuple}_n \cdot n\, x) \texttt{ in} \\ \quad e \end{array}\right)$$

Nous voyons ici quelques codages intéressants qui utilisent les classes de tuple. Ces classes seront aussi utilisées pour les arguments des primitives et pour les multi-méthodes. Pour assurer la validité de ces codages, il est important que les classes de tuple soient minimales dans l'ordre de sous-classement, de sorte que l'accesseur $\mathbf{Tuple}_n \cdot i$ ne puisse être appliqué qu'à un tuple de taille n et à aucun autre objet. Nous faisons de cette propriété essentielle un axiome de toute structure de classes[2] :

$$\forall C \in \mathsf{Classes}, C \sqsubseteq \mathbf{Tuple}_n \Rightarrow C = \mathbf{Tuple}_n$$

Remarque. Une limitation importante du langage est l'absence d'effets de bord : il n'y a pas d'opérateur de remplacement d'un champ. La nature équationnelle et en appel par nom du langage Jazz justifie cette limitation. La présence d'effets de bord seraient toutefois une extension intéressante à considérer, notamment pour ses conséquences sur le système de types. ■

1.3 Constantes et primitives

Les constantes, désignées par la lettre a, sont prises dans un ensemble PrimVals de valeurs primitives fixé à l'avance. Les opérations primitives sont notées avec la lettre f et sont prises dans un ensemble PrimOps de fonctions partielles qui opèrent sur l'ensemble des valeurs primitives. On note $|f|$ l'arité de l'opérateur f, c'est-à-dire le nombre d'arguments qu'il accepte.

Exemple. Pour fixer les idées, on peut supposer que les valeurs primitives comprennent les entiers rationnels et les booléens : $\mathsf{PrimVals} = \mathbb{Q} \cup \{\mathsf{true}, \mathsf{false}\}$. Les rationnels pourront être représentés par un couple d'entiers relatifs n/d irréductibles.

Les opérations primitives pourront être l'addition $+$, la multiplication $\times$ et les tests de comparaison $\leqslant$ et $=$ toutes d'arité 2 et de domaine $\mathbb{Q}$.

On se permettra d'utiliser une notation infixe pour certaines opérations. Par exemple, $e_1 + e_2$ sera une abréviation pour $+\,\langle e_1, e_2\rangle$. ■

Contexte. Notons que les opérations primitives ne sont pas currifiées, c'est-à-dire que l'addition de deux expression e_1 et e_2 s'écrit $+\,\langle e_1, e_2\rangle$ et non pas $(+\,e_1)\,e_2$. Les présentations traditionnelles des petits langages fonctionnels utilisent généralement la forme currifiée pour toutes les fonctions, ce qui permet de se passer des constructeurs de tuple ou de fonction n-aires natives sans perdre en expressivité. En contrepartie, la définition de la sémantique est légèrement plus compliquée : il faut attendre qu'une opération primitive soit appliquée au bon nombre d'argument pour pouvoir réduire l'application et il faut tenir compte du fait qu'une primitive partiellement appliquée est une valeur fonctionnelle. D'un autre côté, dans la section suivante, nous allons considérer les multi-méthodes comme des méthodes à un argument qui peuvent utiliser des motifs tuple pour extraire plusieurs expressions de leur argument. On utilise donc naturellement les tuples pour représenter les multiples arguments d'une multi-méthode. Par régularité, nous avons donc privilégié les fonctions n-aires, suivant en cela l'exemple du langage SML/NJ [MTHM97] plutôt que d'OCAML [LDG+03]. ■

[2] En Java, on dirait que la classe $\mathbf{Tuple}_n$ est « finale ».

1.4 Méthodes

L'expression **meth** $\{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}$ est une MÉTHODE. C'est une expression fonctionnelle qui est définie par une liste de CAS de la forme $\pi_j \Rightarrow e_j$. Chaque cas d'une méthode est conditionné par un MOTIF π_j qui sert à FILTRER l'argument passé à la méthode et à lier des identificateurs avec certaines sous-expressions de l'argument de la méthode.

Le MOTIF UNIVERSEL _ accepte toute valeur (*wildcard*). Un MOTIF PRIMITIF ϖ filtre les valeurs primitives. Nous supposons donné une fois pour toutes pour chaque motif primitif l'ensemble des constantes acceptées par ce motif.

Le motif $C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$ accepte les objets construits avec la classe C ou l'une de ses sous-classes. Les étiquettes $\{\ell_1, \ldots, \ell_n\}$ doivent représenter les champs de la classe C et les motifs $\pi_1, \ldots, \pi_n$ servent à contraindre la valeur des champs de l'objet. Si on ne veut pas poser de contrainte particulière sur un champs ℓ_i, il suffit d'utiliser _ pour le motif π_i. Le motif $\#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$ accepte uniquement les objets de la classe C et pas de ses sous-classes.

Enfin le motif π **as** x permet de lier l'identificateur x avec la valeur filtrée par π à l'exécution. On ne considère que des motifs LINÉAIRES, c'est-à-dire où les variables liées n'apparaissent qu'une fois au plus.

Exemple. Si PrimVals contient les entiers rationnels et les booléens, on peut supposer que les motifs primitifs suivants sont disponibles :
- le motif a qui accepte exactement la valeur a ;
- le motif **rat** qui accepte tous les nombres rationnels (y compris les nombres entiers) ;
- le motif **int** qui accepte tous les nombres entiers relatifs (sous-ensemble $\mathbb{Z}$ de $\mathbb{Q}$) ;
- le motif **bool** qui accepte les deux booléens **true** et **false**. ■

L'exécution d'une méthode consiste à choisir le cas le plus précis parmi tous les motifs π_j qui filtrent l'argument de la méthode (stratégie « *best match* »). La méthode toute entière est alors équivalente à l'expression e_j où on remplace les variables liées dans π_j par les sous-expressions correspondantes de l'argument de la méthode. Cette opération est appelée DISPATCH DYNAMIQUE. Elle suppose donc qu'on définisse une relation de comparaison de précision entre motifs. Cette relation est telle que plus un motif contraint les valeurs filtrées, plus il est précis. Par exemple, le motif universel qui ne pose aucune contrainte est le moins précis de tous les motifs. Le motif $\#C\,\{\ldots\}$ est plus précis que $C\,\{\ldots\}$ car il n'accepte que les objets exactement construits avec la classe C, alors que $C\,\{\ldots\}$ accepte aussi les sous-classes.

Exemple. Voici par exemple la méthode qui renvoie le carré de la norme d'un point :

$$\begin{array}{l}\textbf{meth}\,\{\\ \quad \mathsf{Point}\,\{x = \_\ \textbf{as}\ x;\ y = \_\ \textbf{as}\ y\} \Rightarrow x \times x + y \times y;\\ \quad \mathsf{Point3D}\,\{x = \_\ \textbf{as}\ x;\ y = \_\ \textbf{as}\ y;\ z = \_\ \textbf{as}\ z\} \Rightarrow x \times x + y \times y + z \times z\}\end{array}$$

Pour alléger l'écriture, il est utile d'introduire un peu de sucre syntaxique. Dans un motif filtrant les objets, on se permettra d'omettre les champs non contraints. Par exemple :

$$\begin{array}{rcl}\mathsf{Point}\,\{x = \textbf{int}\} & \equiv & \mathsf{Point}\,\{x = \textbf{int}; y = \_\}\\ \mathsf{Point} & \equiv & \mathsf{Point}\,\{x = \_; y = \_\}\end{array}$$

On utilisera aussi les notations suivantes pour les classes de base $\texttt{Tuple}_n$ et les classes de liste :

$$\begin{array}{rcl}
\langle \pi_1, \ldots, \pi_n \rangle & \equiv & \#\texttt{Tuple}_n\,\{1 = \pi_1; \ldots; n = \pi_n\} \\
\pi_1 \texttt{ :: } \pi_2 & \equiv & \#\mathsf{Cons}\,\{\mathit{head} = \pi_1; \mathit{tail} = \pi_2\} \\
\texttt{[]} & \equiv & \#\mathsf{Nil}\,\{\} \\
\texttt{[}\pi_1\texttt{;} \ldots \texttt{;}\pi_n\texttt{]} & \equiv & \pi_1 \texttt{ :: } (\pi_2 \texttt{ :: } (\ldots \texttt{ :: } (\pi_n \texttt{ :: [])}\ldots))
\end{array}$$

Dans l'exemple de la méthode qui renvoie la norme d'un point, l'implémentation sur la classe Point est automatiquement héritée dans toutes les sous-classes de Point sauf celles où l'implémentation pour Point3D est plus précise. On obtient donc automatiquement une implémentation correcte sur les classes comme PointColoré. Il arrive que cet héritage de l'implémentation d'une méthode ne soit pas approprié. Par exemple, si l'on veut définir le prédicat d'égalité sur les points, on veut prendre en compte tous les champs et on souhaite donc interdire que l'implémentation de la méthode sur les Points soit héritée par mégarde dans les sous-classes. Dans ce cas, il est plus judicieux d'utiliser les motifs de la forme *#C* :

$$\begin{array}{l}
\texttt{meth}\,\{ \\
\quad \langle \#\mathsf{Point}\,\{x = _\ \texttt{as}\ x_1; y = _\ \texttt{as}\ y_1\}, \#\mathsf{Point}\,\{x = _\ \texttt{as}\ x_2; y = _\ \texttt{as}\ y_2\}\rangle \Rightarrow \\
\qquad x_1 = x_2 \ \texttt{and}\ y_1 = y_2; \\
\quad \langle \#\mathsf{Point3D}\,\{x = _\ \texttt{as}\ x_1; y = _\ \texttt{as}\ y_1; z = _\ \texttt{as}\ z_1\}, \#\mathsf{Point3D}\,\{x = _\ \texttt{as}\ x_2; y = _\ \texttt{as}\ y_2; z = _\ \texttt{as}\ z_2\}\rangle \Rightarrow \\
\qquad x_1 = x_2 \ \texttt{and}\ y_1 = y_2 \ \texttt{and}\ z_1 = z_2; \\
\quad \vdots\} \\
\end{array}$$

En effet, le code qui teste l'égalité sur la classe Point n'est typiquement pas valide pour les sous-classes des points colorés comme PointColoré car il faudrait aussi prendre en compte l'égalité des couleurs. En utilisant le motif #Point dans ce cas, on évite qu'une l'implémentation invalide soit héritée. Ceci est particulièrement intéressant si la méthode est formée en agrégeant des cas écrits dans différents modules.

Pour terminer, voici quelques autres exemples :

$$\begin{array}{l}
\texttt{fix}\ \mathit{append} \Rightarrow \texttt{meth}\,\{ \\
\quad \langle \texttt{[]}, _\ \texttt{as}\ l_2\rangle \Rightarrow l_2; \\
\quad \langle _\ \texttt{as}\ h \texttt{ :: } _\ \texttt{as}\ t, _\ \texttt{as}\ l_2\rangle \Rightarrow h \texttt{ :: } \mathit{append}\,\langle t, l_2\rangle\} \\
\mathit{ifthenelse} = \texttt{meth}\,\{ \\
\qquad\qquad \langle \mathsf{true}, _\ \texttt{as}\ \mathit{then}, _\rangle \Rightarrow \mathit{then}; \\
\qquad\qquad \langle \mathsf{false}, _, _\ \texttt{as}\ \mathit{else}\rangle \Rightarrow \mathit{else}\}
\end{array}$$

■

1.5 Récapitulatif

Dans cette section, nous rappelons les définitions introduites dans le chapitre et nous fixons complètement les notations utilisées par la suite

Présupposés

On suppose donnés une fois pour toutes :
- un ensemble infini Vars d'IDENTIFICATEURS ;
- un ensemble infini $\mathsf{Étiquettes}$ de symboles de LABELS dont on suppose qu'il contient notamment les nombres entiers $1, 2, \ldots, n$.

Structure primitive

On suppose fixée une fois pour toutes une structure primitive $\mathcal{C}^0$ qui consiste en :
- un ensemble $\mathsf{PrimVals}$ de VALEURS PRIMITIVES ;
- pour chaque entier n non-nul, un ensemble $\mathsf{PrimOps}_n$, éventuellement vide, de fonctions partielles $f \in \mathsf{PrimVals}^n \rightarrow\!\!\!\!\!\rightarrow \mathsf{PrimVals}$, appelés OPÉRATEURS PRIMITIFS n-AIRES. La réunion de tous les ensembles $\mathsf{PrimOps}_n$ est notée simplement $\mathsf{PrimOps}$, ensemble de tous les OPÉRATEURS PRIMITIFS. L'ARITÉ $|f|$ d'un opérateur f est l'indice n de l'ensemble $\mathsf{PrimOps}_n$ auquel il appartient ;
- un ensemble $\mathsf{PrimMotifs}$ de MOTIFS PRIMITIFS, chaque motif primitif ϖ étant un sous-ensemble des valeurs primitives : $\varpi \subseteq \mathsf{PrimVals}$.

Structure de classes

Une STRUCTURE DE CLASSE $\mathcal{C}$ est la donnée des éléments suivants :
- un ensemble $\mathsf{Classes}$ de CLASSES ;
- un sous-ensemble $\mathsf{ClassesConcrètes} \subseteq \mathsf{Classes}$ de CLASSES CONCRÈTES ;
- pour chaque classe $C \in \mathsf{Classes}$, un ensemble fini de labels $\mathsf{Champs}_C \subseteq \mathsf{Étiquettes}$, appelés CHAMPS de C ;
- une relation $(\sqsubseteq) \in \mathsf{Classes} \times \mathsf{Classes}$ appelée SOUS-CLASSEMENT et qui doit être une relation d'ordre partiel.

Les axiomes suivants doivent être respectés par toute structure de classes :

(HÉRITAGE) $\quad \forall C_1, C_2 \in \mathsf{Classes}, C_1 \sqsubseteq C_2 \Rightarrow \mathsf{Champs}_{C_2} \subseteq \mathsf{Champs}_{C_1}$

($\mathtt{Tuple}_n$) $\quad \forall n \in \mathbb{N}, \mathtt{Tuple}_n \in \mathsf{ClassesConcrètes} \wedge \mathsf{Champs}_{\mathtt{Tuple}_n} = \{1, \ldots, n\}$

($\mathtt{Tuple}_n$-MINIMAL) $\quad \forall n \in \mathbb{N}, \forall C \in \mathsf{Classes}, C \sqsubseteq \mathtt{Tuple}_n \Rightarrow C = \mathtt{Tuple}_n$

Motifs

Les motifs sont les termes engendrés par la grammaire suivante où les conditions syntaxiques de bonne formation figure sur la droite :

$\pi ::= \_$		
$\mid \varpi$	$\varpi \in \mathsf{PrimMotifs}$	
$\mid C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$	$C \in \mathsf{Classes}, fv(\pi_1) \# \ldots \# fv(\pi_n)$	(1)
$\mid \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$	$C \in \mathsf{ClassesConcrètes}, fv(\pi_1) \# \ldots \# fv(\pi_n)$	(1)
$\mid \pi$ **as** x	$x \in \mathsf{Vars}, x \notin fv(\pi)$	

Notes :

(1) Dans ces motifs les labels $\ell_1, \ldots, \ell_n$ doivent être deux-à-deux distincts, ils doivent représenter exactement l'ensemble Champs_C et leur ordre d'apparition est indifférent.

L'ensemble des variables apparaissant dans un motif π est noté $fv(\pi)$.

L'ensemble des motifs est noté Motifs.

Expressions

Les expressions sont les termes engendrés par la grammaire suivante :

$e ::= x$	$x \in \mathsf{Vars}$	
$\mid e_1\, e_2$		
$\mid$ **fun** $x \Rightarrow e$	$x \in \mathsf{Vars}$	
$\mid$ **let** $x = e_1$ **in** e_2	$x \in \mathsf{Vars}$	
$\mid$ **fix** $x \Rightarrow e$	$x \in \mathsf{Vars}$	
$\mid a$	$a \in \mathsf{PrimVals}$	
$\mid f$	$f \in \mathsf{PrimOps}$	
$\mid C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$	$C \in \mathsf{ClassesConcrètes}, \mathsf{Champs}_C = \{\ell_1, \ldots, \ell_n\}$	(1)
$\mid C \cdot \ell$	$C \in \mathsf{Classes}, \ell \in \mathsf{Champs}_C$	
$\mid$ **meth** $\{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}$	(2)	

Notes :

(1) Les labels doivent être distincts deux à deux et leur ordre d'apparition n'est pas signifiant ;

(2) L'ordre d'apparition des cas n'est pas signifiant.

L'ensemble des expressions est noté Exprs.

Variables libres

L'ensemble (fini) des variables libres $fv(e)$ d'une expression e est défini par induction sur e :

$$\begin{aligned}
fv(x) &\triangleq \{x\} \\
fv(e_1\, e_2) &\triangleq fv(e_1) \cup fv(e_2) \\
fv(\mathbf{fun}\ x\ \Rightarrow e_1) &\triangleq fv(e_1) \setminus \{x\} \\
fv(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) &\triangleq fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\
fv(\mathbf{fix}\ x\ \Rightarrow e) &\triangleq fv(e) \setminus \{x\} \\
fv(C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}) &\triangleq fv(e_1) \cup \cdots \cup fv(e_n) \\
fv(C \cdot \ell) &\triangleq \varnothing \\
fv(f) &\triangleq \varnothing \\
fv(a) &\triangleq \varnothing \\
fv(\pi \Rightarrow e) &\triangleq fv(e) \setminus fv(\pi) \\
fv(\mathbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}) &\triangleq fv(\pi_1 \Rightarrow e_1) \cup \ldots \cup fv(\pi_n \Rightarrow e_n)
\end{aligned}$$

Notations

Si l'on veut expliciter la structure de classes $\mathcal{C}$ utilisée, on écrira les différents ensembles définis ci-dessus de la façon suivante : $\mathsf{Classes}^{\mathcal{C}}$, $\mathsf{ClassesConcrètes}^{\mathcal{C}}$, $\mathsf{Champs}^{\mathcal{C}}_{C}$, $\sqsubseteq^{\mathcal{C}}$, $\mathsf{Motifs}^{\mathcal{C}}$ et $\mathsf{Exprs}^{\mathcal{C}}$.

Chapitre 2

Sémantique

Nous allons maintenant donner un sens formel aux expressions du langage. Le style de sémantique adopté consiste à donner des règles d'évaluation qui permettent de réduire peu à peu une expression en un résultat (sémantique par réécriture à « petits pas »). L'objectif de ce chapitre est donc de définir un prédicat $e \longrightarrow e'$ qu'il faut lire « l'expression e' peut se déduire à partir de e en faisant un pas de calcul ».

2.1 Noyau ML

La première règle d'évaluation concerne l'application d'une fonction et correspond à la règle (β) du λ-calcul (voir par exemple [Bar84]) :

(BETA) $$(\texttt{fun}\ x\ \Rightarrow e_1)\, e_2 \longrightarrow [x \mapsto e_2]e_1$$

Dans cette règle, $[x \mapsto e_2]e_1$ est obtenue à partir de e_1 en substituant les occurrences libres de la variable x par e_2. L'opération de substitution est techniquement délicate à définir car il faut faire très attention à éviter les captures de variables. La définition se trouve à la fin de ce chapitre (section 2.7, page 43).

L'évaluation des expressions `let` et `fix` se définit de façon similaire à celle de l'application à l'aide des substitutions :

(BETA-LET) $$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow [x \mapsto e_1]e_2$$

(BETA-FIX) $$(\texttt{fix}\ x\ \Rightarrow e) \longrightarrow [x \mapsto (\texttt{fix}\ x\ \Rightarrow e)]e$$

2.2 Objets

La sémantique de l'accès aux objets est résumée par la règle suivante :

(BETA-OBJ) $$C \cdot \ell_i\ (C'\ \{\ell_1 = e_1; \ldots; \ell_i = e_i; \ldots; \ell_n = e_n\}) \longrightarrow e_i$$

Dans cette règle, on ne suppose pas que C' est une sous-classe de C. C'est en effet l'un des objectifs du système de types d'assurer cette propriété, de telle sorte que l'interprète n'ait pas besoin de la vérifier dynamiquement.

Quand on a effectivement $C' \sqsubseteq C$, notons que ℓ_i est forcément dans l'ensemble des champs de C pour que l'expression $C \cdot \ell_i$ soit bien formée. Par l'axiome (HÉRITAGE), il s'en suit qu'il est aussi forcément un champ de C', de sorte que la règle de réduction est toujours possible dans ce cas.

2.3 Primitives

L'évaluation d'une opération primitive f suppose qu'elle soit appliquée à un tuple de valeurs primitives dont le nombre d'arguments correspond à l'arité de l'opération. On réécrit alors simplement cette application en la valeur calculée par la fonction f :

(BETA-PRIM)
$$\frac{f \in \mathsf{PrimOps}_n \qquad a' = f(a_1, \ldots, a_n)}{f\ \langle a_1, \ldots, a_n \rangle \longrightarrow a'}$$

Notons que cette règle ne s'applique que si $(a_1, \ldots, a_n)$ est bien dans le domaine de f.

2.4 Méthodes

2.4.1 Filtrage

La relation de filtrage relie une expression et un motif. Cette relation est compliquée par le fait que la sémantique du langage peut être en appel par nom. L'expression peut donc n'être pas encore assez réduite pour que l'on puisse décider si le filtrage est vrai ou faux. C'est pourquoi, nous en ferons une relation dans une logique à trois valeurs : $\mathsf{filtre}(e, \pi)$ peut valoir : vrai s'il est établi que e est bien filtrée ou acceptée par π ; faux s'il est définitivement établi que e est rejetée ; ou bien le symbole $\perp$ si e n'est pas assez réduite pour décider si elle est filtrée ou rejetée par π.

Pour définir la relation de filtrage, nous introduisons maintenant les valeurs. Une VALEUR ou FORME NORMALE DE TÊTE est une expression close suffisamment réduite pour qu'on puisse décider sa tête. Les valeurs sont engendrées par la grammaire suivante où on fait figurer le symbole de tête de la valeur sur la droite :

$$\begin{array}{lcl}
V ::= a & \Rightarrow & a \\
\quad \mid C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\} & \Rightarrow & C \\
\quad \mid f & \Rightarrow & \texttt{fun} \\
\quad \mid C \cdot \ell & \Rightarrow & \texttt{fun} \\
\quad \mid \texttt{fun}\ x\ \Rightarrow e & \Rightarrow & \texttt{fun} \\
\quad \mid \texttt{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\} & \Rightarrow & \texttt{fun}
\end{array}$$

La tête d'une valeur V est notée tête(V). Notons qu'un objet peut être une valeur même si ses champs ne le sont pas. Cela est particulièrement utile dans une sémantique en appel par nom. Par exemple, la liste 1 `::` $(\texttt{fix}\ l\ \Rightarrow 1$ `::` $l)$ est une valeur.

Nous pouvons maintenant donner la définition de la relation de filtrage (voir section 2.7, page 41 pour une version complètement formelle). Celle-ci procède par induction sur la structure du motif filtrant, un motif lieur π `as` x étant considéré équivalent pour le filtrage avec π.

Tout d'abord, toute expression e, quel que soit son niveau de réduction, est filtrée par le motif universel _ : on a donc $\mathsf{filtre}(e, \_) = \mathsf{vrai}$ pour tout e, même si ce n'est pas une valeur. Nous supposons maintenant que le motif filtrant n'est pas le motif universel.

Dans tous les autres cas, le filtrage n'est déterminé que si l'expression filtrée est suffisamment réduite pour qu'on puisse en déterminer la tête, c'est-à-dire qu'elle doit être une valeur. Si e n'est pas une valeur, alors le filtrage est indéfini et on note $\mathsf{filtre}(e, \pi) = \bot$. Si e est une valeur, alors $\mathsf{filtre}(e, \pi)$ vaut vrai ou faux selon les règles qui suivent.

Le filtrage par un motif primitif ϖ réussit si et seulement si la valeur e est une constante élément de l'ensemble ϖ. Il échoue si la valeur n'est pas une constante, par exemple si c'est un objet ou une fonction, ou bien si c'est bien une constante, mais qu'elle n'est pas dans ϖ.

Si le motif filtrant π est de la forme $\#C\,\{\ell_i = \pi_i\}$, alors le filtrage réussit si et seulement si la valeur e est un objet construit avec la classe C, c'est-à-dire de la forme $C\,\{\ell_i = e_i\}$, et le filtrage de chacun des champs e_i par le motif π_i réussit. Le filtrage de e par π échoue si e n'est pas un objet ou bien si c'est un objet construit avec une classe différente de C ou bien si c'est bien un objet construit avec la classe C, mais si un de ses champs e_i n'est pas filtré par le motif correspondant π_i, c'est-à-dire si $\mathsf{filtre}(e_i, \pi_i) = \mathsf{faux}$ pour un certain i. Enfin, le filtrage est indéfini dans tous les autres cas, c'est-à-dire quand e est bien un objet construit avec C, qu'aucun des sous-filtrages $\mathsf{filtre}(e_i, \pi_i)$ n'échoue, mais qu'un au moins d'entre eux est indéfini. Notons que si e est bien un objet construit avec la classe C, on peut reformuler la définition de la valeur du filtrage $\mathsf{filtre}(e, \pi)$ comme la conjonction $\mathsf{filtre}(e_1, \pi_1) \wedge \cdots \wedge \mathsf{filtre}(e_n, \pi_n)$ où l'opération $\wedge$ désigne la conjonction la plus « paresseuse » (*et parallèle*).

Il reste le cas où le motif filtrant π est de la forme $C\,\{\ell_i = \pi_i\}$. Dans ce cas, la valeur e doit être un objet construit avec une sous-class C' de C. Cet objet est donc de la forme $C'\,\{\ell_1 = e_1; \ldots; \ell_n = e_n; \ldots; \ell_{n'} = e_{n'}\}$. La valeur du filtrage $\mathsf{filtre}(e, \pi)$ vaut alors la conjonction $\mathsf{filtre}(e_1, \pi_1) \wedge \cdots \wedge \mathsf{filtre}(e_n, \pi_n)$. Notons que les champs $e_{n+1}, \ldots, e_{n'}$ n'interviennent pas dans la définition du filtrage.

Exemple. Considérons l'expression e suivante :

$$e = [\mathsf{Point}\,\{x = 1; y = (\mathbf{fun}\ x\ \Rightarrow x + 1)\ 2\}\ ;e_2]$$

Voici quelques exemples de filtrage :

π	$\mathsf{filtre}(e, \pi)$
$\mathsf{Point}\,\{x = 1\}\ ::\ \_$	vrai
$\mathsf{Point}\,\{x = 2\}\ ::\ \_$	faux
$\mathsf{Point}\,\{y = \mathsf{int}\}\ ::\ \_$	$\bot$

■

2.4.2 L'ordre de précision sur les motifs

Un motif est d'autant plus précis qu'il accepte moins de valeurs. Parmi deux motifs acceptant exactement les mêmes valeurs, on peut toutefois distinguer un motif plus précis. Par exemple, le motif $\#C$ est considéré comme strictement plus précis que C, même si C n'a pas de sous-classe différente de C.

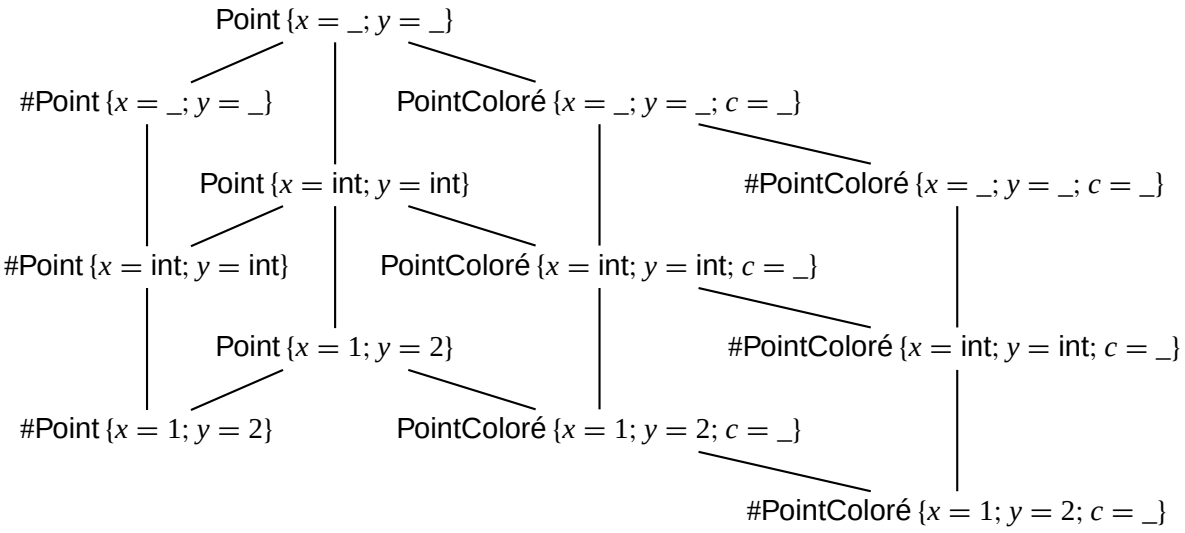
La relation de comparaison de précision est notée $\leqslant$, l'assertion $\pi \leqslant \pi'$ se lisant « π est plus précis que π' ». C'est une relation de pré-ordre (réflexive et transitive) définie par induction sur la structure des deux motifs π et π' :

- le motif universel est moins précis que tout autre motif : $\pi \leqslant \_$;
- la présence d'un lieur ne change pas la comparaison de précision : π **as** x est équivalent à π ;
- un motif primitif est d'autant plus précis qu'il filtre moins de valeurs primitives, c'est à dire que ϖ_1 est plus précis que ϖ_2 si et seulement si $\varpi_1 \subseteq \varpi_2$ (rappelons que les motifs primitifs sont définis comme l'ensemble des constantes qu'ils filtrent) ;
- l'ordre de précision se transmet dans les champs : $C\left\{\ell_i = \pi'_i\right\}$ est plus précis que $C\,\{\ell_i = \pi_i\}$ si chacun des π'_i est plus précis que π_i ; de même que $\#C\left\{\ell_i = \pi'_i\right\}$ est plus précis que $\#C\,\{\ell_i = \pi_i\}$ dans les même conditions.
- à motifs de champs égaux, les motifs qui filtrent les objets sont d'autant plus précis qu'ils mentionnent des classes plus précises ; quand $C' \sqsubseteq C$, on a donc :

$$C'\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n; \ldots; \ell_{n'} = \pi_{n'}\} \leqslant C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}\,;$$

- le motif $\#C\,\{\ell_i = \pi_i\}$ est plus précis que $C\,\{\ell_i = \pi_i\}$;

Exemple. Voici un extrait de l'ordre de précision pour quelques motifs :



Notons que quand $C' \sqsubseteq C$, un motif de la forme $\#C'\,\{\ldots\}$ *n'est pas plus précis* que $\#C\,\{\ldots\}$. En effet, ces deux motifs acceptent des ensembles de valeurs disjoints. ■

2.4.3 Projection

Quand une expression e est filtrée par le motif π, on peut extraire de e les sous-expressions correspondant aux lieurs dans π. Cette opération est appelée PROJECTION et est notée $e \downarrow \pi$. Son résultat est une fonction associant une sous-expression de e à toutes les variables libres de π. La projection se

définit aisément par induction sur le motif :

$$\begin{array}{rcl} e\downarrow\_ & \triangleq & \varnothing \\ e\downarrow\varpi & \triangleq & \varnothing \\ e\downarrow(\pi\ \mathbf{as}\ x) & \triangleq & e\downarrow\pi \oplus [x\mapsto e] \\ (C'\,\{\ell_1=e_1;\ldots;\ell_n=e_n;\ldots\})\downarrow(C\,\{\ell_1=\pi_1;\ldots;\ell_n=\pi_n\}) & \triangleq & e_1\downarrow\pi_1\oplus\cdots\oplus e_n\downarrow\pi_n \\ (C\,\{\ell_1=e_1;\ldots;\ell_n=e_n\})\downarrow(\#C\,\{\ell_1=\pi_1;\ldots;\ell_n=\pi_n\}) & \triangleq & e_1\downarrow\pi_1\oplus\cdots\oplus e_n\downarrow\pi_n \end{array}$$

Exemple. Considérons le motif et l'expression suivants :

$$\pi = (\mathsf{Point}\,\{x=\mathsf{int}\ \mathbf{as}\ x\}\ \mathbf{as}\ p)\ \mathbf{::}\ (\_\ \mathbf{as}\ \mathit{tail})$$
$$e = [\,\mathsf{PointColoré}\,\{x=1;y=2;c=3\}\,;\mathsf{Point}\,\{x=1;y=2\}\,]$$

On vérifie qu'on a bien $\mathsf{filtre}(e,\pi)=\mathsf{vrai}$. La projection de e selon π est alors la fonction suivante :

$$e\downarrow\pi = [x\mapsto 1;\ p\mapsto \mathsf{PointColoré}\,\{x=1;y=2;c=3\}\,;\ \mathit{tail}\mapsto [\,\mathsf{Point}\,\{x=1;y=2\}\,]\,] \quad \blacksquare$$

Remarque. Notons que le domaine de la fonction $\mathsf{filtre}(e,\pi)$ est nécessairement l'ensemble des variables apparaissant dans π, c'est-à-dire $\mathit{fv}(\pi)$. Par ailleurs, remarquons que la condition syntaxique de bonne formation d'un motif objet, c'est-à-dire $\mathit{fv}(\pi_1)\#\cdots\#\mathit{fv}(\pi_n)$, nous permet, dans la définition de la projection sur un tel motif, de considérer les sous-motifs dans un ordre quelconque. ■

2.4.4 Dispatch dynamique

Étant donnée une liste $\pi_1,\ldots,\pi_k$ de motifs et une expression e, l'opération de DISPATCH DYNAMIQUE consiste à déterminer le motif le plus précis qui filtre l'expression e. Si le dispatch réussit, il produit un indice j entre 1 et k tel que e est filtrée par π_j et que π_j est plus précis que tout motif $\pi_{j'}$ qui filtre e. Le dispatch échoue si aucun motif ne filtre e (non-exhaustivité) ou bien si plusieurs motifs conviennent, mais qu'aucun d'eux n'est plus précis que tous les autres (ambiguïté). Enfin, il faut prévoir le cas où le dispatch est indéterminé parce que l'expression e n'est pas assez réduite. Pour traiter cette difficulté, on demande que le filtrage de e par chacun des motif π_j soit déterminé avant de pouvoir décider si le filtrage réussit ou non.

Plus précisément, le dispatch dynamique est donc une fonction notée $\mathrm{dispatch}(e;\pi_1;\ldots;\pi_k)$ qui prend sa valeur dans l'ensemble $\{\bot,1,\ldots,k,\mathsf{erreur}\}$ et qui est définie par les propriété suivante :

- s'il existe un indice j tel que $\mathsf{filtre}(e,\pi_j)=\bot$, alors $\mathrm{dispatch}(e;\pi_1;\ldots;\pi_k)=\bot$ (dispatch indéterminé) ;
- sinon, s'il existe un unique indice j tel que $\mathsf{filtre}(e,\pi_j)=\mathsf{vrai}$ et tel que $\mathsf{filtre}(e,\pi_{j'})=\mathsf{vrai}$ implique $\pi_j\leqslant\pi_{j'}$, alors $\mathrm{dispatch}(e;\pi_1;\ldots;\pi_k)=j$;
- sinon, le dispatch échoue et on note $\mathrm{dispatch}(e;\pi_1;\ldots;\pi_k)=\mathsf{erreur}$.

Remarque. Notons que nous exigeons que le filtrage de l'argument avec *tous* les motifs soit déterminé avant de pouvoir réduire l'application d'une méthode. Ceci est légèrement trop restrictif dans certains cas. Par exemple, si la méthode ne comporte qu'une unique branche $\pi\Rightarrow e$, alors on pourrait directement utiliser cette branche sans réduire l'argument. Cependant, généraliser cette remarque aux

cas où il y a plusieurs branches ne paraîte pas facile. De plus, si la méthode est formée en agrégeant des branches provenant de différents modules, la sémantique de l'application risquerait de changer quand on rajoute une branche, ce qui pourrait être surprenant pour le programmeur. C'est pourquoi, nou savons préféré adopter dès le départ une sémantique simple et stable, quitte à restreindre un peu le niveau de « paresse ». ■

2.4.5 La règle d'évaluation

L'évaluation de l'application d'une méthode consiste à sélectionner un cas de la méthode par dispatch dynamique de l'argument de la méthode sur la liste ses motifs. On extrait ensuite de l'argument une sous-expression pour chacune des variables apparaissant dans le motif sélectionné. On substitue enfin dans le corps de la méthode chacune de ces variables par la sous-expressions correspondante. La règle d'évaluation prend donc la forme suivante :

(BETA-METH)
$$\frac{j = \text{dispatch}(e; \pi_1; \ldots; \pi_k)}{(\texttt{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\})\, e \longrightarrow [e \downarrow \pi_j]e_j}$$

Dans cette règle, la notation $[e \downarrow \pi_j]e_j$ désigne la substitution simultanée de toutes les variables libres de e_j qui sont aussi libres dans π_j par la sous-expression de e déterminée par l'opération de projection. La définition de la substitution simultanée est donnée à la fin de ce chapitre (section 2.7, page 43).

2.5 Contextes

La définition de l'évaluation ne serait pas complète si on ne définissait pas les contextes d'évaluation. Ceux-ci indiquent les sites possibles d'application des règles de réduction dans une expression. Un site de réduction est dénoté par le symbole [] et les contextes sont engendrés par la grammaire suivante :

$$\begin{aligned} E ::= {} & []\, e \\ & \mid e\, [] \\ & \mid \texttt{let}\ x = []\ \texttt{in}\ e \\ & \mid C\,\{\ell_1 = e_1; \ldots; \ell_i = []; \ldots; \ell_n = e_n\} \end{aligned}$$

L'application d'un contexte à une expression est notée $E[e]$ et consiste à placer e à la place du symbole [] dans E. Le résultat est donc une expression. Le fait que [] dénote un site de réduction est exprimée par la règle d'évaluation suivante :

(CONTEXTE)
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

Remarque. Notons que cette règle est récursive car sa prémisse peut très bien être déduite par (CONTEXTE) elle-même. La forme générale d'une réduction est donc la suivante : si e se réduit en e' par une règle différente de (CONTEXTE), alors on a aussi, pour toute liste finie de contextes :

$$E_1[E_2[\ldots E_n[e]]] \longrightarrow E_1[E_2[\ldots E_n[e']]].$$

En général, les sémantiques à petits pas des langages à la ML utilisent des contextes eux-même récursifs, c'est-à-dire dont la grammaire est de la forme $E ::= [] \mid E[e] \mid e[E] \mid \ldots$. Dans notre présentation, les contextes sont de hauteur 1. La relation de réduction qui en résulte est identique mais cette définition présente un avantage technique car, dans les preuves, on évite une double induction sur les contextes et les dérivations. ■

Contexte. La règle (CONTEXTE) est la dernière de notre sémantique. Notons que cette sémantique est non-déterministe. Une expression donnée peut présenter plusieurs sites de réduction possibles, c'est-à-dire qu'elle peut être de la forme $E_1[E_2[\ldots E_n[e]]]$ pour plusieurs listes de contextes possibles. Par exemple, dans l'expression $e_1\, e_2$, on peut réduire la partie gauche de l'application ou bien la partie droite. Par ailleurs, la règle (CONTEXTE) peut parfois être applicable en même temps que d'autres règles. Par exemple, dans l'expression $(\mathbf{fun}\ x\ \Rightarrow \ldots)\, e$, on peut ou bien réduire l'argument de la fonction en utilisant la règle (CONTEXTE) ou bien appliquer la règle (BETA).

Traditionnellement, la présentation de la sémantique à petits pas des langages à la ML restreint les prémisses des règles d'évaluation et la définition des contextes afin d'imposer une stratégie d'évaluation déterministe. Cela permet de modéliser fidèlement le comportement effectivement mis en œuvre dans une implémentation concrète de l'évaluation. Par exemple, si l'on désire modéliser une stratégie en appel par valeur (*call-by-value*), il convient de restreindre l'application de la règle (BETA) au cas où l'argument de la fonction appliquée est une valeur. Si l'on veut de plus imposer l'ordre d'évaluation gauche-droite, il suffit de restreindre les contextes de la forme $e\, E$ au cas où e est une valeur (fonction ou primitive). Ainsi, l'interprète devra forcément commencer par réduire la partie gauche d'une application jusqu'à obtenir une fonction puis réduire la partie droite jusqu'à obtenir une valeur pour enfin appliquer la règle (BETA). En revanche, si on souhaite modéliser une stratégie en appel par valeur (*call-by-name*), il suffit d'interdire les contextes de la forme $e\, E$, de sorte que l'argument d'une application ne soit jamais évalué.

Nous avons préféré définir une relation d'évaluation non-déterministe pour plusieurs raisons. D'abord la stratégie d'évaluation n'est pas très importante du point de vue du système de types. Si celui-ci nous permet d'affirmer statiquement qu'aucune évaluation d'une expression ne conduit à une erreur de type, ce résultat est indépendant de la stratégie d'évaluation choisie, qui consiste simplement à restreindre les évaluations possibles. Un même système de types s'appliquera donc à toutes les stratégies possibles, en particulier aux stratégies d'appel par valeur et d'appel par nom.

Une deuxième raison est que la présence des méthodes rend la définition d'une stratégie déterministe d'appel par nom non-triviale. En effet, s'il est bien connu que le λ-calcul est essentiellement séquentiel, les multi-méthode introduit un certain degré de parallélisme dans notre sémantique. Concrètement, il ne suffit pas, comme dans le λ-calcul, de réduire systématiquement le redex le plus à gauche pour obtenir une stratégie complète. L'application d'une méthode nécessite en effet de réduire suffisamment son argument pour déterminer le filtrage avec tous les motifs de la méthode, mais il n'existe pas d'ordre d'évaluation simple des différentes parties de l'argument. Considérons par exemple la méthode $\mathbf{meth}\ \{\langle \mathsf{false}, \mathsf{false}\rangle \Rightarrow \mathsf{false}; \_ \Rightarrow \mathsf{true}\}$. Appliquons cette méthode à un argument $\langle e_1, e_2\rangle$. La stratégie qui consiste à évaluer e_1 et e_2 en séquence, de gauche à droite, n'est pas complète, car elle boucle sur l'argument $\langle \bot, \mathsf{true}\rangle$ où $\bot$ désigne une expression qui ne termine pas, par exemple $\mathbf{fix}\ x\ \Rightarrow x$. Or, la sémantique opérationnelle permettrait dans ce cas de brancher directement dans le cas $\_ \Rightarrow \ldots$, parce que le filtrage avec les deux motifs est déjà déterminé. Inversement, si on essaye d'évaluer les arguments dans l'ordre droite-gauche, cette stratégie échoue sur l'entrée $\langle \mathsf{true}, \bot\rangle$. De manière générale, cette méthode n'est pas compilable dans le langage sans dispatch, car sa sémantique (*ou parallèle*) n'est pas une fonction stable [Ber78].

Pour évaluer cette méthode, il est nécessaire d'utiliser un évaluateur parallèle. Essentiellement, il faut lancer deux tâches indépendantes pour évaluer chacun des arguments. Si l'une des tâches termine en déterminant que l'argument calculé est différent de false, alors on doit interrompre l'autre tâche et continuer le calcul dans l'alternative $\_ \Rightarrow \mathsf{true}$. Si les deux tâches terminent avec le résultat false, alors on peut continuer dans $\langle \mathsf{false}, \mathsf{false} \rangle \Rightarrow \mathsf{false}$. Écrire une stratégie d'évaluation complète et déterministe revient alors à écrire un séquenceur multi-tâches et à expliciter l'algorithme utilisé pour le séquencement, celui-ci devant être suffisamment équitable pour que la stratégie soit bien complète. Cela représente une « infrastructure » lourde, ce qui explique qu'une stratégie complète pour filtrage parallèle a rarement été envisagée en pratique. On peut toutefois mentionner une étude dans le cadre de Concurrent Haskell [WM02] et remarquer que le langage Jazz, naturellement multi-tâches, pourrait se prêter à une telle implémentation complète. Il convient aussi de citer ici les travaux relatifs à la compilation du filtrage paresseux vers un langage séquentiel (voir par exemple [Mar94]) qui constitue une forme particulière de stratégie incomplète.

Il apparaît donc clairement que le détail de la stratégie d'évaluation déterministe doit rester au niveau de l'implémentation. C'est pourquoi nous étudierons le système de types en considérant une relation de réduction non-déterministe, ce qui permet d'appliquer les résultats obtenus à *toutes* les stratégies possibles, complètes ou non. ■

2.6 Erreurs de type

L'évaluation d'une expression peut aboutir à une situation manifestement erronée comme l'application de l'addition à des booléens ou l'utilisation d'un entier en position de fonction. De telles expressions sont les ERREURS DE TYPE. Plus précisément, sont considérées comme des erreurs de type les situations suivantes :

Erreur de type	Exemple
application d'une primitive à une valeur non construite avec la classe $\texttt{Tuple}_n$ où n est l'arité de la primitive	$+\,(\textbf{fun}\ x\ \Rightarrow x)$ ou $+\,\langle 1\rangle$
application d'une primitive à un tuple de valeurs qui ne sont pas toutes des constantes	$1+(\textbf{fun}\ x\ \Rightarrow x)$
application d'une primitive à un tuple de constantes qui n'est pas dans le domaine de la primitive	$1+\mathsf{true}$
application d'une valeur qui n'est n'est pas fonctionnelle, c'est-à-dire ni une fonction, ni une méthode, ni une primitive	$1\ (2)$
application d'un accesseur de champs à une valeur qui n'est pas un objet	$\mathsf{Point}\cdot x\,(\textbf{fun}\ x\ \Rightarrow x)$
application d'un accesseur de champs à un objet qui n'est pas de la bonne classe	$\mathsf{PointColoré}\cdot c\,(\mathsf{Point}\,\{\ldots\})$
application d'une méthode à un argument filtré par aucun motif (non-couverture)	$(\texttt{meth}$ $\langle\mathsf{Point},\mathsf{PointColoré}\rangle \Rightarrow \ldots$ $\langle\mathsf{PointColoré},\mathsf{Point}\rangle \Rightarrow \ldots)$ $\langle\mathsf{Point}\,\{\ldots\},\mathsf{Point}\,\{\ldots\}\rangle$
application d'une méthode à un argument filtré par plusieurs motifs (ambiguïté)	$(\texttt{meth}$ $\langle\mathsf{Point},\mathsf{PointColoré}\rangle \Rightarrow \ldots$ $\langle\mathsf{PointColoré},\mathsf{Point}\rangle \Rightarrow \ldots)$ $\langle\mathsf{PointColoré}\,\{\ldots\},\mathsf{PointColoré}\,\{\ldots\}\rangle$

En outre, chaque fois qu'une erreur de type apparaît dans une expression à l'emplacement d'un site de réduction possible, on considère que l'expression toute entière est aussi une erreur de type. En d'autres termes, si e est une erreur de type, alors $E[e]$ est aussi une erreur.

Nous noterons $e \longrightarrow \mathsf{erreur}$ quand l'expression e est une erreur de type et $e \longrightarrow^\star \mathsf{erreur}$ quand e se réduit en un nombre fini d'étapes en e' qui est une erreur de type. Ce dernier prédicat sépare les expressions en deux catégories : si $e \longrightarrow^\star \mathsf{erreur}$, on dit que e est OPÉRATIONNELLEMENT MAL TYPÉE, sinon qu'elle est OPÉRATIONNELLEMENT BIEN TYPÉE. L'un des rôles d'un système de types est d'éliminer statiquement toutes les expressions opérationnellement mal typées sans éliminer trop d'expressions bien typées.

Remarque. En présence d'un système de types, un interprète peut donc considérer que l'expression à calculer ne sera jamais une erreur de type, ce qui permet en général d'optimiser l'implémentation. Le comportement d'un interprète ainsi optimisé peut alors être arbitraire en présence d'une erreur de type : corruption mémoire, plantage, etc. C'est pourquoi, le système de types est essentiel dès lors que l'interprète n'est pas défensif.

Notons que dans une sémantique déterministe, la définition des erreurs de type est souvent présentée d'une façon implicite. Les valeurs représentant les résultats irréductibles des calculs et, à chaque instant, une seule réduction étant possible, les expressions irréductibles qui ne sont pas des valeurs dénotent un problème grave d'exécution et sont considérées comme des erreurs de type.

Cette définition présente l'inconvénient de ne pas expliciter exactement les propriétés qu'on peut utiliser pour optimiser l'interprète. Plus grave, elle ne s'étend pas commodémment à une sémantique non-déterministe. En effet, il se peut très bien qu'une expression contenant deux sites de réduction possibles soit réductible sur le premier site et présente une erreur de type faisant planter l'interprète sur le second. Dans ce cas, l'expression est réductible mais il est important de considérer que c'est une erreur de type puisqu'elle provoque un problème dans l'interprète. ■

2.7 Récapitulatif

Nous donnons ici la définition formelle du prédicat d'évaluation et de toutes les relations nécessaires à sa définition.

Valeurs

L'ensemble des valeurs, noté Valeurs, est engendré par la grammaire suivante :

$$\begin{aligned} V ::= & \; a \\ & \mid C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\} \\ & \mid \textbf{fun}\; x \;\Rightarrow e \\ & \mid \textbf{meth}\; \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\} \\ & \mid f \\ & \mid C\cdot\ell \end{aligned}$$

De plus, toute valeur V doit être telle que $fv(V) = \varnothing$.

Nœud de tête d'une valeur

$$\begin{aligned} \text{tête}(a) &\triangleq a \\ \text{tête}(C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}) &\triangleq C \\ \text{tête}(\textbf{fun}\; x \;\Rightarrow e) &\triangleq \textbf{fun} \\ \text{tête}(\textbf{meth}\; \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}) &\triangleq \textbf{fun} \\ \text{tête}(f) &\triangleq \textbf{fun} \\ \text{tête}(C\cdot\ell) &\triangleq \textbf{fun} \end{aligned}$$

Filtrage

$$\begin{aligned} \text{filtre}(e, \_) &\triangleq \text{vrai} \\ \text{filtre}(e, \pi\ \textbf{as}\ x) &\triangleq \text{filtre}(e, \pi) \\ \text{filtre}(e, \pi) &\triangleq \bot \quad \text{si} \begin{cases} e \notin \text{Valeurs} \\ \pi \neq \_ \\ \pi \neq \pi'\ \textbf{as}\ x \end{cases} \\ \text{filtre}(a, \varpi) &\triangleq \text{vrai} \quad \text{si } a \in \varpi \\ \text{filtre}(C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}, & \\ \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}) &\triangleq \text{filtre}(e_1, \pi_1) \wedge \cdots \wedge \text{filtre}(e_n, \pi_n) \end{aligned}$$

$$
\begin{array}{rcll}
\mathsf{filtre}(C'\,\{\ell_1 = e_1; \ldots; \ell_n = e_n; \ldots\}\,, & & & \\
\qquad C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}) & \triangleq & \mathsf{filtre}(e_1, \pi_1) \wedge \cdots \wedge \mathsf{filtre}(e_n, \pi_n) & \text{si } C' \sqsubseteq C \\
\mathsf{filtre}(e, \pi) & \triangleq & \mathsf{faux} & \text{dans tous les autres cas}
\end{array}
$$

Dans ces définitions, $\wedge$ désigne la conjonction parallèle qui vaut vrai si tous les arguments valent vrai, faux si l'un des argument vaut faux et $\bot$ dans les autres cas.

Pré-ordre de précision

Il est défini par les règles d'inférence suivantes.

$$\frac{}{\pi \leqslant \pi} \qquad \frac{\pi \leqslant \pi' \qquad \pi' \leqslant \pi''}{\pi \leqslant \pi''} \qquad \frac{}{\pi \leqslant (\pi\ \mathbf{as}\ x)} \qquad \frac{}{(\pi\ \mathbf{as}\ x) \leqslant \pi}$$

$$\frac{\dfrac{}{\pi \leqslant \_} \qquad \dfrac{\varpi_1 \subseteq \varpi_2}{\varpi_1 \leqslant \varpi_2} \qquad C' \sqsubseteq C}{C'\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n; \ldots; \ell_{n'} = \pi_{n'}\} \leqslant C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}}$$

$$\frac{\dfrac{}{\#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} \leqslant C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}} \qquad \pi'_1 \leqslant \pi_1 \quad \ldots \quad \pi'_n \leqslant \pi_n}{C\left\{\ell_1 = \pi'_1; \ldots; \ell_n = \pi'_n\right\} \leqslant C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}} \qquad \pi'_1 \leqslant \pi_1 \quad \ldots \quad \pi'_n \leqslant \pi_n$$
$$\frac{}{\#C\left\{\ell_1 = \pi'_1; \ldots; \ell_n = \pi'_n\right\} \leqslant \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}}$$

Projection

$$
\begin{array}{rcll}
e \downarrow \_ & \triangleq & \varnothing & \\
e \downarrow \varpi & \triangleq & \varnothing & \\
e \downarrow (\pi\ \mathbf{as}\ x) & \triangleq & e \downarrow \pi \oplus [x \mapsto e] & \\
(C'\,\{\ell_1 = e_1; \ldots; \ell_n = e_n; \ldots\}) \downarrow (C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}) & \triangleq & e_1 \downarrow \pi_1 \oplus \cdots \oplus e_n \downarrow \pi_n & \\
(C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}) \downarrow (\#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}) & \triangleq & e_1 \downarrow \pi_1 \oplus \cdots \oplus e_n \downarrow \pi_n & (\text{si } C' \sqsubseteq C)
\end{array}
$$

Fonction de dispatch

$$\text{dispatch}(e; \pi_1; \ldots; \pi_k) \quad \triangleq \quad \begin{cases} \bot & \text{ssi } \exists j \in [1,k], \mathsf{filtre}(e, \pi_j) = \bot \\ j & \text{ssi } \begin{cases} \mathsf{filtre}(e, \pi_j) = \mathsf{vrai} \\ \forall l \in [1,k], \quad \mathsf{filtre}(e, \pi_l) \neq \bot \\ \qquad\qquad \wedge \mathsf{filtre}(e, \pi_l) = \mathsf{vrai} \Rightarrow \pi_j \leqslant \pi_l \\ \qquad\qquad \wedge \big(\mathsf{filtre}(e, \pi_l) = \mathsf{vrai} \wedge \pi_l \leqslant \pi_j\big) \Rightarrow l = j \end{cases} \\ \mathsf{erreur} & \text{sinon} \end{cases}$$

Substitutions

Une SUBSTITUTION S est une fonction partielle à domaine fini des variables dans les expressions. Un RENOMMAGE R est une substitution dont le codomaine est composé uniquement de variables. Pour tous ensembles finis de variables X_1 et X_2 on suppose donné un renommage $\mathcal{R}(X_1, X_2)$ tel que :

- $\mathcal{R}(X_1, X_2)$ est injective ;
- X_1 est le domaine de $\mathcal{R}(X_1, X_2)$;
- le codomaine de $\mathcal{R}(X_1, X_2)$ est disjoint de X_2 ;
- pour toute variable $x \in X_1 \setminus X_2$, $\mathcal{R}(X_1, X_2)(x) = x$.

Cette fonction existe bien car l'ensemble Vars est supposé infini.

Une substitution est étendue à l'ensemble des expressions de la façon suivante :

$$\begin{aligned}
[S]x &\triangleq \begin{cases} S(x) & \text{si } x \in \mathrm{dom}(S) \\ x & \text{sinon} \end{cases} \\
[S](e\,e') &\triangleq ([S]e)\ \big([S]e'\big) \\
[S](\mathbf{fun}\ x\ \Rightarrow e) &\triangleq \mathbf{fun}\ [R]x\ \Rightarrow [S \oplus R]e \quad (1) \\
[S](\mathbf{let}\ x = e'\ \mathbf{in}\ e) &\triangleq \mathbf{let}\ [R]x = [S]e'\ \mathbf{in}\ [S \oplus R]e \quad (1) \\
[S](\mathbf{fix}\ x\ \Rightarrow e) &\triangleq \mathbf{fix}\ [R]x\ \Rightarrow [S \oplus R]e \quad (1) \\
[S]C\cdot\ell &\triangleq C\cdot\ell \\
[S]C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\} &\triangleq C\,\{\ell_1 = [S]e_1; \ldots; \ell_n = [S]e_n\} \\
[S]\,f &\triangleq f \\
[S]a &\triangleq a \\
[S](\mathbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}) &\triangleq \mathbf{meth}\ \{[S](\pi_1 \Rightarrow e_1); \ldots; [S](\pi_k \Rightarrow e_k)\}
\end{aligned}$$

où :

$$[S](\pi \Rightarrow e) \quad \triangleq \quad [R]\pi \Rightarrow [S']e \quad (2)$$

et :

$$\begin{array}{rcl}
[R]_ & \triangleq & _ \\
[R]\varpi & \triangleq & \varpi \\
[R](\pi \ \mathbf{as}\ x) & \triangleq & [R]\pi \ \mathbf{as}\ [R]x \\
[R]C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} & \triangleq & C\,\{\ell_1 = [R]\pi_1; \ldots; \ell_n = [R]\pi_n\} \\
[R]\#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} & \triangleq & \#C\,\{\ell_1 = [R]\pi_1; \ldots; \ell_n = [R]\pi_n\}
\end{array}$$

Note :

(1) $R = \mathcal{R}(\{x\}, fv(S \restriction V) \cup V)$ où $V = fv(e) \setminus \{x\}$;

(2) $R = \mathcal{R}(fv(\pi), fv(S \restriction V) \cup V)$ où $V = fv(\pi \Rightarrow e)$;

avec, pour toute substitution S :

$$fv(S) \quad \triangleq \quad \bigcup_{x \in \mathrm{dom}(S)} fv(S(x))$$

Remarque. On montre facilement que l'opération de substitution préserve la bonne formation syntaxique des expressions.

Exemple. Soit e l'expression suivante :

$$e = \mathbf{meth}\ \{\langle \_\ \mathbf{as}\ x, \_\ \mathbf{as}\ y\rangle \Rightarrow x + y + z + t\}$$

et S la substitution :

$$S = [x \mapsto u, z \mapsto x + u, t \mapsto u]$$

L'expression $[S]e$ vaut alors :

$$\mathbf{meth}\ \left\{\langle \_\ \mathbf{as}\ x', \_\ \mathbf{as}\ y\rangle \Rightarrow x' + y + (x + u) + u\right\}$$

avec $x' = \mathcal{R}(\{x, y\}, \{x, u\})(x)$. ■

Contextes

Grammaire :

$$\begin{array}{rcll}
E & ::= & []\,e & \\
 & | & e\,[] & \\
 & | & \mathbf{let}\ x = []\ \mathbf{in}\ e & \\
 & | & C\,\{\ell_1 = e_1; \ldots; \ell_i = []; \ldots; \ell_n = e_n\} & C \in \mathsf{ClassesConcrètes} \quad (1)
\end{array}$$

Notes :

(1) Les labels $\ell_1, \ldots, \ell_n$ doivent être distincts deux à deux, représenter les champs de la classe C et leur ordre d'apparition n'est pas signifiant ;

Application d'un contexte à une expression :

$$\begin{aligned}
([]\,e)[e_0] &\triangleq e_0\,e \\
(e\,[])[e_0] &\triangleq e\,e_0 \\
(\mathbf{let}\ x = []\ \mathbf{in}\ e)[e_0] &\triangleq \mathbf{let}\ x = e_0\ \mathbf{in}\ e \\
(C\,\{\ell_1 = e_1; \ldots; \ell_i = []; \ldots; \ell_n = e_n\})[e_0] &\triangleq C\,\{\ell_1 = e_1; \ldots; \ell_i = e_0; \ldots; \ell_n = e_n\}
\end{aligned}$$

Notons que la contrainte de bonne formation sur les contextes d'objets font que l'application $E[e]$ est toujours une expression syntaxiquement bien formé dès que e est aussi bien formée.

Règles d'évaluation

(CONTEXTE)
$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

(BETA)
$$(\mathbf{fun}\ x\ \Rightarrow e_1)\,e_2 \longrightarrow [x \mapsto e_2]e_1$$

(BETA-LET)
$$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \longrightarrow [x \mapsto e_1]e_2$$

(BETA-FIX)
$$(\mathbf{fix}\ x\ \Rightarrow e) \longrightarrow [x \mapsto (\mathbf{fix}\ x\ \Rightarrow e)]e$$

(BETA-OBJ)
$$C{\cdot}\ell_i\,(C'\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}) \longrightarrow e_i$$

(BETA-PRIM)
$$\frac{f \in \mathsf{PrimOps}_n \qquad a' = f(a_1, \ldots, a_n)}{f\,\langle a_1, \ldots, a_n\rangle \longrightarrow a'}$$

(BETA-METH)
$$\frac{j = \mathrm{dispatch}(e; \pi_1; \ldots; \pi_k)}{(\mathbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\})\,e \longrightarrow [e \downarrow \pi_j]e_j}$$

Erreurs de type

(ERREUR-PRIM-1)
$$\frac{f \in \mathsf{PrimOps}_n \qquad \text{tête}(V) \neq \mathbf{Tuple}_n}{f\,V \longrightarrow \mathsf{erreur}}$$

(ERREUR-PRIM-2)
$$\frac{V_i \notin \mathsf{PrimVals}}{f\,\langle V_1, \ldots, V_n\rangle \longrightarrow \mathsf{erreur}}$$

(ERREUR-PRIM-3)
$$\frac{(V_1, \ldots, V_n) \notin \mathrm{dom}(f)}{f\ \langle V_1, \ldots, V_n \rangle \longrightarrow \text{erreur}}$$

(ERREUR-FUN)
$$\frac{\text{tête}(V) \neq \textbf{fun}}{V\ e \longrightarrow \text{erreur}}$$

(ERREUR-GET-1)
$$\frac{\text{tête}(V) \notin \text{ClassesConcrètes}}{C \cdot \ell\ V \longrightarrow \text{erreur}}$$

(ERREUR-GET-2)
$$\frac{C' = \text{tête}(V) \qquad C' \not\sqsubseteq C}{C \cdot \ell\ V \longrightarrow \text{erreur}}$$

(ERREUR-MÉTHODE)
$$\frac{\text{dispatch}(e; \pi_1; \ldots; \pi_k) = \text{erreur}}{(\textbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\})\ e \longrightarrow \text{erreur}}$$

(ERREUR-CONTEXTE)
$$\frac{e \longrightarrow \text{erreur}}{E[e] \longrightarrow \text{erreur}}$$

Chapitre 3

Typage algébrique

Nous allons maintenant introduire un système de types pour le langage du chapitre 1. C'est-à-dire que nous allons séparer les expressions en deux catégories : celles qui sont « bien typées » et celles qui ne le sont pas. La propriété fondamentale qu'un système de types doit garantir est que l'évaluation d'une expression bien typée ne provoque aucune des erreurs de types mentionnées en 2.6. En rejetant les expressions mal typées, on peut ainsi attraper avant l'exécution toute une classe d'erreurs de programmation. C'est l'intérêt majeur d'un système de type.

Dans ce chapitre, nous nous attachons à fournir un système qui soit suffisamment expressif (il accepte assez de programmes) et correct (il rejette bien les erreurs de type). En revanche, nous ne nous occupons pas des autres aspects. L'intérêt de ce système de types est de fournir une « boîte à outils » pour montrer la correction de systèmes plus pratiques qui seront introduits plus loin.

L'objectif étant simplement de montrer la correction, nous ne préciserons pas complètement ce qu'est un type dans ce système, mais nous nous contenterons d'en axiomatiser les propriétés requises pour assurer la correction. C'est pourquoi, le système de types présenté dans ce chapitre est qualifié de « algébrique » et la structure mathématique ainsi axiomatisée sera appelée algèbre de types.

3.1 Principe

Un système de types consiste en une définition d'un prédicat, appelé JUGEMENT DE TYPAGE qui relie une expression à un type. Comme cette expression peut contenir des variables libres, le jugement de typage fait aussi intervenir un environnement de typage qui liste les hypothèses qu'on fait sur ces variables libres. Le jugement de typage s'écrit donc $G \vdash e : T$ et se lit « dans l'environnement G, l'expression e possède le type T ».

S'il existe un type T tel que $G \vdash e : T$, alors l'expression e est dite BIEN TYPÉE dans l'environnement G. Elle est MAL TYPÉE dans le cas contraire.

La correction du système de types provient de deux ingrédients :

1. D'abord, une expression bien typée est opérationnellement bien typée. Ainsi, le système de types élimine les erreurs de type immédiates ;
2. Ensuite, une expression bien typée reste bien typée quand on la réduit par un pas de calcul. On élimine ainsi les erreurs de type qui peuvent survenir plus loin dans le calcul.

3.2 Types monomorphes

On suppose donné un ensemble $\mathcal{A}$ de TYPES MONOMORPHES ou MONOTYPES. Les éléments de cet ensemble sont notés avec les lettres t ou u (éventuellement avec un prime ou un indice). Le contenu précis de cet ensemble est laissé ouvert car ce qui nous intéresse, ce sont simplement certaines propriétés sur les opérations définies sur $\mathcal{A}$ que nous allons axiomatiser dans cette section. L'approche est donc algébrique et l'ensemble $\mathcal{A}$ avec ses opérations est appelé ALGÈBRE DE TYPES.

Exemple. La raison pour laquelle nous ne fixons pas l'algèbre de types est que diverses algèbres sont envisageables. Il est cependant utile d'introduire dès maintenant un exemple concret d'algèbre pour illustrer ses axiomes et les règles de typage.

Pour fixer les idées, nous considérerons un exemple simple. Les monotypes de cette algèbre sont les termes finis et clos (sans variables) construits sur l'alphabet constitué des noms de classes, du symbole $\rightarrow$ et des types primitifs int, rat et bool :

$$
\begin{aligned}
t ::= {} & \mathsf{int} \mid \mathsf{rat} \mid \mathsf{bool} \\
& \mid t \rightarrow t \\
& \mid \mathsf{ObjetColoré} \mid \mathsf{Point} \mid \mathsf{PointColoré} \mid \mathsf{Point3D} \mid \mathsf{Point3DColoré} \mid \ldots \\
& \mid \mathsf{List}\langle t \rangle \mid \mathsf{Cons}\langle t \rangle \mid \mathsf{Nil}\langle t \rangle \mid \ldots
\end{aligned}
$$

Voici quelques exemples de monotypes dans cette algèbre :

$$
\begin{array}{c}
\mathsf{bool} \quad \mathsf{Point} \\
\mathsf{Cons}\langle \mathsf{Point} \rangle \quad \mathsf{List}\langle \mathsf{bool} \rangle \\
\mathsf{Point} \rightarrow \mathsf{int} \\
\mathsf{bool} \rightarrow \mathsf{List}\langle \mathsf{bool} \rangle \rightarrow \mathsf{Cons}\langle \mathsf{bool} \rangle \\
\vdots
\end{array}
$$

Ainsi, à chaque classe C est associé un monotype ou un constructeur de types si la classe est considérée comme paramétrée. Par exemple, les classes Point ou PointColoré ne sont pas des classes paramétrées et constituent donc directement des monotypes. En revanche, les classes List, Cons et Nil sont paramétrées par le type des éléments de la liste et sont donc des constructeurs qui prennent un type en paramètre. Les types des valeurs primitives (entiers, booléens) sont des constantes non paramétrées (int, bool, etc.). Enfin, la flêche $\rightarrow$ est un opérateur de types qui ne correspond pas à une classe. ■

3.3 Types polymorphes

Dans les systèmes de type à la ML, le polymorphisme est traditionnellement exprimé à l'aide de schémas de types, un schéma de types étant une description en *compréhension* d'un ensemble de types monomorphes. Typiquement, un schéma de types en ML est formé d'un terme contenant des variables libres et décrit l'ensemble des instances que peut prendre ce terme quand les variables sont substituées par n'importe quel terme clos.

Dans ce chapitre, nous nous intéressons seulement à la correction du système, sans vouloir effectivement calculer sur les types. Le polymorphisme peut alors être sans syntaxe particulière : un type polymorphe est simplement un ensemble de monotypes donné en *extension*, sans qu'il soit décrit par un terme avec variables. L'avantage de cette approche est qu'on sépare bien ce qui est nécessaire pour assurer la correction du typage de l'aspect calcul sur les schémas de types, qui fera l'objet des chapitres suivants.

Un TYPE POLYMORPHE, encore appelé POLYTYPE, est donc défini comme un ensemble quelconque *non vide* de monotypes. On utilisera la lettre T pour désigner les polytypes. Un polytype singleton de la forme $\{t\}$ est identifié au monotype t.

Exemple. Voici quelques exemples de polytypes dans l'algèbre fermée simple :

$$\{\mathsf{Point} \to \mathsf{int}, \mathsf{PointColoré} \to \mathsf{int}, \mathsf{Point3D} \to \mathsf{int}, \mathsf{Point3DColoré} \to \mathsf{int}\}$$
$$\{\mathsf{Cons}\langle t\rangle \to t \mid t \in \mathcal{A}\}$$
$$\vdots$$

■

Cette définition des types polymorphes est étroitement liée aux deux règles de typage concernant le polymorphisme : l'instantiation et la généralisation :

(INST) $$\frac{G \vdash e : T \qquad t \in T}{G \vdash e : t}$$

(GEN) $$\frac{\forall t \in T, G \vdash e : t}{G \vdash e : T}$$

La définition de la généralisation est extensionnelle : pour pouvoir donner un type polymorphe T à une expression e, il suffit qu'on puisse lui donner chacun des éléments monomorphes de T.

Remarquons que le traitement du polymorphisme peut être fait indépendamment de toutes les autres règles de typage. En ce sens, ce polymorphisme est superficiel : on se contente de plaquer la généralisation et l'instantiation par-dessus les « vraies » règles qui sont monomorphes. Cela correspond au fait que les quantificateurs sont en tête des schémas de types.

Remarque. La règle (GEN) pose un problème technique car elle contient un nombre infini de prémisses si l'ensemble T est infini. La présentation du système de types et les preuves associées doivent donc faire appel à une théorie mathématique plus élaborée que celle des arbres finis et des nombres naturels. En particulier, on a besoin d'un moyen de raisonner par induction sur les dérivations de typage. Pour résoudre ce problème de fondements, plusieurs solutions sont possibles. On peut se placer dans le cadre de la théorie des ensembles et utiliser la récursion transfinie, c'est-à-dire considérer que la taille d'une dérivation est un nombre ordinal plutôt qu'un nombre entier. On peut aussi se placer dans le cadre de la théorie des constructions inductives pour profiter de ses schémas d'induction généralisés. Le résultat de toutes ces théories est de permettre les raisonnement par induction sur les dérivations de typage exactement comme si elles étaient finies. C'est pourquoi, la théorie choisie importe peu et nous considèrerons les inductions possibles « comme d'habitude » (cf. section 4.6, page 76 pour plus de précisions théoriques).

3.4 Environnements de typage

Un ENVIRONNEMENT DE TYPAGE est une fonction partielle à domaine fini des variables dans les polytypes. C'est une liste d'assertions de la forme $x : T$ qu'on note :

$$G ::= [x_1 \colon T_1, \ldots, x_n \colon T_n]$$

L'environnement vide est noté $\varnothing$. L'extension d'un environnement G par une assertion de la forme $x : T$ est notée $G[x : T]$ et vaut $G \oplus x \mapsto T$.

Pour typer une variable, il suffit d'extraire l'assertion correspondante dans l'environnement. D'où la règle de typage :

$$\text{(VAR)} \qquad G \vdash x \colon G(x)$$

3.5 Typage de `let` et de `fix`

Le typage des expressions **`let`** est assuré par la règle suivante :

$$\text{(LET)} \qquad \frac{G \vdash e_1 \colon T_1 \qquad G[x \colon T_1] \vdash e_2 \colon T_2}{G \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \colon T_2}$$

La correction vis-à-vis de la règle d'évaluation découle du fait que la relation $\vdash$ satisfait au *lemme de substitution* (voir démonstration section 4.4, page 71) : si e_2 a le type T_2 dans l'environnement G et si e_1 a le type T_1 dans l'environnement étendu $G[x : T_2]$, alors la substitution $[x \mapsto e_1]e_2$ a aussi le type T_1 dans l'environnement G.

Ce lemme est caractéristique des systèmes de typage compositionnels où, pour typer une expression complexe formée de plusieurs sous-expressions, la seule information requise concernant ces sous-expressions est leur type. Par exemple, pour typer e_2 dans l'expression **`let`** $x = e_1$ **`in`** e_2, il n'est pas nécessaire de connaître exactement e_1, mais seulement son type.

La règle de typage des expressions **`fix`** ressemble beaucoup à celle du **`let`** :

$$\text{(FIX)} \qquad \frac{G[x \colon T] \vdash e \colon T}{G \vdash \texttt{fix}\ x\ \Rightarrow e \colon T}$$

Remarquons que cette règle autorise la récursion polymorphe.

3.6 Sous-typage

Pour refléter le sous-classement au niveau des types, toute algèbre $\mathcal{A}$ doit venir avec une relation d'ordre qui permet de comparer l'information qu'apportent deux types. Cette relation, notée $\leqslant$, est appelée SOUS-TYPAGE.

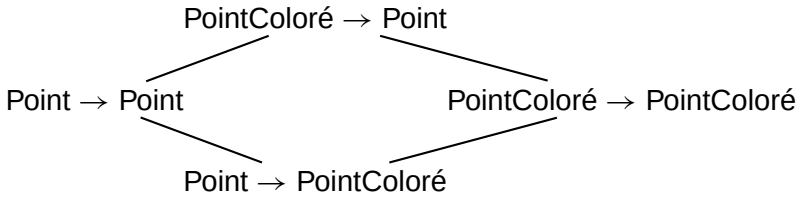
AXIOME (SOUS-TYPAGE-ORDRE).
La relation binaire $\leqslant$ est un ordre partiel sur $\mathcal{A}$, c'est-à-dire qu'elle est réflexive, transitive et antisymétrique[1]

Quand $t \leqslant t'$, on dit que t est un SOUS-TYPE de t', c'est-à-dire que t apporte plus d'information que t'. On dit aussi que t' est un SUPER-TYPE de t ou qu'il est moins précis que t.

La règle de typage suivante, dite de « subsomption », indique qu'on peut toujours accepter de perdre de l'information sur le typage d'une expression :

(SUB)
$$\frac{G \vdash e : t \qquad t \leqslant t'}{G \vdash e : t'}$$

Exemple. Dans notre exemple simple d'algèbre, la relation de sous-typage entre deux monotypes est définie par comparaison structurelle des deux termes. Les types primitifs sont ordonnés selon les valeurs qu'ils contiennent. Par exemple, on a $\mathsf{int} \leqslant \mathsf{rat}$. Pour les classes non-paramétrées, comme Point, la relation de sous-typage se confond avec celle de sous-classement : on a $C \leqslant C'$ chaque fois que $C \sqsubseteq C'$. Par exemple, on a $\mathsf{PointColoré} \leqslant \mathsf{Point}$. Pour les classes paramétrées, la comparaison de fait sur le constructeur de type, mais aussi sur les paramètres. Par exemple, on a : $\mathsf{Cons}\langle\mathsf{PointColoré}\rangle \leqslant \mathsf{List}\langle\mathsf{PointColoré}\rangle$ parce que $\mathsf{Cons} \sqsubseteq \mathsf{List}$. On a aussi $\mathsf{List}\langle\mathsf{PointColoré}\rangle \leqslant \mathsf{List}\langle\mathsf{Point}\rangle$ parce que $\mathsf{PointColoré} \leqslant \mathsf{Point}$. On voit donc que List se comporte comme un opérateur *covariant* (monotone). D'autres variances sont nécessaires. Par exemple, l'opérateur $\rightarrow$ qui admet deux paramètres de type, doit être contravariant relativement au premier paramètre et covariant relativement au deuxième. Ainsi, on a les relations de sous-typage suivantes :



La raison de cette variance de la flèche est plus générale que notre exemple d'algèbre. Cette contrainte s'applique en fait à toutes les algèbres de types. Elle est nécessaire pour assurer la sûreté du typage de l'application, comme nous allons maintenant l'expliquer. ■

3.7 Typage des fonctions et de l'application

Toute structure $\mathcal{A}$ doit venir avec un opérateur binaire $\rightarrow$ pour construire les types fonctionnels. Le type d'une fonction acceptant un argument de type t et renvoyant un résultat de type u est représenté par l'élément $t \rightarrow u$.

[1] En fait, tous les résultats de cette thèse sont préservé si la relation de sous-typage n'est qu'un préodre, c'est-à-dire si elle n'est pas supposée antisymétrique. Il ne nous a cependant pas paru utile d'introduire ce degré de généralité.

Les règles de typage des fonctions et de l'application de fonction sont les suivantes :

$$\text{(FUN)} \qquad \frac{G[x : t'] \vdash e : t}{G \vdash \mathbf{fun}\ x\ \Rightarrow e : t' \to t}$$

$$\text{(APP)} \qquad \frac{G \vdash e_1 : t_2 \to t \qquad G \vdash e_2 : t_2}{G \vdash e_1\, e_2 : t}$$

L'interprétation de la relation de sous-typage en terme de quantité d'information disponible induit immédiatement des contraintes sur le sous-typage des types fonctionnels :

- si $u \leqslant u'$, alors on connaît plus d'information sur le résultat des fonctions de type $t \to u$ que sur les fonctions de type $t \to u'$. Donc $t \to u$ est plus précis que $t \to u'$;
- si $t' \leqslant t$, toute fonction de type $t \to u$ doit accepter au moins les arguments de type t', c'est-à-dire être de type $t' \to u$, donc $t \to u$ est un sous-type de $t' \to u$.

Notons que ces deux propriétés ne sont pas absolument nécessaires à la correction du typage. Elles sont simplement utiles pour être cohérent avec l'interprétation « sous-typage = quantité d'information » et elles sont respectées par toutes les algèbres de types connues.

En revanche, les propriétés réciproques sont réellement indispensables pour la correction. Supposons que $t \to u \leqslant t' \to u'$. Par la règle de subsomption, toute fonction de type $t \to u$ peut être considérée comme ayant le type $t' \to u'$. On peut donc lui passer un argument de type t' et celui-ci doit pouvoir être traité comme ayant le type t a l'intérieur de la fonction. Pour être sûr que cela ne pose pas de problème, on demande que t' soit un sous-type de t. Par ailleurs, le résultat de la fonction est de type u, mais il peut être utilisé avec le type u', ce qui n'est sûr que si u est un sous-type de u'.

Au final, nous demandons que toute algèbre de types vérifie l'axiome suivant :

AXIOME (FLÈCHE-VARIANCE).

$$\forall t t' u u',\ t \to u \leqslant t' \to u' \iff t' \leqslant t \wedge u \leqslant u'$$

Remarque. Les règles de typage des fonctions et de l'application sont *monomorphes*. Grâce à la règle de généralisation (GEN), une application donnée peut bien être associée à un type polymorphe, mais pour chacune de ses instances monomorphes, il faut qu'on puisse appliquer la règle monomorphe (APP). Cela implique qu'une fonction peut bien être polymorphe, c'est-à-dire être appliquée à toutes sortes d'arguments, mais une fois qu'elle est appliquée, son argument est monomorphe.

Cette restriction est due au traitement superficiel et prédicatif du polymorphisme. Pour avoir des arguments polymorphes, il faudrait disposer d'un opérateur flèche qui travaille lui-même sur des types polymorphes. Au niveau des schémas de types, on aurait alors des quantificateurs sous la flèche.

3.8 Typage des objets

3.8.1 L'abstraction d'une classe

Le typage des objets requiert une abstraction des classes dans les types. On supposera que dans toute algèbre de types, chaque classe C est associée à un ensemble de monotypes noté $\overline{\#C}$ qui est l'ensemble des types possibles des objets de cette classe. Pour assurer la sûreté du typage, il faut imposer l'axiome suivant :

AXIOME (CLASSES COVARIANTES).
Soient C et C' deux classes. Soient t et t' deux monotypes tels que $t \in \overline{\#C}$ et $t' \in \overline{\#C'}$. Si $t' \leqslant t$, alors $C' \sqsubseteq C$.

Exemple. Dans l'algèbre simple, $\overline{\#C}$ est simplement l'ensemble des termes construits avec C. Par exemple, $\overline{\#\mathsf{List}}$ est l'ensemble des types de la forme $\mathsf{List}\langle t\rangle$, $\overline{\#\mathsf{Point}}$ vaut $\{\mathsf{Point}\}$, etc. ■

L'axiome (CLASSES-COVARIANTES) est indispensable pour assurer la sûreté du typage. En effet, si le type t' d'un objet e construit avec la classe C' est un sous-type de t construit avec C, cela signifie que, par la règle de subsomption, on peut accéder à e avec un accesseur de la classe C. Pour éviter l'erreur de type (ERREUR-GET-2), il faut donc imposer que C' soit une sous-classe de C.

3.8.2 L'abstraction des champs

Pour chaque classe C et chaque champs $\ell \in \mathsf{Champs}_C$, on supposera donnée une abstraction de l'accès au champs ℓ de la classe C. Cette abstraction, notée $\overline{C\cdot\ell}$, est une fonction de l'ensemble $\overline{\#C}$ vers les monotypes.

Pour assurer la sûreté du typage, il est nécessaire de supposer que cette fonction est monotone. Considérons en effet un objet e de classe C' et de type $t' \in \overline{\#C'}$. Supposons également que t' soit un sous-type de t élément de $\overline{\#C}$. Par l'axiome (CLASSES-COVARIANTES), on sait que C' est une sous-classe de C. Fixons maintenant un champ ℓ_i de la classe C et notons e_i le champ correspondant dans l'objet e. En utilisant l'accesseur $C'\cdot\ell_i$, on donne le type $t'_i = \overline{C'\cdot\ell}(t')$ à e_i. Cependant, on utilisant l'accesseur de la super-classe $C\cdot\ell$, on trouve aussi le type $t_i = \overline{C\cdot\ell}(t)$ pour ce même champ. Donc, le système de types va permettre d'appliquer à e_i toute opération qui attend le type t_i. Il faut donc que e_i, de type t'_i accepte toute ces opérations. En conséquence, il faut que t'_i soit un sous-type de t_i.

Cette propriété de monotonie est imposée par l'axiome suivant :

AXIOME (CHAMPS-COVARIANTS).
Soit C' une sous-classe de C et ℓ un champs de C. Si t' est dans l'ensemble $\overline{\#C'}$ et si t est dans $\overline{\#C}$, alors $t' \leqslant t$ implique $\overline{C'\cdot\ell}(t') \leqslant \overline{C\cdot\ell}(t)$.

Exemple. Dans notre algèbre simple, les fonctions $\overline{C\cdot\ell}$ peuvent être engendrées automatiquement des déclarations entrées par le programmeur. dans le tableau suivant, on a fait figurer dans la colonne de gauche ces déclarations dans une syntaxe hypothétique et dans la colonne de droite les abstractions

engendrées :

abstract class ObjetColoré {	$\overline{\text{ObjetColoré}\cdot c} : \text{ObjetColoré} \mapsto \text{Couleur}$
c: Couleur;	
}	
class Point {	$\overline{\text{Point}\cdot x} : \text{Point} \mapsto \text{int}$
x: int;	$\overline{\text{Point}\cdot y} : \text{Point} \mapsto \text{int}$
y: int;	
}	
class PointColoré $\sqsubseteq$ Point, ObjetColoré {}	$\overline{\text{PointColoré}\cdot x} : \text{PointColoré} \mapsto \text{int}$
	$\overline{\text{PointColoré}\cdot y} : \text{PointColoré} \mapsto \text{int}$
	$\overline{\text{PointColoré}\cdot c} : \text{PointColoré} \mapsto \text{Couleur}$
class List$\langle\alpha_\oplus\rangle$ {}	
class Cons$\langle\alpha_\oplus\rangle \sqsubseteq$ List {	$\forall t, \overline{\text{Cons}\cdot \mathit{head}} : \text{Cons}\langle t\rangle \mapsto t$
head: α;	$\forall t, \overline{\text{Cons}\cdot \mathit{tail}} : \text{Cons}\langle t\rangle \mapsto \text{List}\langle t\rangle$
tail: $CList\langle\alpha\rangle$;	
}	
class Nil$\langle\alpha_\oplus\rangle \sqsubseteq$ List {}	

Remarquons que l'abstraction $\overline{\text{Point}\cdot x}$ n'est pas directement applicable à un type dans $\overline{\#\text{PointColoré}}$: il faut d'abord convertir le type du point coloré en un point normal avant de pouvoir extraire le champ x par l'accesseur $\text{Point}\cdot x$.

Notons également comment l'abstraction $\overline{\text{Cons}\cdot \mathit{head}}$ traite le polymorphisme : cette fonction est capable d'extraire le type de la tête d'une liste pour tous les types t possibles. Il est intéressant de remarquer que l'axiome (CHAMPS-COVARIANTS) ne permet pas d'écrire n'importe quel type pour les champs. Par exemple, le constructeur Cons étant covariant, on a l'inégalité $\text{Cons}\langle\text{int}\rangle \leqslant \text{Cons}\langle\text{rat}\rangle$. Par l'axiome (CHAMPS-COVARIANTS), on doit donc avoir $\text{int} \leqslant \text{rat}$, ce qui est bien le cas. En revanche, si le type du champ *head* avait été $\alpha \to \alpha$, cela aurait conduit à une contradiction. L'axiome impose donc que les paramètres de type d'une déclaration de classe apparaissent dans les types des champs avec une variance compatible avec leur variance déclarée et qui détermine l'ordre de sous-typage.

D'une façon également intéressante, l'axiome permet aussi une redéfinition covariante du type des champs dans les sous-classes. Par exemple, on peut écrire :

class Segment {	**class** SegmentColoré $\sqsubseteq$ *Segment* {
p_1, p_2: Point;	p_1, p_2: PointColoré;
}	}

En revanche, la déclaration suivante est interdite :

class Illégale $\sqsubseteq$ *Segment* {
p_1, p_2: bool;
}

■

Remarque. En présence d'effets de bord (champs modifiables), l'axiome de covariance des champs engendrerait un système de types incorrect. En effet, un objet de la classe SegmentColoré pourrait être considéré comme un Segment, de sorte qu'on pourrait en remplacer le champ p_1 par un Point, ce qui est incompatible avec la déclaration de la classe SegmentColoré. Pour préserver la correction, il convient que les champs modifiables soient invariants, et non pas simplement covariants. ■

3.8.3 Règles de typage

Munis des abstractions des classes et des champs, nous pouvons maintenant écrire les règles de typage des objets et des accesseurs :

(OBJET)
$$\frac{t \in \overline{\#C} \qquad G \vdash e_1 : \overline{C \cdot \ell_1}(t) \qquad \ldots \qquad G \vdash e_n : \overline{C \cdot \ell_n}(t)}{G \vdash C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\} : t}$$

(GET)
$$\frac{}{G \vdash C \cdot \ell : t \to \overline{C \cdot \ell}(t)}$$

3.9 Typage des primitives

Toute algèbre de types doit définir, pour chaque valeur primitive a une abstraction de cette valeur qui est un monotype $\overline{a}$. Un tel monotype est appelé TYPE PRIMITIF et l'ensemble des types primitifs est appelé ALGÈBRE PRIMITIVE et est noté $\mathcal{A}^0$.

La règle de typage des constantes est alors tout naturellement la suivante :

(CST)
$$\frac{}{G \vdash a : \overline{a}}$$

Considérons maintenant une opération primitive f. On peut lui donner le type $\langle t_1^0, \ldots, t_n^0 \rangle \to t^0$ si elle accepte *toutes* les valeurs primitives de type t_i^0 ou d'un sous-type et donne un résultat de type t^0 ou d'un sous-type.

Pour exprimer cette règle, il est utile de définir pour tout type type primitif t^0 l'ensemble des valeurs primitives qui peuvent prendre le type t^0. Cet ensemble est noté $\underline{t^0}$ et on a :

$$\underline{t^0} \quad \triangleq \quad \{a \in \mathsf{PrimVals} \mid \overline{a} \leqslant t^0\}$$

Pour écrire la règle de typage, une dernière difficulté technique surgit car on ne dispose pas du constructeur des types de tuples dans l'algèbre. Ce constructeur étant connu indirectement par les propriétés de ses champs à travers les fonctions $\overline{\texttt{Tuple}_n \cdot i}$, cela complique légèrement l'écriture de la règle :

(PRIM)
$$\frac{\forall a_1 \in \underline{t_1^0}, \ldots, a_n \in \underline{t_n^0},\ f(a_1, \ldots, a_n) \in \underline{t^0} \qquad \forall i, \overline{\texttt{Tuple}_n \cdot i}(t) = t_i^0}{G \vdash f : t \to t^0}$$

Exemple. Voici une abstraction possible des valeurs primitives dans notre exemple simple d'algèbre :

$$a \in \mathbb{Z} \Rightarrow \overline{a} = \mathsf{int}$$
$$a \in \mathbb{Q} \setminus \mathbb{Z} \Rightarrow \overline{a} = \mathsf{rat}$$
$$a \in \{\mathsf{true}, \mathsf{false}\} \Rightarrow \overline{a} = \mathsf{bool}$$

La règle (PRIM) peut alors donner les types suivants aux opérations primitives : l'addition et la multiplication peuvent recevoir l'ensemble des types $\langle \mathsf{int}, \mathsf{int} \rangle \to \mathsf{int}$, $\langle \mathsf{rat}, \mathsf{rat} \rangle \to \mathsf{int}$ et $\langle \mathsf{rat}, \mathsf{rat} \rangle \to \mathsf{rat}$. Par la règle (GEN), on peut leur donner le type polymorphe qui contient ces trois monotypes. Ce type polymorphe est équivalent par rapport à un ordre de sous-typage généralisé au polytype contenant l'ensemble des monotypes de la forme $\langle t, t \rangle \to t$ pour tout $t \in \{\mathsf{int}, \mathsf{rat}\}$.

De même, l'opération de comparaison peut recevoir un polytype équivalent à $\langle \mathsf{rat}, \mathsf{rat} \rangle \to \mathsf{bool}$.

Notons que l'opération de division pose un petit problème technique : comme elle n'est pas définie quand le dénominateur est nul, la règle (PRIM) ne peut lui donner *aucun* type correct car tous nos types arithmétiques contient la valeur 0. Pour traiter ce problème, il y a plusieurs solutions. On peut étendre l'ensemble des valeurs primitives avec des valeurs supplémentaires ($+\infty$, $-\infty$, $+0$, -0, NaN) afin de rendre les opération totales. On peut aussi étendre légèrement la sémantique opérationnelle et la règle (BETA-PRIM) pour autoriser le calcul a être bloqué sans que cela ne soit une erreur de type. On peut aussi prévoir un mécanisme d'exceptions pour cela. ■

3.10 Séparation

Un ingrédient essentiel de la correction du système de types est que les types des fonctions, des objets et des valeurs primitives soient bien séparés. Par exemple, il ne faut pas qu'un type d'objet soit un sous-type d'un type de fonction, sinon on pourrait être amené à accepter l'utilisation de l'objet comme une fonction, ce qui constitue une erreur de type.

Deux ensembles de monotypes sont dits SÉPARÉS s'ils n'ont pas d'éléments directement comparables :

$$X \not\gtrless Y \quad \text{ssi} \quad \forall t \in X, \forall u \in Y, t \not\leqslant u \wedge u \not\leqslant t$$

AXIOME (SÉPARATION).
On pose les notations suivantes pour désigner respectivement les ensembles de types de fonctions, d'objets et de valeurs primitives :

$$\overline{\mathbf{fun}} \triangleq \{t \to t' \mid t, t' \in \mathcal{A}\} \qquad \textit{Types des fonctions}$$
$$\overline{\mathsf{Classes}} \triangleq \bigcup_{C \in \mathsf{Classes}} \overline{\#C} \qquad \textit{Types des objets}$$
$$\mathcal{A}^0 \triangleq \bigcup_{a \in \mathsf{PrimVals}} \overline{a} \qquad \textit{Types des valeurs primitives}$$

Les ensembles $\overline{\mathbf{fun}}$, $\overline{\mathsf{Classes}}$ *et* $\mathcal{A}^0$ *sont deux-à-deux séparés.*

3.11 Typage des méthodes

Considérons une méthode de la forme **meth** $\{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}$. Le typage de cette méthode doit résoudre trois problèmes potentiels :

1. D'abord, il faut en interdire toute utilisation en dehors d'un contexte où une fonction est attendue. Grâce à l'axiome de séparation, il suffit pour cela de donner à la méthode un type fonctionnel de la forme $t \to t'$;
2. Ensuite, il faut qu'aucun argument ne fasse échouer le dispatch dynamique. Naturellement, on peut restreindre cette exigence au cas où l'application de la méthode est bien typée, c'est-à-dire quand l'argument a le type t. On dira alors que le type $t \to t'$ est couvert par les motifs $\pi_1, \ldots, \pi_k$, ce que l'on notera $t \to t' \bowtie \pi_1; \ldots; \pi_k$ (voir section 3.11.1) ;
3. Enfin, il faut vérifier qu'on peut donner le type $t \to t'$ à chaque cas $\pi_j \Rightarrow e_j$ de définition de la méthode. À cet effet, on définit un jugement de typage auxiliaire noté $G \vdash (\pi_j \Rightarrow e_j) : t \to t'$ (voir section 3.11.2). Ce jugement doit se lire : « pour tout argument e de type t filtré par le motif π_j, l'expression réduite $[e \downarrow \pi_j]e_j$ est de type t' ».

En résumé, nous proposons donc d'écrire la règle de typage des méthodes de la façon suivante :

(METH)
$$\frac{t \to t' \bowtie \pi_1; \ldots; \pi_k \qquad \forall j, G \vdash \pi_j \Rightarrow e_j : t \to t'}{G \vdash \textbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\} : t \to t'}$$

Remarque. Notons que cette règle de typage peut être utilisée en conjonction avec la règle de généralisation pour fournir un type polymorphe à une méthode. Par exemple, considérons la méthode suivante (opération de réflexion de points selon l'axe y) en supposant que seules les classes Point et PointColoré soient définies :

meth {
  #Point $\{x = \_$ **as** $x; y = \_$ **as** $y\} \Rightarrow$ Point $\{x = -x; y = y\}$;
  #PointColoré $\{x = \_$ **as** $x; y = \_$ **as** $y; c = \_$ **as** $c\} \Rightarrow$ PointColoré $\{x = -x; y = y; c = c\}\}$

Par la règle (METH), on peut donner le type Point $\to$ Point à cette méthode. En effet, ce type autorise l'application de la méthode à des objets de classe Point et PointColoré. Si l'argument est un Point, c'est la première branche qui est choisie et le résultat de l'application est alors un Point. Si l'argument est un PointColoré, la deuxième branche est exécutée et le résultat est un PointColoré. Dans tous les cas, le type du résultat est un sous-type de Point et la méthode est donc bien typée.

Par ailleurs, on peut également donner le type PointColoré $\to$ PointColoré à la méthode. Notons que dans ce cas-là, il n'y a pas d'expression qui soit filtrée par le motif de la première branche et qui soit un sous-type de PointColoré. C'est pourquoi, la première branche est « morte » et ne peut donc pas poser de problème de typage. Seule la deuxième branche est possible et elle produit bien une expression de type PointColoré.

Grâce à la règle (GEN), il est alors possible d'unir ces deux types pour donner à la méthode le type polymorphe {Point $\to$ Point, PointColoré $\to$ PointColoré}.

Notons également que chaque cas de la méthode est typé indépendamment, c'est-à-dire qu'il n'y a pas de transmission horizontale d'information de type. Un cas $\pi_j \Rightarrow e_j$ est typé en sachant qu'il reçoit un argument filtré par le motif π_j, mais pas en sachant que c'est le motif le plus précis parmi

$\pi_1, \ldots, \pi_k$ ayant cette propriété. Cette restriction est intentionnelle : cela permet d'une part au système de types d'être indépendant de la stratégie de dispatch utilisée (*best-match, first-match, ...*), et d'autre part, cela autorise le typage séparé de cas écrits dans des modules différents.

Ainsi, le système de types est fondamentalement conçu pour qu'on puisse (et qu'on doive) typer séparément chaque cas d'une méthode. Ce choix de conception a certaines conséquences parfois déroutantes. Par exemple, modifions la méthode ci-dessus pour que le premier cas utilise un motif de sous-classe de la forme Point{...} plutôt que #Point{...}. Dans ce cas, il n'est pas possible de donner le type PointColoré $\to$ PointColoré à la méthode. En effet, un objet de classe PointColoré est maintenant filtré par le motif de la première branche. Si la méthode avait le type PointColoré $\to$ PointColoré, alors il faudrait que la première branche produise un objet de type PointColoré, ce qui n'est pas le cas, puisqu'elle retourne un Point. Ce qui est déroutant, c'est que le PointColoré sera en fait toujours passé à la deuxième branche et donc ne posera jamais de problème dans la première. Le système de types n'est donc pas aussi précis qu'il pourrait l'être en théorie, mais c'est le prix à payer pour qu'on puisse faire le typage de chaque cas séparément. ■

Le reste de cette section est consacré à la définition du test de couverture (3.11.1) et du typage des cas (3.11.2).

3.11.1 Test de couverture

Le test de couverture est un prédicat noté $t \to t' \bowtie \pi_1; \ldots; \pi_k$. S'il est vrai, il doit assurer qu'aucune valeur passée à une méthode de type $t \to t'$ ne fait échouer le dispatch avec les motifs $\pi_1, \ldots, \pi_k$. Notons que le type t' n'intervient pas du tout dans cette définition. Il est cependant commode de le faire apparaître dans la notation, notamment quand le test de couverture sera étendu aux ensembles de monotypes.

Nous allons donner une définition très extensive du test de couverture. Si aucune expression de type t ne fait échouer le dispatch, alors, on considère que le test de couverture réussit. Cette définition simple est cependant infondée car elle utilise le système de types qu'on est en train de définir. En particulier, l'une des expressions e pourrait être la méthode même qu'on est en train de typer. En fait, ceci n'est pas vraiment un problème : du point de vue du filtrage, toutes les fonctions et toutes les méthodes sont équivalentes et sont représentées par le symbole **`fun`**. C'est pourquoi, il suffit de tester le dispatch uniquement sur un ensemble plus restreint d'expressions, dites EXPRESSIONS DE TESTS :

$$
\begin{aligned}
e^{\text{test}} ::= {} & a \\
& \mid \bot_{\texttt{fun}} && \text{où } \bot_{\texttt{fun}} \equiv \texttt{fun}\ x \Rightarrow \texttt{fix}\ y \Rightarrow y \\
& \mid C\left\{\ell_1 = e_1^{\text{test}}; \ldots; \ell_n = e_n^{\text{test}}\right\}
\end{aligned}
$$

Dans cette grammaire, $\bot_{\texttt{fun}}$ est choisie car elle représente la fonction avec le type le plus général possible puisqu'elle prend n'importe quel argument et ne retourne jamais. Notons aussi que toutes les expressions de test sont sans variable libre.

Le typage des expressions de test ne dépend maintenant que des règles de typage énoncées jusqu'à présent car aucune expression de test n'est une méthode. On peut donc donner la définition bien fondée suivante du TEST DE COUVERTURE :

$$
\frac{\forall e^{\text{test}}, \left(\varnothing \vdash e^{\text{test}} : t\right) \Rightarrow \text{dispatch}(e^{\text{test}}; \pi_1; \ldots; \pi_k) \neq \text{erreur}}{t \to t' \bowtie \pi_1; \ldots; \pi_k}
$$

3.11.2 Typage des cas

Dans la règle de typage, le prédicat $G \vdash \pi \Rightarrow e : t \to t'$ exprime que pour tous les arguments possibles de type t, l'expression e prend le type t' quand on y remplace les variables libres de π par les sous-expressions correspondantes de l'argument. Le typage de l'expression e a lieu dans G étendu avec un environnement G' qui décrit le type des variables libres de π. Pour typer e, nous allons utiliser non seulement l'information que l'argument a le type t, mais aussi le fait qu'il est filtré dynamiquement par π.

Par exemple, considérons une méthode de type $\mathsf{Point} \to \mathsf{Point}$ et un cas de cette méthode utilisant le motif ObjetColoré. L'information dont on dispose pour typer le corps de ce cas est alors non seulement que l'argument a le type Point, mais aussi qu'il est d'une sous-classe de ObjetColoré.

Pour ce faire, nous introduisons un jugement auxiliaire de typage des motifs, noté $t \vdash \pi : t''; G'$. Le « contrat » intuitif que ce jugement doit satisfaire est le suivant : pour tout argument e de type t tel que e est filtré par le motif π, il doit exister un type t'' plus précis que t et un environnement G' tels que $t \vdash \pi : t''; G'$. En outre, e doit alors avoir le type t'', ce qui reflète l'information supplémentaire qu'on a déduit du fait que qu'il est filtré par π. Enfin, la projection de e sur le motif π doit avoir le type G'.

Supposons que ce jugement intermédiaire soit déjà défini et qu'il remplisse bien ce contrat. On peut alors l'exploiter pour formuler la règle de typage des cas : pour atteindre l'objectif de typer le corps e en exploitant au maximum l'information de type statique et dynamique, nous demandons de typer e dans *tous* les environnements G' possibles, ce qui s'écrit de la façon suivante :

$$\text{(CAS)} \qquad \frac{\forall t'' G', (t \vdash \pi : t''; G') \Rightarrow (G \oplus G' \vdash e : t')}{G \vdash \pi \Rightarrow e : t \to t'}$$

Remarque. Notons que cette règle de typage peut avoir une infinité de prémisses, ce qui la fait ressembler à la règle de généralisation (GEN). Il peut paraître étrange que le typage fin des cas exige de typer e dans une infinité d'environnements. En fait, cela peut très bien représenter une exigence de typage plus faible qu'un environnement unique, car chacun peut donner type précis aux arguments, alors qu'un environnement unique pourrait représenter une approximation grossière.

Pour illustrer ce phénomène, reprenons notre exemple ci-dessus de la méthode de type $\mathsf{Point} \to \mathsf{Point}$ et ayant un cas défini avec le motif $\pi = \mathsf{ObjetColoré}$. Il est tout-à-fait correct de poser le jugement intermédiaire $\mathsf{Point} \vdash \pi : \mathsf{Point}; \varnothing$ car on vérifie facilement qu'il remplit bien le contrat exprimé plus haut. Cependant, ce jugement engendrerait un système d'une précision médiocre, puisqu'on ne pourrait même pas utiliser le fait que l'argument est bien un ObjetColoré. C'est pourquoi, on cherche plutôt à définir le typage des motifs de telle sorte qu'on ait $\mathsf{Point} \vdash \pi : t''; \varnothing$ pour tous les types t'' qui sont simultanément sous-types de Point et de ObjetColoré. Cela revient, dans la règle (CAS), à exiger de typer le corps du cas successivement pour tous les tels types t''.

Dans certaines algèbres particulières, l'ensemble des types t'' à utiliser pour typer le corps du cas peut admettre une borne supérieure, par exemple PointColoré dans notre exemple. Notons toutefois que dans le cas général, cette borne supérieure n'a aucune raison d'exister, car aucune structure particulière n'est imposé à l'ordre partiel des monotypes.

Pour finir avec la règle (CAS), remarquons que seul l'environnement G' est utilisée dans le typage de e, mais pas le type t''. La raison en est que t'' ne sert que lors de la définition du jugement $t \vdash \pi : t''; G'$ pour traiter les lieurs, comme nous allons maintenant le voir. ■

Le jugement $t \vdash \pi : t'; G$ est défini à partir de règles d'inférences que nous allons maintenant présenter une à une.

Le type de sortie t' généré par le jugement de typage d'un motif est *a priori* un type plus précis que le type d'entrée t. Quand le motif est un lieur pour la variable x, on peut donc ajouter ce type dans l'environnement généré, d'où la règle suivante :

(π-LIEUR)
$$\frac{t \vdash \pi : t'; G}{t \vdash \pi \ \mathbf{as}\ x : t'; G[x : t']}$$

Le motif universel n'ayant pas de variable libre génère un environnement vide. Il accepte n'importe quel type en entrée et produit le même type en sortie puisqu'il ne pose aucune contrainte sur son argument :

(π-UNIVERSEL)
$$\frac{}{t \vdash \_ : t; \varnothing}$$

Pour qu'il existe un argument de type t accepté par un motif primitif ϖ, il suffit que t soit le type d'une constante acceptée par ϖ :

(π-PRIM)
$$\frac{a \in \varpi}{\overline{a} \vdash \varpi : \overline{a}; \varnothing}$$

Les types des objets filtrés par les motifs $\#C\,\{\ldots\}$ doivent être dans $\overline{\#C}$ et le type de chacun des champs doit être compatible avec les motifs des champs :[2]

(π-CLASSE)
$$\frac{t \in \overline{\#C} \qquad \forall i, \overline{C \cdot \ell_i}(t) \vdash \pi_i : t'_i; G_i}{t \vdash \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t; G_1 \oplus \cdots \oplus G_n}$$

Cette dernière règle pose cependant un léger problème : le type des motifs portant sur les champs de l'objet peut très bien être strictement plus précis que le type du champ obtenu par $\overline{C \cdot \ell_i}$. Par exemple, si on considère le motif $\#\mathsf{Cons}\,\{tail = \#\mathsf{Cons}\}$, on constate que le motif sur le champ *tail* contraint celui-ci à être un Cons, ce qui est plus précis que le type général du champ, qui est une List. En conséquence, il est nécessaire de rajouter la règle de subsomption suivante :

(π-SUB)
$$\frac{t \vdash \pi : t'; G \qquad t \leqslant u}{u \vdash \pi : t'; G}$$

Cette règle se comprend intuitivement : si π peut accepter un argument de type t, alors par la règle (SUB), celui-ci peut aussi être considéré avec type u.

Remarque. Dans la règle (π-CLASSE), on remarque que le type t'_i présent dans la prémisse n'est pas du tout utilisé dans la conclusion de la règle. Concrètement, cela signifie que l'information supplémentaire qu'on a déduit sur le type de chacun des champs de l'objet filtré est perdue.

[2] Rappelons que les règles de bonnes formations syntaxiques imposent que les domaines des G_i soient disjoints deux à deux.

Par exemple, considérons à nouveau le motif $\#\mathsf{Cons}\left\{\mathit{tail} = \#\mathsf{Cons}\right\}$ et le type $t = \mathsf{List}\langle\mathsf{int}\rangle$. Si une expression *e* est filtrée par ce motif, on sait que *e* est un objet construit avec la classe Cons et on sait aussi que son champ *tail* est également construit avec la classe Cons. Cependant, le choix de notre algèbre de types ne nous donne aucun moyen d'exprimer ces propriétés dans un type unique. Le mieux qu'on puisse faire est de demander de typer le corps du cas en faisant l'hypothèse que l'argument de la méthode est de type $\mathsf{Cons}\langle\mathsf{int}\rangle$. Le problème, c'est que si on applique l'accesseur $\mathsf{Cons}\cdot\mathit{tail}$ à l'argument, alors le résultat sera de type $\mathsf{List}\langle\mathsf{int}\rangle$, un type moins précis que l'information dont on dispose.

On peut noter que le programmeur peut toujours utiliser un lieur pour obtenir le typage le plus précis. Par exemple, au lieu d'utiliser l'accesseur $\mathsf{Cons}\cdot\mathit{tail}$, il pourrait commencer par écrire le motif $\#\mathsf{Cons}\left\{\mathit{tail} = \#\mathsf{Cons}\ \mathbf{as}\ t\right\}$. De cette façon, la variable t a bien le type précis $\mathsf{Cons}\langle\mathsf{int}\rangle$. Ce mécanisme peut être utilisé pour l'accès à tous les champs profonds et règle essentiellement le problème.

Si on ne disposait pas des lieurs, il faudrait alors garder trace de toutes les contraintes sur le type des champs lors d'un filtrage profond. Cela constituerait une extension considérable des environnements de typage dont les conséquences ne sont pas claires. Remarquons finalement que, intrinsèquement, le problème ne vient pas du filtrage, mais du fait que, dans notre exemple, la classe Cons n'a qu'un seul paramètre de type. Comme c'est le programmeur qui déclare les classes, on peut considérer que c'un choix délibéré de sa part de perdre de l'information au profit de la lisibilité des types. En effet, dans une algèbre de types appropriée, contenant notamment des types récursifs, on pourrait très bien imaginer que la classe Cons ait deux paramètres de type (un pour chaque champ), ce qui permettrait de ne jamais perdre d'information sur le type des champs. Un tel système tend cependant à produire des types difficile à lire et à manipuler. C'est pourquoi, perdre de l'information dès la déclaration de classe est parfois un bon compromis qui se paie par un pouvoir expressif légèrement moins grand. ■

Finalement, la dernière règle que nous introduisons concerne les motifs ouverts acceptant des objets d'une classe ou d'une sous-classe :

(π-SOUS-CLASSE)
$$\frac{t' \leqslant t \qquad t \vdash \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t; G}{t' \vdash C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t'; G}$$

3.11.3 Un dernier axiome

La correction du typage des motifs de la forme $C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$ exige un dernier axiome sur les algèbres de type. En effet, un argument accepté par ce motif est un objet construit avec une classe C' sous-classe de C. On lui donne donc un type élément de $\overline{\#C'}$. Cependant, la règle de typage du motif implique qu'on vérifie le corps du cas uniquement dans l'hypothèse où l'argument a un sous-type quelconque d'un élément de $\overline{\#C}$. Pour assurer la correction, on doit donc supposer que tout élément de $\overline{\#C'}$ est toujours approximable par un élément de $\overline{\#C}$:

AXIOME (APPROXIMATION).
Si $C' \sqsubseteq C$ et si $t' \in \overline{\#C'}$, alors il existe $t \in \overline{\#C}$ tel que $t' \leqslant t$.

3.12 Récapitulatif

Cette section reprend succinctement toutes les définitions données dans ce chapitre.

Pré-algèbres de types

Une pré-algèbre de types est la donnée des éléments suivants :
- un ensemble $\mathcal{A}$ de MONOTYPES ;
- une relation binaire $\leqslant \subset \mathcal{A} \times \mathcal{A}$;
- un opérateur binaire $\rightarrow \in \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$;
- pour chaque classe $C \in \mathsf{Classes}$, un ensemble $\overline{\#C} \subseteq \mathcal{A}$;
- pour chaque classe $C \in \mathsf{Classes}$ et chaque étiquette $\ell \in \mathsf{Champs}_C$, une fonction totale $\overline{C \cdot \ell} \in \overline{\#C} \rightarrow \mathcal{A}$;
- pour chaque valeur primitive $a \in \mathsf{PrimVals}$, un monotype $\overline{a} \in \mathcal{A}$.

Quand on voudra expliciter l'algèbre de types utilisée, on utilisera les notations suivantes pour chacun des éléments de l'algèbre : $\leqslant^{\mathcal{A}}$, $\rightarrow^{\mathcal{A}}$, $\overline{\#C}^{\mathcal{A}}$, $\overline{C \cdot \ell}^{\mathcal{A}}$ et $\overline{a}^{\mathcal{A}}$.

Définitions auxiliaires

Les ensembles suivants sont définis :

$$
\begin{array}{rcll}
\overline{\mathsf{fun}} & \triangleq & \{t \rightarrow t' \mid t, t' \in \mathcal{A}\} & \\
\overline{\mathsf{Classes}} & \triangleq & \bigcup_{C \in \mathsf{Classes}} \overline{\#C} & \\
\mathcal{A}^0 & \triangleq & \bigcup_{a \in \mathsf{PrimVals}} \overline{a} & \\
\underline{t^0} & \triangleq & \{a \in \mathsf{PrimVals} \mid \overline{a} \leqslant t^0\} & (t^0 \in \mathcal{A}^0)
\end{array}
$$

ainsi que les notations :

$$
\begin{array}{rcl}
X \not\geqslant Y \quad \text{ssi} \quad \forall x \in X, y \in Y, x \not\leqslant y \wedge y \not\leqslant x & & \\
\uparrow X & \triangleq & \{y \mid \exists x \in X, x \leqslant y\} \\
\downarrow X & \triangleq & \{y \mid \exists x \in X, y \leqslant x\}
\end{array}
$$

Les lettres T, T' désignent des ensembles non-vides de monotypes. Les lettres $t^0, u^0, \ldots$ des éléments de l'ensemble $\mathcal{A}^0$.

Axiomes

Une algèbre de types est une pré-algèbre qui vérifie les axiomes suivants :

(SOUS-TYPAGE-ORDRE) $\begin{cases} \forall t \in \mathcal{A}, t \leqslant t \\ \forall t t' t'' \in \mathcal{A}, t \leqslant t' \wedge t' \leqslant t'' \Rightarrow t \leqslant t'' \\ \forall t t' \in \mathcal{A}, t \leqslant t' \wedge t' \leqslant t \Rightarrow t = t' \end{cases}$

(FLÈCHE-VARIANCE) $\forall t t' u u' \in \mathcal{A}, t \to u \leqslant t' \to u' \iff t' \leqslant t \wedge u \leqslant u'$

(CLASSES-COVARIANTES) $\forall C C' \in \mathsf{Classes}, \forall t \in \overline{\#C}, \forall t' \in \overline{\#C'}, t' \leqslant t \Rightarrow C' \sqsubseteq C$

(CHAMPS-COVARIANTS)
$\forall C C' \in \mathsf{Classes}, \forall \ell \in \mathsf{Champs}_C, \forall t \in \overline{\#C}, t' \in \overline{\#C'}, C' \sqsubseteq C \wedge t' \leqslant t \Rightarrow \overline{C'\cdot\ell}(t') \leqslant \overline{C\cdot\ell}(t)$

(SÉPARATION) $\overline{\mathbf{fun}} \not\geqslant \overline{\mathsf{Classes}} \wedge \overline{\mathbf{fun}} \not\geqslant \mathcal{A}^0 \wedge \overline{\mathsf{Classes}} \not\geqslant \mathcal{A}^0$

(APPROXIMATION) $\forall C C' \in \mathsf{Classes}, C' \sqsubseteq C \Rightarrow \overline{\#C'} \subseteq \downarrow\overline{\#C}$

Expressions de test

$$
\begin{aligned}
e^{\text{test}} ::= {} & a \\
& \mid \bot_{\mathbf{fun}} \qquad\qquad \text{où } \bot_{\mathbf{fun}} \equiv \mathbf{fun}\ x\ \Rightarrow \mathbf{fix}\ y\ \Rightarrow y \\
& \mid C\left\{\ell_1 = e_1^{\text{test}}; \ldots; \ell_n = e_n^{\text{test}}\right\}
\end{aligned}
$$

Règles de typage

(VAR) $G \vdash x : G(x)$

(APP) $\dfrac{G \vdash e_1 : t_2 \to t \qquad G \vdash e_2 : t_2}{G \vdash e_1\, e_2 : t}$

(FUN) $\dfrac{G[x : t] \vdash e : t'}{G \vdash \mathbf{fun}\ x\ \Rightarrow e : t \to t'}$

(OBJET) $\dfrac{t \in \overline{\#C} \qquad G \vdash e_1 : \overline{C\cdot\ell_1}(t) \qquad \ldots \qquad G \vdash e_n : \overline{C\cdot\ell_n}(t)}{G \vdash C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\} : t}$

(GET) $\dfrac{t' = \overline{C\cdot\ell}(t)}{G \vdash C\cdot\ell : t \to t'}$

(CST) $$\frac{}{G \vdash a : \overline{a}}$$

(PRIM) $$\frac{\forall a_1 \in \underline{t_1^0}, \ldots, a_n \in \underline{t_n^0},\ f(a_1, \ldots, a_n) \in \underline{t^0} \qquad \forall i, \overline{\mathtt{Tuple}_n \cdot i}(t) = t_i^0}{G \vdash f : t \to t^0}$$

(INST) $$\frac{G \vdash e : T \qquad t \in T}{G \vdash e : t}$$

(GEN) $$\frac{T \neq \varnothing \qquad \text{pour tout } t \in T,\ G \vdash e : t}{G \vdash e : T}$$

(SUB) $$\frac{G \vdash e : t \qquad t \leqslant t'}{G \vdash e : t'}$$

(LET) $$\frac{G \vdash e_1 : T_1 \qquad G[x : T_1] \vdash e_2 : T_2}{G \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : T_2}$$

(FIX) $$\frac{G[x : T] \vdash e : T}{G \vdash \mathtt{fix}\ x \Rightarrow e : T}$$

(METH) $$\frac{t \to t' \bowtie \pi_1; \ldots; \pi_k \qquad \forall j, G \vdash \pi_j \Rightarrow e_j : t \to t'}{G \vdash \mathtt{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\} : t \to t'}$$

(CAS) $$\frac{\forall t'' G' (t \vdash \pi : t''; G') \Rightarrow G \oplus G' \vdash e : t'}{G \vdash \pi \Rightarrow e : t \to t'}$$

(π-PRIM) $$\frac{a \in \varpi}{\overline{a} \vdash \varpi : \overline{a}; \varnothing}$$

(π-UNIVERSEL) $$\frac{}{t \vdash \_ : t; \varnothing}$$

(π-LIEUR) $$\frac{t \vdash \pi : t'; G}{t \vdash (\pi\ \mathtt{as}\ x) : t'; G[x : t']}$$

(π-SUB) $$\frac{t \vdash \pi : t'; G \qquad t \leqslant u}{u \vdash \pi : t'; G}$$

(π-CLASSE) $$\frac{t \in \overline{\#C} \qquad \forall i, \overline{C \cdot \ell_i}(t) \vdash \pi_i : t_i'; G_i}{t \vdash \#C\ \{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t; G_1 \oplus \cdots \oplus G_n}$$

(π-SOUS-CLASSE)
$$\frac{t' \leqslant t \qquad t \vdash \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t; G}{t' \vdash C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t'; G}$$

et enfin :

(COUVERTURE)
$$\frac{\forall e^{\text{test}},\, \left(\varnothing \vdash e^{\text{test}} : t\right) \Rightarrow \text{dispatch}(e^{\text{test}}; \pi_1; \ldots; \pi_k) \neq \textsf{erreur}}{t \to t' \bowtie \pi_1; \ldots; \pi_k}$$

Chapitre 4

Sûreté du typage algébrique

L'objectif de ce chapitre est de montrer le théorème de correction du typage algébrique vis-à-vis de la sémantique opérationnelle.

4.1 Énoncé

THÉORÈME 1 (SÛRETÉ DU TYPAGE). *S'il existe T tel que $\varnothing \vdash e : T$, alors $e \not\longrightarrow^\star$ erreur.*

La preuve de ce théorème découle de deux lemmes. Le lemme de progression (lemme 4.7, page 69) montre qu'une expression bien typée ne provoque pas d'erreur de type à l'exécution. Le lemme de préservation du typage (lemme 4.12, page 74) montre que le typage est préservé quand on évalue une expression. Le reste de ce chapitre est consacré à la démonstration de ces deux lemmes.

4.2 Préliminaires

4.2.1 Subsomption généralisée

Soit T et T' deux polytypes. La relation binaire $\leqslant^\sharp$ est définie comme suit :

$$T \leqslant^\sharp T' \text{ ssi } \forall t' \in T', \exists t \in T, t \leqslant t'$$

Cette relation est un préordre qui étend la relation $\leqslant$ aux polytypes. En particuler, quand on identifie t et $\{t\}$, on a $\{t\} \leqslant^\sharp \{u\}$ chaque fois que $t \leqslant u$. Notons aussi qu'on a $T \leqslant^\sharp t$ chaque fois que $t \in T$.

LEMME 4.1 (SUBSOMPTION GÉNÉRALISÉE). *La règle suivante de* SUBSOMPTION GÉNÉRALISÉE *est admissible :*

(SUB$^\sharp$)
$$\frac{G \vdash e : T \qquad T \leqslant^\sharp T'}{G \vdash e : T'}$$

Démonstration. Soit $t' \in T'$. Par définition de $T \leqslant^\sharp T'$, il existe $t \in T$ tel que $t \leqslant t'$. On déduit $G \vdash e : t$ par la règle (INST) puis $G \vdash e : t'$ par (SUB). On a donc prouvé ce dernier jugement pour tout $t \in T'$. On peut donc appliquer la règle (GEN) pour conclure. ■

4.2.2 Simplification des dérivations

Une dérivation de typage est dite SIMPLE si elle ne se termine ni par (SUB), ni par (INST), ni par (GEN). Quand un jugement $G \vdash e : T$ admet une dérivation simple, on écrit $G \vdash^0 e : T$. L'intérêt de cette notion est que la dernière règle d'une dérivation simple est déterminée par la forme de l'expression, ce qui est très utile pour les preuves par induction sur la structure de l'expression.

LEMME 4.2. *Pour toute dérivation de $G \vdash e : T$ et tout élément $t \in T$, il existe une dérivation simple $G \vdash^0 e : T'$ pour un certain $T' \leqslant^\sharp t$.*

Démonstration. On procède par induction sur la dérivation de $G \vdash e : T$ (**1**) (voir section 4.6, page 76 pour les problèmes de fondements).

Si la dérivation de (1) termine par (SUB), nous savons que $t = T$ et qu'il existe t' tel que $t' \leqslant t$ (**2**) et $G \vdash e : t'$ (**3**). De l'hypothèse d'induction appliquée à (3), on déduit qu'il existe T' tel que $T' \leqslant^\sharp t'$ (**4**) et $G \vdash^0 e : T'$. En appliquant la transitivité de $\leqslant^\sharp$ à (2) et (4), on conclut que $T' \leqslant^\sharp t$.

Si la dérivation de (1) termine par (INST), alors T est forcément un monotype et on a $t = T$. De plus, il existe T'' tel que $t \in T''$ (**5**) et $G \vdash e : T''$ (**6**). En appliquant l'hypothèse d'induction à (5) et (6), on voit qu'il existe T' tel que $T' \leqslant^\sharp t$ et $G \vdash^0 e : T'$.

Si la dérivation de (1) termine par (GEN), alors l'une des prémisses de la règle est $G \vdash e : t$ (**7**) (car $t \in T$). En appliquant l'hypothèse d'induction à (7), on trouve T' tel que $T' \leqslant^\sharp t$ et $G \vdash^0 e : T'$.

Sinon, la dérivation de (1) est déjà simple. On conclut en prenant $T' = T$, car on a bien $T \leqslant^\sharp t$ puisque $t \in T$. ■

4.2.3 Type des valeurs

LEMME 4.3. *Si $G \vdash e_1\, e_2 : t$, alors il existe t_2 tel que $G \vdash e_1 : t_2 \to t$ et $G \vdash e_2 : t_2$.*

Démonstration. Appliquons le lemme 4.2 à $G \vdash e_1\, e_2 : t$. Il vient qu'il existe $t' \leqslant t$ (**1**) et $G \vdash^0 e_1\, e_2 : t'$ (**2**). Comme l'expression $e_1\, e_2$ est une application et que la dérivation (2) est simple, elle ne peut donc se terminer que par la règle (APP). On a donc $G \vdash e_2 : t_2$ et $G \vdash e_1 : t_2 \to t'$ (**3**). Par l'axiome (FLÈCHE-VARIANCE) appliqué à (1), il vient $t_2 \to t' \leqslant t_2 \to t$ (**4**) et, en appliquant (SUB) à (3) et (4), on conclut $G \vdash e_1 : t_2 \to t$. ■

Un élément de l'ensemble $\mathsf{Classes} \cup \mathsf{PrimVals} \cup \{\mathbf{fun}\}$ est appelé NŒUD et noté avec les lettre N, N', etc. Pour tout nœud N, $\overline{N}$ est défini comme l'ensemble $\overline{\#C}$ si N vaut C, $\overline{a}$ s'il vaut a et $\overline{\mathbf{fun}}$ s'il vaut **fun**.

LEMME 4.4. *Soit V une valeur telle que $G \vdash V : t$ et $N = \text{tête}(V)$. Alors, $t \in \uparrow\overline{N}$.*

Démonstration. Par le lemme de simplification 4.2, considérons une dérivation simple de $G \vdash V : t$. On a donc $T \leqslant^\sharp t$ (**1**) et $G \vdash^0 V : T$ (**2**). On considère ensuite tous les cas possibles de valeur.

• Si $V = a$, on a tête$(V) = a$ et, comme la dérivation (2) se termine forcément par (CST), $T = \overline{a}$ et donc $\overline{a} \leqslant t$.

• Si $V = C\,\{\ldots\}$, on a tête$(V) = C$ et, comme la dérivation (2) se termine nécessairement par (OBJET), T est un monotype dans $\overline{\#C}$ et donc $t \in \uparrow\overline{\#C}$.

• Si $V = \texttt{fun}\ x\ \Rightarrow \ldots$ ou $V = C{\cdot}\ell$ ou $V = f$, on a tête$(V) = \texttt{fun}$. La dérivation de (2) se terminant nécessairement par (FUN), (GET) ou (PRIM), T est forcément un type fonctionnel élément de $\overline{\texttt{fun}}$ donc de $\uparrow\overline{\texttt{fun}}$.

• Si $V = \texttt{meth}\ \{\ldots\}$, on a tête$(V) = \texttt{fun}$ et, la dérivation de (2) se terminant forcément par (METH), T est un monotype fonctionnel. Donc $t \in \uparrow\overline{\texttt{fun}}$. ■

4.3 Lemme de progression

LEMME 4.5. *Si $G \vdash E[e] : t$, alors il existe t' tel que $G \vdash e : t'$.*

Démonstration. Par le lemme de simplification 4.2, considérons un typage simple de $E[e]$. Si E est de la forme $[]\,e'$ ou $e'\,[]$, alors la dérivation termine forcément par la règle (APP), l'une des prémisses de cette règle est un typage de e. Si E vaut $\texttt{let}\ x = []\ \texttt{in}\ e'$, alors le typage simple termine par la règle (LET) et fournit un typage de e. En appliquant la règle (INST), on en obtient un typage monomorphe. Enfin, si E est de la forme $C\,\{\ell = []; \ldots\}$, alors la dernière règle appliquée dans le typage simple est forcément (OBJET) et une des prémisses fournit un typage monomorphe de e. ■

LEMME 4.6. *Soit e une expression quelconque. Il existe une expression de test e^{test} telle que :*
- *pour tout type t et tout environnement G, si $G \vdash e : t$, alors $G \vdash e^{test} : t$;*
- *pour tout motif π, $\mathsf{filtre}(e, \pi) = \mathsf{filtre}(e^{test}, \pi)$.*

Démonstration. On obtient e^{test} en remplaçant dans e toute occurrence d'une expression dont la tête est $\texttt{fun}$ par $\bot_{\texttt{fun}}$. Une induction simple sur la dérivation de $G \vdash e : t$ permet d'obtenir les résultats espérés. Cette preuve est fastidieuse mais sans difficulté. ■

LEMME 4.7 (PROGRESSION). *Si $G \vdash e : t$ alors $e \not\longrightarrow \mathsf{erreur}$.*

Démonstration. Par l'absurde, supposons que $e \longrightarrow \mathsf{erreur}$ et procédons par induction sur la dérivation de cette assertion. On considère donc la dernière règle appliquée R dans cette dérivation.

• Si R est (ERREUR-CONTEXTE), alors on a $e = E[e']$ et $e' \longrightarrow \mathsf{erreur}$. Par le lemme 4.5, on sait que $G \vdash e' : t'$ pour un certain t'. En appliquant l'hypothèse d'induction, on voit que $e' \not\longrightarrow \mathsf{erreur}$, ce qui est absurde.

Dans tous les autres cas, on constate que dans toutes les règles de déduction d'une erreur, e est forcément l'application d'une valeur, appelons la V, notons $N =$ tête(V) et écrivons $e = V\,e'$. En appliquant le lemme 4.3, il vient $G \vdash V : t' \to t$ (**1**) et $G \vdash e' : t'$ (**2**). En appliquant le lemme 4.4 à (1), il vient que $t' \to t \in \uparrow\overline{N}$. Par l'axiome (SÉPARATION), on déduit que N est forcément égal à $\texttt{fun}$. R ne peut donc pas être (ERREUR-FUN) et V_0 est donc une valeur fonctionnelle, c'est-à-dire une fonction, une méthode, une primitive ou un accesseur. Examinons maintenant ces différents cas.

• $V = f$. Dans les différentes règles R applicables dans ce cas, observons que e' est toujours une valeur, notons la V' et posons $N' =$ tête(V'). Considérons maintenant une dérivation simple de (1). Celle-ci se termine forcément par la règle (PRIM) et on a $G \vdash^0 f : u' \to u^0$ (**3**), $u' \to u^0 \leqslant t' \to t$ (**4**) et, pour tout $i \in [1, n]$, $\overline{\texttt{Tuple}_n{\cdot}i}(u') = u'_i$ (**5**) et $u'_i \in \mathcal{A}^0$ (**6**) et enfin $\underline{u'_1} \times \cdots \times \underline{u'_n} \subseteq \mathrm{dom}(f)$ (**7**). Comme $\overline{\texttt{Tuple}_n{\cdot}i}$ a pour domaine $\overline{\#\texttt{Tuple}_n}$, on sait que $u' \in \overline{\#\texttt{Tuple}_n}$ (**8**). Par l'axiome (FLÈCHE-VARIANCE) appliqué à (4), il vient $t' \leqslant u'$ (**9**). En appliquant le lemme 4.4 à la

dérivation (2) et sachant que $V' = e'$, on peut écrire $t' \in \uparrow\overline{N'}$ (**10**). Par transitivité de $\leqslant$ appliqué à (9) et (10), on a donc $u' \in \uparrow\overline{N'}$ (**11**). Par l'axiome (SÉPARATION) appliqué à (8) et (11), on en déduit que le nœud N' est une classe. Notons cette classe C'. Par l'axiome (CLASSES-COVARIANTES), on déduit ensuite que $C' \sqsubseteq \texttt{Tuple}_n$ et enfin, par l'axiome ($\texttt{Tuple}_n$-MINIMAL) que $C' = \texttt{Tuple}_n$. La règle R appliquée pour déduire l'erreur de type opérationnelle ne peut donc être (ERREUR-PRIM-1) et V' est nécessairement un tuple de la forme $\langle e'_1, \ldots, e'_n \rangle$. Considérons alors une dérivation simple de (2). Celle-ci se termine forcément par la règle (OBJET) et on a $G \vdash \langle e'_1, \ldots, e'_n \rangle : v'$ (**12**), $v' \leqslant t'$ (**13**) et, pour tout $i \in [1, n]$, $G \vdash e'_i : v'_i$ (**14**) et $\overline{\texttt{Tuple}_n \cdot i}(v') = v'_i$ (**15**). Par transitivité de $\leqslant$ appliquée à (13) et (9), on a aussi $v' \leqslant u'$ (**16**). Puis, en appliquant l'axiome (CHAMPS-COVARIANTS) à cette dernière inégalité, il vient $\overline{\texttt{Tuple}_n \cdot i}(v') \leqslant \overline{\texttt{Tuple}_n \cdot i}(u')$, c'est-à-dire $v'_i \leqslant u'_i$ (**17**). On peut donc appliquer la règle (SUB) à (14) et (17) et déduire $G \vdash e'_i : u'_i$ (**18**).

Supposons maintenant que l'une des expressions e'_i soit une valeur V'_i. Notons N'_i sa tête et supposons qu'elle ne soit pas une valeur primitive, c'est-à-dire que $V'_i \notin \mathsf{PrimVals}$. Par le lemme 4.4 appliqué à (18), on sait que $u'_i \in \uparrow\overline{N'_i}$. Par ailleurs, par (6), on sait que $u'_i \in \mathcal{A}^0$. Par l'axiome (SÉPARATION), on déduit que $N'_i \in \mathsf{PrimVals}$. L'hypothèse $V'_i \notin \mathsf{PrimVals}$ est donc absurde et R ne peut être (ERREUR-PRIM-2).

Supposons maintenant que toutes les expressions e'_i soient des valeurs. Celles-ci sont donc toutes des constantes que nous notons a_i et on a $u'_i \in \uparrow\overline{a_i}$. Par définition de $\overline{u'_i}$, cela signifie que la constante a_i est dans l'ensemble $\overline{u'_i}$ et par (7) que $(a_1, \ldots, a_n) \in \mathrm{dom}(f)$. Donc la règle (ERREUR-PRIM-3) est inapplicable pour déduire que e est une erreur de type.

On a donc montré qu'une application bien typée d'une primitive ne peut jamais être une erreur de type opérationnelle.

• $V = C \cdot \ell$. Dans les différentes règles R applicables dans ce cas, observons que e' est toujours une valeur, notons la V' et posons $N' = \text{tête}(V')$. Considérons maintenant une dérivation simple de V. Celle-ci se termine forcément par la règle (GET) et il existe donc u et u' tels que $\overline{C \cdot \ell}(u') = u$ (**19**) et $u' \to u \leqslant t' \to t$ (**20**). Par la contravariance de la flèche appliquée à cette dernière inégalité, il vient que $t' \leqslant u'$. Or u' étant dans le domaine de $\overline{C \cdot \ell}$, il est aussi dans $\overline{\#C}$. Par ailleurs, le lemme 4.4 appliqué à (2) nous donne $t' \in \uparrow\overline{N'}$. L'axiome de séparation nous impose donc $N' \in \mathsf{Classes}$ et celui de covariance des classes $N' \sqsubseteq C$. En fin de compte, on a donc montré qu'une application bien typée de l'accesseur $C \cdot \ell$ ne peut jamais être une erreur de type opérationnelle.

• $V = \texttt{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}$ Nous allons montrer dans ce cas que le dispatch ne peut échouer. Considérons un typage simple de V, celui-ci se termine nécessairement par la règle (METH) et il existe donc u et u' tels que $u' \to u \leqslant t' \to t$ (**21**) et $u' \to u \bowtie \pi_1; \ldots; \pi_k$. Par la variance de la flèche appliquée à (21), on sait que $t' \leqslant u'$ (**22**) et on a donc $G \vdash e' : u'$ par la règle (SUB) appliquée à (2) et (22). En appliquant le lemme 4.6 à e' on peut trouver une expression de test e^{test} telle que $G \vdash e^{\text{test}} : u'$. Par définition du test de couverture (règle (COUVERTURE)), on en déduit que $\mathrm{dispatch}(e^{\text{test}}; \pi_1; \ldots; \pi_k) \neq \mathsf{erreur}$. Comme, par ailleurs, e' et e^{test} sont filtrées par exactement les mêmes motifs, on a donc $\mathrm{dispatch}(e^{\text{test}}; \pi_1; \ldots; \pi_k) = \mathrm{dispatch}(e'; \pi_1; \ldots; \pi_k)$. On conclut qu'on ne peut déduire que e est une erreur de type opérationnelle par la règle (ERREUR-MÉTHODE).

En fin de compte, on a montré que dans tous les cas, si e est bien typée, alors aucune règle ne permet de conclure que c'est une erreur de type opérationnelle. C'est donc que e est opérationnellement bien typée. ■

4.4 Lemme de substitution

LEMME 4.8. *Soit R un renommage de variables du programme, G un environment, π un motif et t un monotype. Le jugement $t \vdash [R]\pi : t'; G$ est équivalent à $t \vdash \pi : t'; G \circ R$.*

Démonstration. Par induction triviale sur le motif π. ■

Par la suite, on utilise la notation $G \vdash S : G'$ si $\mathrm{dom}(S) = \mathrm{dom}(G')$ et pour tout $x \in \mathrm{dom}(S)$, $G \vdash S(x) : G'(x)$.

LEMME 4.9 (LEMME DE SUBSTITUTION). *Si $G \oplus G' \vdash e : T$ et $G \vdash S : G'$, alors $G \vdash [S]e : T$.*

Démonstration. L'idée de la preuve est très simple : il s'agit de remplacer dans la dérivation de $G \oplus G' \vdash e : T$ toutes les occurrences de la règle (VAR) qui portent sur une variable x dans le domaine de S par la dérivation de $G \vdash S(x) : G'(x)$. La démonstration est rendue compliquée car il faut tenir compte des renommage nécessaires pour éviter les captures lors de substitutions. Du coup, elle est assez longue et technique, mais ne contient rien de plus que l'idée de base.

Nous allons montrer une propriété plus générale : dans les hypothèses suivante :

(**1**) $$\mathrm{dom}(G') = \mathrm{dom}(S)$$

(**2**) $$G \oplus G' \vdash e : T$$

(**3**) $$\forall x \in fv(e), x \notin \mathrm{dom}(S) \Rightarrow (G \oplus G'')(x) = (G \oplus G')(x)$$

(**4**) $$\forall x \in fv(e), x \in \mathrm{dom}(S) \Rightarrow G \oplus G'' \vdash S(x) : G'(x)$$

on a le résultat $G \oplus G'' \vdash [S]e : T$ (**5**).

On procède par induction sur la dérivation de (2), en considérant la dernière règle R appliquée.

• Si R est (CST), (PRIM), (GET), le résultat est trivial : e n'a pas de variable libre et $[S]e = e$ a le type T quel que soit l'environnement de typage.

• Si R est (APP), (OBJET), (GEN), (INST) ou (SUB), le résultat découle immédiatement de l'hypothèse d'induction appliquée aux prémisses de la règle.

• Si R est (VAR), alors e est une variable x et $T = (G \oplus G')(x)$.
  - Si $x \notin \mathrm{dom}(S)$, alors $[S]x = x$. On a, par l'hypothèse (3) : $(G \oplus G'')(x) = (G \oplus G')(x)$. Donc $(G \oplus G')(x) = T$ et, en appliquant la règle (VAR), il vient $G \oplus G'' \vdash x : T$. On conclut en remarquant $x = [S]x$;
  - Si $x \in \mathrm{dom}(S)$, alors $[S]x = S(x)$, $x \in \mathrm{dom}(G')$ et donc $(G \oplus G')(x) = G'(x)$. On conclut en écrivant l'hypothèse (4) $G \oplus G'' \vdash S(x) : G'(x)$ et en remarquant que $S(x) = [S]x$ et $G'(x) = T$.

• Il reste le cas délicat où la règle R concerne une expression e qui contient un lieur. Considérons le cas le plus compliqué, c'est-à-dire quand $R =$ (METH). Dans ce cas, nous avons $T = t \to t'$ et :

$$e = \texttt{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\}$$
$$t \to t' \bowtie \pi_1; \ldots; \pi_k$$
(**6**) $$\forall j, G \oplus G' \vdash \pi_j \Rightarrow e_j : t \to t'$$

La définition de la substitution sur les cas est alors la suivante : $[S](\pi_j \Rightarrow e_j) = [R_j]\pi_j \Rightarrow [S \oplus R_j]e_j$ où $R_j = \mathcal{R}(fv(\pi_j), fv(S \restriction V_j) \cup V_j)$ et $V_j = fv(e_j) \setminus fv(\pi_j)$. Prenons maintenant G_j et t''_j quelconques

tels que $t \vdash [R_j]\pi_j : t''_j; G_j$ (**7**). Par le lemme 4.8, on déduit de (7) que $t \vdash \pi_j : t''_j; G_j \circ R_j$. Une des prémisses de (6) nous permet alors d'écrire $G \oplus G' \oplus (G_j \circ R_j) \vdash e_j : t'$ (**8**) et cette dernière dérivation est une sous-dérivation de (2), sur laquelle on peut donc appliquer l'hypothèse d'induction. Posons maintenant $G'_j = G' \oplus (G_j \circ R_j)$, $S_j = S \oplus R_j$ et $G''_j = G'' \oplus G_j$ et montrons que e_j, t', G'_j, S_j et G''_j conviennent pour appliquer l'hypothèse d'induction.

On a $\mathrm{dom}(S_j) = \mathrm{dom}(S) \cup \mathrm{dom}(R_j)$ et $\mathrm{dom}(G'_j) = \mathrm{dom}(G') \cup \mathrm{dom}(R_j)$. On a donc bien $\mathrm{dom}(S_j) = \mathrm{dom}(G'_j)$ (hypothèse (1)). Le jugement (8) peut s'écrire $G \oplus G'_j \vdash e_j : t'$ (hypothèse (2)). Considérons maintenant $x \in fv(e_j)$ tel que $x \notin \mathrm{dom}(S_j)$. On a donc $x \notin fv(\pi_j)$ et $x \notin \mathrm{dom}(S)$. Du coup, $x \in V_j$ et donc $x \notin \mathrm{codom}(R_j)$. On a donc $(G \oplus G''_j)(x) = (G \oplus G'')(x)$ En appliquant l'hypothèse (3), il vient $(G \oplus G'')(x) = (G \oplus G')(x)$ et finalement $(G \oplus G'_i)(x)$ car $x \notin fv(\pi)_j$. L'hypothèse (3) est donc satisfaite par les candidats à l'application de l'hypothèse d'induction.

Considérons finalement $x \in fv(e_j)$ tel que $x \in \mathrm{dom}(S_j)$. Nous avons deux cas selon que $x \in fv(\pi_j)$ ou non :

- si $x \in fv(\pi_j)$, dans ce cas, $S_j(x) = R_j(x)$, $G'_j(x) = G_j(R_j(x))$ et $G''_j(R_j(x)) = G_j(R_j(x))$. On peut donc appliquer la règle (VAR) pour déduire $G \oplus G''_j \vdash S_j(x) : G'_j(x)$;
- si $x \notin fv(\pi_j)$. On a alors : $S_j(x) = S(x)$ et $G'_j(x) = G'(x)$. En appliquant l'hypothèse (4), on a donc $G \oplus G'' \vdash S_j(x) : G'_j(x)$. On conclut en remarquant que $fv(S_j(x)) \# \mathrm{codom}(R_j) = \mathrm{dom}(G_j)$, ce qui nous permet d'écrire $(G \oplus G'') \oplus G_j \vdash S_j(x) : G'_j(x)$ et l'environnement $(G \oplus G'') \oplus G_j$ est par définition égal à $G \oplus G''_j$.

Dans tous les cas, l'hypothèse (4) est donc respectée.

Nous pouvons donc appliquer l'hypothèse d'induction et il vient $G \oplus G''_j \vdash [S_j]e_j : t'$. On a donc montré que pour tout G_j et t''_j tels que $G_j \vdash [R_j]\pi_j : t \to t''_j$, on a aussi $(G \oplus G'') \oplus G_j \vdash [S_j]e_j : t'$. Donc, toutes les hypothèses sont respectées pour pouvoir réappliquer la règle (CAS) et déduire $G \oplus G'' \vdash [R_j]\pi_j \Rightarrow [S_j]e_j : t \to t'$. Il suffit alors d'appliquer la règle (METH) pour conclure.

• Les cas où R est (FUN), (LET) ou (FIX) sont similaires. ■

4.5 Lemme de préservation du typage

On définit la relation binaire $\preceq$ sur les expressions par la propriété suivante :

$$e \preceq e' \quad \text{ssi} \quad \forall G, \forall t, (G \vdash e : t) \Rightarrow \big(G \vdash e' : t\big)$$

LEMME 4.10. *Pour tout contexte E, si $e \preceq e'$, alors $E[e] \preceq E[e']$.*

Démonstration. On examine tous les cas possibles pour le contexte E.

• Si $E = []\,e_0$, alors $E[e] = e\,e_0$. Prenons G et t quelconques tels que : $G \vdash e\,e_0 : t$ (**1**). Par le lemme 4.3 appliqué à cette dernière dérivation, il existe t_0 tel que $G \vdash e : t_0 \to t$ (**2**) et $G \vdash e_0 : t_0$ (**3**). Par l'hypothèse $e \preceq e'$ appliquée à (1), on sait que $G \vdash e' : t_0 \to t$ (**4**). On déduit en appliquant la règle (APP) à (3) et (4) que $G \vdash e'\,e_0 : t$ et on conclut en remarquant que $e'\,e_0 = E[e']$.

• Si $E = e_0\,[]$, la preuve est presque identique au cas $[]\,e_0$.

• Si $E = \texttt{let}\ x = []\ \texttt{in}\ e_0$, alors $E[e] = \texttt{let}\ x = e\ \texttt{in}\ e_0$. Prenons G et t quelconques tels que $G \vdash \texttt{let}\ x = e\ \texttt{in}\ e_0 : t$ (**5**). Par le lemme 4.2, on peut trouver une dérivation simple de (5). Celle-ci termine forcément par la règle (LET) et on a $G \vdash e : T_0$ (**6**), $G[x : T_0] \vdash e_0 : T$ (**7**) et $T \leqslant^\sharp t$ (**8**).

Prenons maintenant un élément t_0 quelconque de T_0. Par la règle (INST) appliquée à (6) on a $G \vdash e : t_0$ (**9**). On peut appliquer l'hypothèse $e \preceq e'$ à (9) pour déduire $G \vdash e' : t_0$. Comme on a montré ce dernier jugement pour tout $t_0 \in T_0$, on peut appliquer la règle (GEN) pour trouver $G \vdash e' : T_0$ (**10**). On peut alors réappliquer la règle (LET) à (10) et (7), de sorte que $G \vdash \texttt{let}\ x = e'\ \texttt{in}\ e_0 : T$, c'est-à-dire $G \vdash E[e'] : T$ (**11**). On conclut en appliquant la règle (SUB$^\sharp$) à (11) et (8).

• Si $E = C\,\{\ell_1 = e_1; \ldots; \ell_i = []; \ldots; \ell_n = e_n\}$. Prenons G et t quelconques tels que :

(**12**) $$G \vdash \underbrace{C\,\{\ell_1 = e_1; \ldots; \ell_i = e; \ldots; \ell_n = e_n\}}_{E[e]} : t$$

Par le lemme de simplification 4.2, on peut trouver une dérivation simple de ce jugement. Celle-ci se termine forcément par la règle (OBJET) et il existe t_0 tel que $t_0 \leqslant t$, $t_0 \in \overline{\#C}$, $G \vdash e : \overline{C \cdot \ell_i}(t_0)$ (**13**) et $G \vdash e_j : \overline{C \cdot \ell_j}(t_0)$ pour tout $j \neq i$. En appliquant l'hypothèse $e \preceq e'$ à (13), il vient $G \vdash e' : \overline{C \cdot \ell_i}(t_0)$. On peut alors réappliquer les règles (OBJET) puis (SUB) pour conclure :

$$G \vdash \underbrace{C\,\{\ell_1 = e_1; \ldots; \ell_i = e'; \ldots; \ell_n = e_n\}}_{E[e']} : t$$

■

LEMME 4.11 (PROJECTION). *Si $G \vdash e : t$ et $\mathsf{filtre}(e, \pi) = \mathsf{vrai}$, alors il existe G' et t' tels que $t \vdash \pi : t'; G'$, $t' \leqslant t$, $G \vdash e : t'$ et $G \vdash e \downarrow \pi : G'$*

Démonstration. Écrivons $G \vdash e : t$ (**1**) et $\mathsf{filtre}(e, \pi) = \mathsf{vrai}$ (**2**) et procédons par induction sur la structure du motif π.

• Si $\pi = \_$, il suffit de prendre $t' = t$ et $G' = \varnothing$.

• Si $\pi = \varpi$, alors, par définition du filtrage, on a $e = a$ (**3**) et $a \in \varpi$ (**4**). Prenons $t' = \overline{a}$ et $G' = \varnothing$. Par le lemme 4.4, on sait que $t' \leqslant t$. Par la règle (π-PRIM), on a aussi $t' \vdash \pi : t'; \varnothing$ et par la règle (π-SUB) $t \vdash \pi : t'; \varnothing$. Enfin $e \downarrow \pi$ étant vide, l'environnement vide $\varnothing$ en est un typage correct.

• $\pi = \pi_0\ \texttt{as}\ x$. Par définition du filtrage et de la projection, on a $\mathsf{filtre}(e, \pi_0) = \mathsf{vrai}$ (**5**) et $e \downarrow \pi = e \downarrow \pi_0 \oplus [x \mapsto e]$ (**6**). Appliquons l'hypothèse d'induction à (5) et (1). On peut donc trouver t' et G'_0 tels que $t' \leqslant t$ (**7**), $t \vdash \pi_0 : t'; G'_0$ (**8**), $G \vdash e \downarrow \pi_0 : G'_0$ (**9**) et $G \vdash e : t'$. Posons alors $G' = G'_0[x : t']$. Par la règle (π-LIEUR) appliquée à (8), on a $t' \vdash \pi : t'; G'$. Par ailleurs, par (6) et (9), on a aussi $G \vdash e \downarrow \pi : G'$.

• Si $\pi = \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$, alors, par définition du filtrage, on a $e = C\,\{\ell_1 = e_1; \ldots; \ell_n = e_n\}$ (**10**) et pour tout $i \in [1, n]$,$\mathsf{filtre}(e_i, \pi_i) = \mathsf{vrai}$ (**11**). Par le lemme de simplification 4.2, on peut considérer une dérivation simple de (1). Vu la nature de e, celle-ci se termine forcément par la règle (OBJET) et il existe t' tel que $t' \leqslant t$ (**12**), $t' \in \overline{\#C}$ (**13**), $G \vdash e : t'$ (**14**) et $G \vdash e_i : \overline{C \cdot \ell_i}(t')$ (**15**). Appliquons l'hypothèse d'induction à (11) et (15) pour chaque $i \in [1, n]$. Il existe donc t'_i et G'_i tels que $t'_i \leqslant \overline{C \cdot \ell_i}(t')$ (**16**), $G \vdash e_i : t'_i$ (**17**), $\overline{C \cdot \ell_i}(t') \vdash \pi_i : t'_i; G'_i$ (**18**) et $G \vdash e_i \downarrow \pi_i : G'_i$ (**19**). Les équations (13) et (18) constituent les prémisses de la règle (π-CLASSE), ce qui permet de déduire $t' \vdash \pi : t'; G'_1 \oplus \cdots \oplus G'_n$. Posons finalement $G' = G'_1 \oplus \cdots \oplus G'_n$. Il reste à montrer que $G \vdash e \downarrow \pi : G'$. Prenons $x \in fv(\pi)$. Comme l'ensemble des variables libres de chacun des sous-motifs π_i sont deux-à-deux disjoints, il existe un unique i tel que $x \in fv(\pi_i)$ et on a alors $(e \downarrow \pi)(x) = (e_i \downarrow \pi_i)(x)$. On conclut en appliquant (19) et en remarquant que les autres G'_i ne peuvent masquer la variable x.

• $\pi = C\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$. Par définition du filtrage, on a pour une certaine classe $C' \sqsubseteq C$: $e = C'\{\ell_1 = e_1; \ldots; \ell_n = e_n; \ldots; \ell_{n'} = e_{n'}\}$ (**20**) et pour tout $i \in [1,n]$, $\mathsf{filtre}(e_i, \pi_i) = \mathsf{vrai}$ (**21**). Par le lemme de simplification 4.2, on peut considérer une dérivation simple de (1). Étant donnée la forme de e, cette dérivation se termine forcément par la règle (OBJET) et il existe t' tel que $t' \leqslant t$ (**22**), $t' \in \overline{\#C'}$ (**23**), $G \vdash e : t'$ (**24**) et, pour tout $i \in [1,n]$, $G \vdash e_i : \overline{C'\cdot\ell_i}(t')$ (**25**). L'axiome (APPROXIMATION) appliqué à (23) nous dit qu'il existe $t'' \in \overline{\#C}$ tel que $t' \leqslant t''$. En appliquant l'axiome (CHAMPS-COVARIANTS) à cette dernière inégalité et sachant $C' \sqsubseteq C$, il vient $\overline{C'\cdot\ell_i}(t') \leqslant \overline{C\cdot\ell_i}(t'')$ (**26**). Par la règle (SUB) appliquée à (25) et (26), on a $G \vdash e_i : \overline{C\cdot\ell_i}(t'')$ (**27**). Appliquons maintenant l'hypothèse d'induction à (21) et (27) pour chaque $i \in [1,n]$. Il existe donc t'_i et G'_i tels que $t'_i \leqslant \overline{C\cdot\ell_i}(t'')$ (**28**), $G \vdash e_i : t'_i$ (**29**), $\overline{C\cdot\ell_i}(t'') \vdash \pi_i : t'_i; G'_i$ (**30**) et $G \vdash e_i \downarrow \pi_i : G'_i$ (**31**). Le jugement (30) et le fait que $t'' \in \overline{\#C}$ constituent les prémisses de la règle (π-CLASSE), ce qui permet de déduire :

(**32**) $$t'' \vdash \#C\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : t''; G'_1 \oplus \cdots \oplus G'_n$$

Finalement, cette dernière équation plus l'inégalité $t' \leqslant t''$ sont les prémisses de (π-SOUS-CLASSE) et on conclut $t' \vdash \pi : t'; G'_1 \oplus \cdots \oplus G'_n$.

Le typage de la substitution $e \downarrow \pi$ est identique au cas précédent. ■

LEMME 4.12 (PRÉSERVATION DU TYPAGE). *Si $e \longrightarrow e'$, alors $e \preceq e'$.*

Démonstration. Par induction sur la dérivation de $e \longrightarrow e'$.

Si la dernière règle appliquée est (CONTEXTE), alors il existe E, e_0 et e'_0 tels que $e = E[e_0]$, $e' = E[e'_0]$ et $e_0 \longrightarrow e'_0$ (**1**). En appliquant l'hypothèse d'induction à (1), il vient $e_0 \preceq e'_0$ (**2**). On conclut en appliquant le lemme 4.10 à (2).

• Si la dernière règle appliquée dans la dérivation de $e \longrightarrow e'$ est différente de (CONTEXTE), considérons une dérivation quelconque de $G \vdash e : t$. Par le lemme de simplification 4.2, il existe un polytype $T \leqslant^\sharp t$ et une dérivation simple de $G \vdash^0 e : T$. Supposons qu'on puisse en déduire que $G \vdash e' : T$. Dans ce cas, on sait, par définition de $\leqslant^\sharp$, qu'il existe un certain $t' \in T$ tel que $t' \leqslant t$. Par application de la règle (INST) suivie de la règle (SUB), il vient que $G \vdash e' : t$.

Pour montrer $e \preceq e'$, il suffit donc de prouver que $G \vdash^0 e : T$ (**3**) implique $G \vdash e' : T$. Pour cela, nous considérons les différents cas possibles pour la règle R utilisée dans la dérivation $e \longrightarrow e'$.

• Si $R =$ (BETA), alors $e = (\mathtt{fun}\ x \Rightarrow e_1)\, e_2$ et $e' = [x \mapsto e_2]e_1$. La dérivation de (3) étant simple, on a $T = t_1$, $G \vdash e_2 : t_2$ (**4**) et $G \vdash \mathtt{fun}\ x \Rightarrow e_1 : t_2 \to t_1$. Par le lemme de simplification 4.2, il existe t'_1 et t'_2 tels que $t'_2 \to t'_1 \leqslant t_2 \to t_1$ (**5**) et $G \vdash^0 (\mathtt{fun}\ x \Rightarrow e_1) : t'_2 \to t'_1$ (**6**). La dérivation de (6) se terminant nécessairement par (FUN), on a $G[x : t'_2] \vdash e_1 : t'_1$ (**7**). En appliquant l'axiome (FLÈCHE-VARIANCE) à (5), il vient $t_2 \leqslant t'_2$ (**8**) et $t'_1 \leqslant t_1$ (**9**). En appliquant la règle (SUB) à (4) et (8), on a $G \vdash e_2 : t'_2$ (**10**). Par le lemme de substitution 4.9 appliqué à (7) et (10), on sait que $G \vdash [x \mapsto e_2]e_1 : t'_1$ (**11**). Enfin, par la règle (SUB) appliquée à (9) et (11), on conclut $G \vdash [x \mapsto e_2]e_1 : t_1$, c'est-à-dire $G \vdash e' : T$.

• Si $R =$ (BETA-LET), alors on a $e = (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)$ et $e' = [x \mapsto e_1]e_2$. La dérivation du jugement (3) étant simple, elle se termine forcément par (LET) et on a $G \vdash e_1 : T_1$ (**12**) et $G[x : T_1] \vdash e_2 : T$ (**13**). Par le lemme de substitution 4.9 appliqué à (12) et (13), on conclut $G \vdash [x \mapsto e_1]e_2 : T$, c'est-à-dire $G \vdash e' : T$.

• Si $R =$ (BETA-FIX), alors on a $e = \mathbf{fix}\ x \Rightarrow e_0$ et $e' = [x \mapsto e]e_0$. La dérivation simple du jugement (3) se terminant forcément par (FIX), on a $G[x : T] \vdash e_0 : T$ (**14**). Par le lemme de substitution 4.9 appliqué à (3) et (14), on conclut $G \vdash [x \mapsto e]e_0 : T$, c'est-à-dire $G \vdash e' : T$.

• Si $R =$ (BETA-OBJ), on a, l'indice i étant fixé, $e = C\cdot\ell_i\ (C'\{\ell_1 = e_1; \ldots; \ell_n = e_n\})$ et $e' = e_i$. La dérivation simple de (3) se terminant nécessairement par (APP), on a $T = t_i$, $G \vdash C\cdot\ell_i : t \to t_i$ (**15**) et $G \vdash C'\{\ell_1 = e_1; \ldots; \ell_n = e_n\} : t$ (**16**). En appliquant le lemme de simplification 4.2 à (15), on voit qu'il existe t'' et t_i'' tels que $t'' \to t_i'' \leqslant t \to t_i$ (**17**) et $G \vdash^0 C\cdot\ell_i : t'' \to t_i''$ (**18**). Cette dernière dérivation étant simple, elle termine forcément par la règle (GET) dont une des prémisses est $\overline{C\cdot\ell_i}(t'') = t_i''$ (**19**) dont on déduit aussi $t'' \in \overline{\#C}$ (**20**). Par ailleurs, en appliquant une nouvelle fois le lemme de simplification 4.2 cette fois à (16), on trouve t''' tel que $t''' \leqslant t$ (**21**) et $G \vdash^0 C'\{\ell_1 = e_1; \ldots; \ell_n = e_n\} : t'''$ (**22**). Cette dernière dérivation, étant simple, se termine forcément par la règle (OBJET) dont on déduit $G \vdash e_i : t_i'''$ (**23**) et $\overline{C'\cdot\ell_i}(t''') = t_i'''$ (**24**), d'où également $t''' \in \overline{\#C'}$ (**25**). En appliquant l'axiome (FLÈCHE-VARIANCE) à (17), on déduit $t \leqslant t''$ (**26**) et $t_i'' \leqslant t_i$ (**27**). Par transitivité de $\leqslant$ appliquée à (21) et (26), on sait donc que $t''' \leqslant t''$ (**28**). Par l'axiome (CLASSES-COVARIANTES) appliqué à (25), (20) et (28), on déduit que $C' \sqsubseteq C$ (**29**). Puis, par l'axiome (CHAMPS-COVARIANTS) appliqué à (29) et (28), on déduit $\overline{C'\cdot\ell_i}(t''') \leqslant \overline{C\cdot\ell_i}(t'')$ (**30**). En tenant compte des égalités (19) et (24), (30) peut se réécrire $t_i''' \leqslant t_i''$ (**31**). En appliquant la transitivité de $\leqslant$ à (31) et (27), il vient alors $t_i''' \leqslant t_i$ (**32**). Par la règle (SUB) appliquée à (23) et (32), on conclut $G \vdash e_i : t_i$, c'est-à-dire $G \vdash e' : T$.

• Si, $R =$ (BETA-PRIM), on a alors $e = f\langle a_1, \ldots, a_n\rangle$ (**33**), $a' = f(a_1, \ldots, a_n)$ (**34**) et $e' = a'$ (**35**). La dérivation simple de (3) se terminant nécessairement par (APP), on a $T = t'$ (**36**), $G \vdash f : t \to t'$ (**37**) et $G \vdash \langle a_1, \ldots, a_n\rangle : t$ (**38**). En appliquant le lemme 4.2 à (37), on trouve t_0 et t_0' tels que $G \vdash^0 f : t_0 \to t_0'$ (**39**) et $t_0 \to t_0' \leqslant t \to t'$ (**40**). La dérivation de (39) se terminant nécessairement par la règle (PRIM), on déduit, l'indice i prenant toutes les valeurs entre 1 et n, $t_0' \in \mathcal{A}^0$ (**41**), $\overline{\mathbf{Tuple}_n\cdot i}(t_0) = t_i^0$ (**42**) et $\forall a_1 \in \underline{t_1^0}, \ldots, a_n \in \underline{t_n^0}$, $f(a_1, \ldots, a_n) \in \underline{t_0'}$ (**43**). Par ailleurs, ré-appliquons le lemme de simplification 4.2 à (38) : il existe donc u tel que $u \leqslant t$ (**44**) et $G \vdash^0 \langle a_1, \ldots, a_n\rangle : u$ (**45**). La dérivation de (45) se terminant forcément par (OBJET), on déduit que $G \vdash a_i : u_i$ (**46**) et $\overline{\mathbf{Tuple}_n\cdot i}(u) = u_i$ (**47**) où l'indice i prend toutes les valeurs entre 1 et n. Appliquons maintenant l'axiome (FLÈCHE-VARIANCE) à (40). On déduit $t \leqslant t_0$ (**48**) et $t_0' \leqslant t'$ (**49**). Par transitivité de $\leqslant$ appliquée à (44) et (48), on sait $u \leqslant t_0$ (**50**). Par l'axiome (CHAMPS-COVARIANTS) appliqué à (50) et à la classe $\mathbf{Tuple}_n$, il vient $\overline{\mathbf{Tuple}_n\cdot i}(u) \leqslant \overline{\mathbf{Tuple}_n\cdot i}(t_0)$ (**51**). En tenant compte des égalités (42) et (47), (51) peut se réécrire $u_i \leqslant t_i^0$ (**52**). Par le lemme de typage des valeurs 4.4 appliqué à (46), on sait que $\overline{a_i} \leqslant u_i$ (**53**). Par transitivité de $\leqslant$ appliquée à (53) et (52), il vient $\overline{a_i} \leqslant t_i^0$ (**54**). Par définition de $\underline{t_i^0}$, il vient alors $a_i \in \underline{t_i^0}$ (**55**). L'une des prémisse de (43) appliquée à (55) nous permet de déduire $f(a_1, \ldots, a_n) \in \underline{t_0'}$, c'est-à-dire $a' \in \underline{t_0'}$ (**56**). Par définition de $\underline{t_0'}$, on sait alors que $\overline{a'} \leqslant t_0'$ (**57**). Par transitivité de $\leqslant$ appliquée à (57) et (49), on a $\overline{a'} \leqslant t'$ (**58**). Par (CST), on sait $G \vdash a' : \overline{a'}$ (**59**) et on conclut enfin par (SUB) appliquée à (59) et (58) que $G \vdash a' : t'$, c'est-à-dire $G \vdash e' : T$.

• Si $R =$ (BETA-METH), alors il existe un indice j tel que $e = (\mathbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\})\, e_0$ (**60**), $\mathsf{filtre}(e_0, \pi_j) = \mathsf{vrai}$ (**61**) et $e' = [e_0 \downarrow \pi_j]e_j$ (**62**). La dérivation simple de (3) se terminant nécessairement par (APP), on a $T = t_1$ (**63**), $G \vdash e_0 : t_0$ (**64**) et $G \vdash \mathbf{meth}\ \{\pi_1 \Rightarrow e_1; \ldots; \pi_k \Rightarrow e_k\} : t_0 \to t_1$ (**65**). On peut réappliquer le lemme de simplification 4.2 à ce dernier jugement pour en trouver une dérivation simple. Celle-ci se termine forcément par la règle METH et il existe t_0' et t_1' tels que $t_0' \to t_1' \leqslant t_0 \to t_1$ (**66**) et $G \vdash \pi_j \Rightarrow e_j : t_0' \to t_1'$ (**67**). Par la variance de la flèche appliqué à (66), on sait que $t_0 \leqslant t_0'$ (**68**) et $t_1' \leqslant t_1$ (**69**). En appliquant la règle (SUB) appliquée à (64) et (68), on sait que $G \vdash e_0 : t_0'$ (**70**). Appliquons maintenant le lemme de projection 4.11 à

ce dernier jugement en sachant que $\mathsf{filtre}(e_0, \pi_j) = \mathsf{vrai}$. On trouve qu'il existe t_0'' et G' tels que $t_0' \vdash \pi_j : t_0''; G'$ (**71**) et $G \vdash e_0 \downarrow \pi_j : G'$ (**72**). Or, dans les prémisses de la dérivation de (67), on a, entre autres $\left(t_0' \vdash \pi_j : t_0''; G'\right) \Rightarrow \left(G \oplus G' \vdash e_j : t_1'\right)$. On en déduit donc la partie droite de cette implication, c'est-à-dire $G \oplus G' \vdash e_j : t_1'$. On peut alors appliquer le lemme de substitution 4.9 à ce jugement et à (72) pour déduire $G \vdash [e_0 \downarrow \pi_j]e_j : t_1'$ (**73**). On conclut en appliquant la règle (SUB) à (73) et (69) que $G \vdash [e_0 \downarrow \pi_j]e_j : t_1$, c'est-à-dire que $G \vdash e' : T$. ■

4.6 À propos de l'induction sur les dérivations infinies

Dans toutes les preuves de ce chapitre, nous nous sommes permis de faire des inductions sur des dérivations potentiellement infinies. Le langage mathématique du discours que nous manipulons informellement a donc potentiellement un problème de fondement qui pourrait remettre en question tous nos résultats. En réalité, il n'en est rien et pour s'en convaincre complètement, nous allons expliciter dans le cadre du calcul des constructions inductives [CP90] un petit fragment du système de types et de la preuve de correction. La modélisation est faite dans l'assistant de preuve CoQ [CDT01].

Le fragment considéré contient en particulier la règle de typage (GEN), qui quantifie sur tous les éléments d'un ensemble éventuellement infini. On prouve formellement le lemme de simplification 4.2 :

```
(* On suppose donnée une algèbre de types *)
Parameter monotype: Set.
(* Voici sa relation de sous-typage *)
Parameter leq: monotype -> monotype -> Prop.

(* Voici les axiomes nécessaires pour ce fragment *)
Axiom leq_refl: (t: monotype) (leq t t).
Axiom leq_trans: (t,u,v: monotype) (leq t u) -> (leq u v) -> (leq t v).

(* L'ensemble des expression est laissé ouvert:
   on ne s'intéresse ici qu'au polymorphisme *)
Parameter expr: Set.

(* Cette relation contient toutes les règles de typage monomorphes:
   (APP), (FUN), etc. *)
Parameter well_typed_0: expr -> monotype -> Prop.

(* Un polytype est un ensemble quelconque de monotypes. On ne dit qu'il
   est non-vide que dans la règle Gen ci-dessous *)
Definition polytype := monotype -> Prop.

(* Cette fonction construit le polytype singleton à partir d'un monotype *)
Definition Singleton: monotype -> polytype :=
   [t: monotype][u: monotype] t=u.

(* Definition du système de types (sans environnement pour simplifier) *)
Inductive well_typed: expr -> polytype -> Prop :=
   (* Toute les règles "monomorphes" *)
   Other: (e: expr; t: monotype)
            (well_typed_0 e t) ->
```

```
            (well_typed e (Singleton t))

|  (* Règle d'instantiation *)
   Inst: (e: expr; T: polytype; t: monotype)
            (well_typed e T) ->
            (T t) ->
            (well_typed e (Singleton t))

|  (* Règle de subsomption *)
   Sub: (e: expr; t1, t2: monotype)
            (well_typed e (Singleton t2)) ->
            (leq t2 t1) ->
            (well_typed e (Singleton t1))

|  (* Règle de généralisation *)
   Gen: (e: expr; T: polytype)
            (* T est non-vide *)
            (EX t: monotype | (T t)) ->
            ((t: monotype) (T t)->(well_typed e (Singleton t))) ->
            (well_typed e T)
.

(* Lemme de simplification~(4.2) *)
Lemma Simpl: (e: expr; T: polytype)(well_typed e T) ->
             (t: monotype)(T t) ->
             (EX t0: monotype | (well_typed_0 e t0) /
(leq t0 t)).
Proof.
Intros e T Derivation.
NewInduction Derivation.

(* Règle Other *)
Intros. Red in H0; Rewrite <- H0. Split with t.
Split. Trivial. Apply leq_refl.

(* Règle Inst *)
Intros. Red in H0; Rewrite <- H0. Apply IHDerivation. Trivial.

(* Règle Sub *)
Intros u1 H1. Red in H1. Rewrite <- H1.
Cut (EX t0:monotype | (well_typed_0 e t0) /
(leq t0 t2)).
Intro. Elim H0. Intros. Split with x. Split. Tauto.
Apply leq_trans with u:=t2. Tauto. Trivial.
Apply IHDerivation. Red; Trivial.

(* Règle Gen *)
Intros. Trivial. Apply (H1 t H2 t). Red. Trivial.
Qed.
```

La preuve du théorème peut alors être vérifiée dans l'outil CoQ (ici dans sa version 7.2) :

```
Welcome to Coq 7.2 (December 2001)
```

```
Coq < Load preuve.

Coq < Check Simpl.
Simpl
     : (e:expr; T:polytype)
        (well_typed e T)
        ->(t:monotype)
           (T t)->(EX t0:monotype | (well_typed_0 e t0)/\(leq t0 t))
```

Chapitre 5

Annotations et vérification de types

Sauf cas particulier, le système de types présenté au chapitre 3 est indécidable. D'abord à cause de la récursion (expression **`fix`**) dont la règle de typage est polymorphe. Cela est connu pour être une source d'indécidabilité dans le typage de ML[KTU93] : en présence de récursion polymorphe, le typage de ML est équivalent à un problème de semi-unification indécidable plutôt qu'à un problème simple d'unification. Ensuite parce que le typage des méthodes est probablement aussi difficile. Indépendamment de la décidabilité, il n'est de toute façon pas souhaitable d'inférer le type d'une méthode si les cas de définition sont écrits dans des modules différents, ce qui est l'un des objectifs des méthodes. Par ailleurs, une méthode est souvent naturellement récursive et la récursion polymorphe est intéressante dans certains cas. Par exemple, pour définir une méthode générale d'impression d'objet, le cas de définition sur les couples donnera naturellement lieu à un appel récursif par composante du couple, c'est-à-dire deux invocations sur des types *a priori* non reliés.

C'est pourquoi, il est nécessaire de demander au programmeur d'*annoter* le programme pour guider le typage et exprimer son intention, par exemple concernant le type des méthodes. Dans ce chapitre, nous définissons le langage utilisé pour écrire les annotations de type et nous donnons la syntaxe des expressions annotées. Nous introduisons ensuite un système de types pour ces expression. Ce système fonctionne en restreignant certaines des règles du système algébrique à des types exprimables dans le langage des annotations. Nous nous limitons au cas de la vérification de types, c'est-à-dire que le programmeur doit donner le type des méthodes et des définitions récursives, mais aussi le type complet des fonctions. La raison de cette limitation est que l'inférence de types est strictement plus complexe que la vérification et nous ne ferons que l'aborder dans la conclusion de cette thèse.

Voici comment est organisé ce chapitre : dans la section 5.1, nous définissons le langage de types utilisé pour écrire les annotations. Celui-ci est un langage du premier ordre standard. Dans la section 5.2, nous en donnons la sémantique en utilisant un modèle du langage du premier ordre dans l'algèbre de types utilisé pour le typage. Dans la section 5.3, nous introduisons et justifions la syntaxe des expressions annotées. Nous en donnons une sémantique par effacement des annotations. Section 5.4, nous donnons les règles de typage des expressions annotées et nous en montrons la correction par rapport au typage algébrique.

5.1 Langage de types

On suppose donné une fois pour toutes un ensemble infini $\mathsf{VarsType}$ de VARIABLES DE TYPE. On utilisera les lettres α, β, etc. pour désigner les variables de types.

Un LANGAGE DE TYPES $\mathcal{L}$ est un langage logique du premier ordre, formé donc d'un ensemble de symboles de constructeurs de termes et d'un ensemble de symboles de prédicats. Les symboles constructeurs de termes sont aussi appelés CONSTRUCTEURS DE TYPES et sont notés avec les lettres c, c', etc. Les symboles de prédicat ou PRÉDICATS DE TYPES sont notés avec les lettres p, p', etc. L'ensemble de tous les symboles d'un langage sera supposé dénombrable ou fini.

Chaque constructeur de types et chaque prédicat de types vient avec son arité, notée $|c|$ et $|p|$. L'arité d'un constructeur de types est le nombre de sous-termes auquel il peut être appliqué pour former un terme. Cette arité peut être nulle, auquel cas le constructeur est un symbole de constante. L'arité d'un prédicat de types est le nombre de termes auquel il peut être appliqué pour former une contrainte atomique.

Les termes construits sur l'alphabet des constructeurs de types sont appelés MONOTYPES SYNTAXIQUES. On supposera prédéfini le constructeur de types fonctionnels $\rightarrow$. L'ensemble des monotypes syntaxiques est donc engendré par la grammaire suivante :

$\tau ::= \alpha$	Variable de types
$\mid \tau \rightarrow \tau$	Type fonctionnel
$\mid c\langle \underbrace{\tau, \ldots, \tau}_{\lvert c\rvert \text{ termes}} \rangle$	Type construit

Une CONTRAINTE est une conjonction finie de contraintes atomiques formées avec un symbole de prédicat du langage ou bien le prédicat prédéfini de sous-typage $\leqslant$. La contrainte vide est notée vrai. On se donne également la possibilité de masquer certaines variables d'une contrainte en utilisant le connecteur logique $\exists$. L'ensemble des contraintes est donc engendré par la grammaire suivante :

$\kappa ::= \mathsf{vrai}$	Contrainte vide
$\mid \tau \leqslant \tau$	Contrainte de sous-typage
$\mid p\langle \underbrace{\tau, \ldots, \tau}_{\lvert p\rvert \text{ termes}} \rangle$	Application d'un prédicat du langage
$\mid \kappa \wedge \kappa$	Conjonction
$\mid \exists \mathcal{V} \,.\, \kappa$	Contrainte existentielle

Dans cette dernière production, $\mathcal{V}$ désigne un ensemble fini de variables de type, notation que nous utiliserons constamment par la suite.

Remarque. Les contraintes de la forme $\exists \mathcal{V} \,.\, \kappa$ sont très utiles pour gérer les variables fraîches introduites quand on fabrique des contraintes dans un algorithme de typage. Elles peuvent également se révéler intéressantes pour le programmeur pour exprimer des contraintes incluant des variables « anonymes », comme par exemple l'existence d'un super-type commun à deux types. ■

Enfin, un POLYTYPE SYNTAXIQUE ou SCHÉMA DE TYPE est de la forme suivante :

$$\sigma ::= \forall \mathcal{V} \mid \kappa \,.\, \tau$$

Intuitivement, un schéma de type représente un polytype algébrique contenant toutes les instances de τ quand les variables $\mathcal{V}$ prennent des valeurs satisfaisant à la contrainte κ. En général, les variables $\mathcal{V}$ apparaissent dans la contrainte κ ou dans le monotype τ, mais on peut aussi y trouver d'autres variables, qui sont alors considérées comme libres.

Exemple. Un langage de types possible est dérivé des déclarations de classes entrées par le programmeur. Pour chaque classe C paramétrée par n types, on peut supposer qu'il correspond un constructeur de type également noté C et d'arité n. Par ailleurs, on peut supposer qu'il existe un symbole de constante (un constructeur d'arité 0) pour chaque type primitif. Dans ces conditions, voici quelques exemples de monotypes syntaxiques : $\mathsf{Cons}\langle\alpha\rangle$, $\alpha \to \mathsf{int}$, $\mathsf{int} \to \mathsf{int}$, etc. de contraintes : $\alpha \leqslant \mathsf{rat}$, $\mathsf{Cons}\langle\alpha\rangle \leqslant \alpha$, etc. et de schémas de types : $\forall\alpha \mid \mathsf{vrai} \,.\, \alpha \to \alpha$, $\forall\alpha \mid \alpha \leqslant \mathsf{rat} \,.\, \mathtt{Tuple}_2\langle\alpha, \alpha\rangle \to \alpha$, etc.

On se permettra en général d'omettre d'écrire la contrainte d'un schéma si elle est égale à vrai, comme dans $\forall\alpha \,.\, \alpha \to \alpha$. On identifiera également un schéma où l'ensemble des variables et la contrainte sont vides avec son monotype, chaque fois qu'il n'y a pas d'ambiguïté possible. Par exemple, on identifiera $\forall\varnothing \mid \mathsf{vrai} \,.\, \mathsf{int}$ et int.

Un langage de types n'est pas forcément limité à ces constructeurs de types. Par exemple, on peut imaginer qu'un certain langage définisse un opérateur $\sqcup$ sur les types, de sorte qu'on puisse former le monotype $\tau_1 \sqcup \tau_2$. On peut également trouver dans un langage des prédicats, par exemple un prédicat *Imprimable* d'arité 1, avec lequel on puisse former le type $\forall\alpha \mid \mathit{Imprimable}\langle\alpha\rangle \,.\, \alpha \to \mathsf{string}$.

Notons que, pour l'instant, nous ne définissons que la syntaxe des types, pas leur sémantique. C'est pourquoi, l'opérateur $\sqcup$ et le prédicat *Imprimable* n'ont pas de signification particulière. ■

Variables de types libres

L'ensemble des variables de type libres d'un monotype τ est noté $\mathit{ftv}(\tau)$. Cet ensemble est défini comme suit :

$$\begin{aligned}
\mathit{ftv}(\alpha) &\triangleq \{\alpha\} \\
\mathit{ftv}(\tau_1 \to \tau_2) &\triangleq \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2) \\
\mathit{ftv}(c\langle\tau_1, \ldots, \tau_n\rangle) &\triangleq \mathit{ftv}(\tau_1) \cup \cdots \cup \mathit{ftv}(\tau_n)
\end{aligned}$$

De même, les variables libres d'une contrainte sont :

$$\begin{aligned}
\mathit{ftv}(\mathsf{vrai}) &\triangleq \varnothing \\
\mathit{ftv}(\tau_1 \leqslant \tau_2) &\triangleq \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2) \\
\mathit{ftv}(p\langle\tau_1, \ldots, \tau_n\rangle) &\triangleq \mathit{ftv}(\tau_1) \cup \cdots \cup \mathit{ftv}(\tau_n) \\
\mathit{ftv}(\kappa_1 \wedge \kappa_2) &\triangleq \mathit{ftv}(\kappa_1) \cup \mathit{ftv}(\kappa_2) \\
\mathit{ftv}(\exists\mathcal{V} \,.\, \kappa) &\triangleq \mathit{ftv}(\kappa) \setminus \mathcal{V}
\end{aligned}$$

Et celles d'un schéma de type :

$$\mathit{ftv}(\forall\mathcal{V} \mid \kappa \,.\, \tau) \triangleq (\mathit{ftv}(\tau) \cup \mathit{ftv}(\kappa)) \setminus \mathcal{V}$$

5.2 Modèles

Un modèle d'un langage de types fournit une interprétation de chaque élément du langage dans une algèbre de types, domaine d'interprétation. Formellement, un modèle $\mathcal{M}$ d'un langage $\mathcal{L}$ est la donnée des éléments suivants :

- une algèbre de types domaine d'interprétation, notée $[\![\mathcal{M}]\!]$;
- pour chaque symbole de constructeur c d'arité n, une fonction n-aire totale $[\![c]\!]$ opérant dans $[\![\mathcal{M}]\!]$, c'est-à-dire un élément de $[\![\mathcal{M}]\!]^n \to [\![\mathcal{M}]\!]$;
- pour chaque symbole de prédicat p d'arité n, un prédicat n-aire $[\![c]\!]$ opérant dans $[\![\mathcal{M}]\!]$, c'est-à-dire un sous-ensemble de $[\![\mathcal{M}]\!]^n$.

Une fois données les interprétations des éléments de base du langage, on peut étendre ces interprétations à tous les autres constructions syntaxiques introduites dans la section précédente. Comme celles-ci peuvent contenir des variables de types libres, il faut d'abord en donner la valeur. Pour cela, on appelle VALUATION une fonction partielle à domaine finie des variables de types VarsType vers l'algèbre $\mathcal{A}$. On utilise les lettres ρ, ρ', $\hat{\rho}$, etc.[1] pour désigner les valuations.

L'interprétation d'un monotype syntaxique τ dans une valuation $\hat{\rho}$ dont le domaine contient au moins les variables de type libres de τ est un monotype de l'algèbre $[\![\mathcal{M}]\!]$ noté ${}_{\hat{\rho}}[\![\tau]\!]$.[2] Cette interprétation est définie par les équations suivantes :

$$
\begin{array}{rcll}
{}_{\hat{\rho}}[\![\alpha]\!] & \triangleq & \hat{\rho}(\alpha) & \text{(on a forcément } \alpha \in \mathrm{dom}(\hat{\rho})\text{)} \\
{}_{\hat{\rho}}[\![\tau_1 \to \tau_2]\!] & \triangleq & {}_{\hat{\rho}}[\![\tau_1]\!] \to {}_{\hat{\rho}}[\![\tau_2]\!] & \\
{}_{\hat{\rho}}[\![c\langle \tau_1, \ldots, \tau_n \rangle]\!] & \triangleq & [\![c]\!]({}_{\hat{\rho}}[\![\tau_1]\!], \ldots, {}_{\hat{\rho}}[\![\tau_n]\!]) &
\end{array}
$$

L'interprétation d'une contrainte est une valeur de vérité définie par les équivalences suivantes :

$$
\begin{array}{rll}
{}_{\hat{\rho}}[\![\mathsf{vrai}]\!] & \text{est vrai} & \\
{}_{\hat{\rho}}[\![\tau_1 \leqslant \tau_2]\!] & \text{ssi} & {}_{\hat{\rho}}[\![\tau_1]\!] \leqslant {}_{\hat{\rho}}[\![\tau_2]\!] \\
{}_{\hat{\rho}}[\![p\langle \tau_1, \ldots, \tau_n \rangle]\!] & \text{ssi} & ({}_{\hat{\rho}}[\![\tau_1]\!], \ldots, {}_{\hat{\rho}}[\![\tau_n]\!]) \in [\![p]\!] \\
{}_{\hat{\rho}}[\![\kappa_1 \wedge \kappa_2]\!] & \text{ssi} & {}_{\hat{\rho}}[\![\kappa_1]\!] \text{ et } {}_{\hat{\rho}}[\![\kappa_2]\!] \text{ sont vrais} \\
{}_{\hat{\rho}}[\![\exists \mathcal{V} \,.\, \kappa]\!] & \text{ssi} & \text{il existe } \rho \text{ tel que } \mathrm{dom}(\rho) = \mathcal{V} \text{ et } {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!] \text{ est vrai}
\end{array}
$$

Remarque. Dans cette dernière équation, notons que $\mathrm{dom}(\hat{\rho})$ n'est pas forcément disjoint de $\mathrm{dom}(\rho)$, ce qui signifie que certaines variables de $\mathcal{V}$ peuvent masquer celles définies par $\hat{\rho}$. Nous aurions pu supposer que les variables de type sont systématiquement renommées pour éviter les masquages. Une telle hypothèse est une simplification commode, mais risque d'introduire un certain manque de rigueur. Nous avons donc préféré traiter complètement le problème des masquages, sans supposer une α-conversion implicite. ■

L'interprétation d'un schéma de types $\forall \mathcal{V} \,|\, \kappa \,.\, \tau$ est l'ensemble, éventuellement vide, de monotypes algébriques que décrit l'interprétation de τ quand $\mathcal{V}$ varie dans les solutions de κ :

$$
{}_{\hat{\rho}}[\![\forall \mathcal{V} \mid \kappa \,.\, \tau]\!] \quad \triangleq \quad \left\{ {}_{\hat{\rho}\oplus\rho}[\![\tau]\!] \mid \mathrm{dom}(\rho) = \mathcal{V} \text{ et } {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!] \right\}
$$

[1] La notation $\hat{\rho}$ forme un tout, le chapeau étant utilisé au même titre qu'un prime ou un indice.

[2] La notation utilisée ici est l'inverse de la notation habituelle $[\![\tau]\!]_{\hat{\rho}}$. Notre notation est légèrement plus pratique car elle s'associe bien avec l'opérateur de composition $\circ$ sur les valuations. La notation ${}_{\hat{\rho}}[\![\tau]\!]$ doit en fait se lire « $\hat{\rho}$ appliqué à τ ».

Notons que pour que l'interprétation d'un schéma de types soit non-vide, c'est-à-dire constitue un polytype valide pour le système de types algébrique, il faut et il suffit que la contrainte $\exists \mathcal{V} \,.\, \kappa$ soit vraie dans la valuation $\hat{\rho}$. Dans ce cas, on dit que le schéma de type est SATISFIABLE.

Notations

Quand le modèle $\mathcal{M}$ utilisé pour interpréter un élément syntaxique n'est pas implicitement connu, on utilisera la notation ${}_{\hat{\rho}}[\![\cdot]\!]^{\mathcal{M}}$ à la place de ${}_{\hat{\rho}}[\![\cdot]\!]$. De même, si la valuation $\hat{\rho}$ est de domaine vide, c'est-à-dire ne définit aucune variable de type, on se permettra d'omettre de l'écrire et on notera simplement $[\![\cdot]\!]$ ou bien $[\![\cdot]\!]^{\mathcal{M}}$ si on veut expliciter le modèle utilisé.

Exemple. Voici un exemple de modèle. Considérons le langage de types dérivé des déclarations du programmeur et tel qu'à chaque classe C correspond un constructeur de type C.

Un modèle possible pour ce langage est celui constitué de tous les termes finis construits avec l'alphabet des constructeurs et avec la flêche et ordonné par la relation de sous-typage structurelle engendrée par les variances déclarées par le programmeur (algèbre structurelle simple). L'algèbre structurelle simple n'est bien sûr pas le seul modèle possible pour ce langage : on peut aussi considérer le modèle des types infinis ou bien le plus petit treillis contenant l'ordre structurel. Dans le modèle treillis, l'opérateur supplémentaire $\sqcup$ peut être interprété comme l'opérateur de borne supérieure.

Si le langage contenu le prédicat *Imprimable*, celui-ci peut être interprété par le prédicat algébrique défini comme suit : $\mathit{Imprimable}\langle t \rangle$ est vrai pour tout les types primitifs t. $\mathit{Imprimable}\langle C\langle t \rangle\rangle$ est vrai si et seulement si C vaut Cons, Nil ou List et t est lui-même imprimable.

Nous voyons donc que notre cadre autorise une grande souplesse dans le choix de la syntaxe des types comme dans celui de son interprétation et tous les résultats de cette thèse (correction du typage, réduction à des problèmes de contraintes) en sont indépendants, ce qui était l'un des objectifs recherchés. ■

5.3 Le langage annoté

Dans cette section, nous introduisons un nouveau langage d'expressions. Il dérive du langage non-typé présenté au chapitre 1 par le fait qu'il contient des annotations de types explicites sur les définitions récursives, les méthodes et l'argument des fonctions. L'introduction de ce nouveau langage permet au programmeur de guider le typage des expressions récursives polymorphes et d'exprimer son intention concernant le type des méthodes, ce qui est nécessaire pour aboutir à un système de types décidables.

5.3.1 Syntaxe des expressions annotées

Les expressions annotées sont notées avec la lettre ε et leur grammaire coïncide avec celle des expressions non-typées sauf sur certaines production. Elles sont engendrées par la grammaire suivante :

$$
\begin{aligned}
\varepsilon ::= {} & x \\
& \mid \varepsilon_1\, \varepsilon_2 \\
& \mid \textbf{fun}\, x : \tau \;\Rightarrow \varepsilon \\
& \mid \textbf{let}\, x = \varepsilon_1\, \textbf{in}\, \varepsilon_2 \\
& \mid \textbf{fix}\, x : \sigma \;\Rightarrow \varepsilon \\
& \mid C\,\{\ell_1 = \varepsilon_1; \ldots; \ell_n = \varepsilon_n\} \\
& \mid C \cdot \ell \\
& \mid a \\
& \mid f \\
& \mid \textbf{meth}\colon \tau \to \tau'\,\{\pi_1 \Rightarrow \varepsilon_1; \ldots; \pi_k \Rightarrow \varepsilon_k\} \\
& \mid \forall \mathcal{V} \mid \kappa \,.\, \varepsilon \qquad \text{(Introduction de variables de types)}
\end{aligned}
$$

Les définitions récursives polymorphes sont annotées avec un schéma de types qui représente le type polymorphe de l'expression attendu ou deviné par le programmeur. Celui-ci peut être correct ou non, ce que nous devrons vérifier lors du typage.

L'expression $\forall \mathcal{V} \mid \kappa \,.\, \varepsilon$ introduit les variables de types $\mathcal{V}$ contraintes par κ et qui peuvent être utilisées par les annotations de ε. La raison de cette construction est qu'il est souvent nécessaire d'écrire des annotations qui contiennent des variables libres. Par exemple, la méthode qui renverse les éléments d'une liste possède naturellement le type $\forall \alpha \,.\, \mathsf{List}\langle\alpha\rangle \to \mathsf{List}\langle\alpha\rangle$, mais l'implémentation efficace habituelle en temps linéaire utilise en interne une fonction à deux arguments (un accumulateur et la liste), donc d'un type devant nécessairement utiliser α.

L'expression $\textbf{fun}\, x : \tau \;\Rightarrow \varepsilon$ est une fonction dont le programmeur spécifie que le type de l'argument est égal à τ. Ce monotype peut contenir des variables de types, de sorte qu'on peut écrire par exemple l'identité polymorphe comme : $\forall \alpha \mid \mathsf{vrai} \,.\, \textbf{fun}\, x : \alpha \;\Rightarrow x$. Notons dans cette dernière expression que le corps de la fonction doit accepter un argument x de tout type. L'annotation n'est pas une simple contrainte posée sur le type de l'argument, mais est une spécification complète du domaine de la fonction. Par exemple, la fonction $\forall \alpha \mid \mathsf{vrai} \,.\, \textbf{fun}\, x : \alpha \;\Rightarrow x + 1$ sera mal typée, car son corps n'accepte pas un argument booléen, alors qu'on a spécifié que le type bool appartient au domaine de la fonction. Pour qu'elle soit bien typée, le programmeur devra raffiner les annotations et écrire par exemple $\forall \alpha \mid \alpha \leqslant \mathsf{rat} \,.\, \textbf{fun}\, x : \alpha \;\Rightarrow x + 1$. C'est en ce sens que nous ne parlerons dans ce chapitre que de la vérification de types et pas de l'inférence.

L'expression $\textbf{meth}\colon \tau \to \tau'\,\{\pi_1 \Rightarrow \varepsilon_1; \ldots; \pi_k \Rightarrow \varepsilon_k\}$ est une méthode dont on a spécifié explicitement le type. Par simplification, nous restreignons l'annotation de type à être syntaxiquement un type fonctionnel. De même, nous nous limitons à une annotation monomorphe, étant donné qu'on peut toujours écrire $\forall \mathcal{V} \mid \kappa \,.\, \textbf{meth}\colon \tau \to \tau'\,\{\pi_1 \Rightarrow \varepsilon_1; \ldots; \pi_k \Rightarrow \varepsilon_k\}$ pour donner le type polymorphe $\forall \mathcal{V} \mid \kappa \,.\, \tau \to \tau'$ à la méthode.

Variables de type libres

L'ensemble des variables de type libres d'une expression annotée est défini comme suit :

$$ftv(x) \quad \triangleq \quad \varnothing$$

$$
\begin{aligned}
ftv(\varepsilon_1\,\varepsilon_2) &\triangleq ftv(\varepsilon_1) \cup ftv(\varepsilon_2)\\
ftv(\mathbf{fun}\,x\colon\tau\;\Rightarrow\varepsilon) &\triangleq ftv(\tau) \cup ftv(\varepsilon)\\
ftv(\mathbf{let}\,x=\varepsilon_1\,\mathbf{in}\,\varepsilon_2) &\triangleq ftv(\varepsilon_1) \cup ftv(\varepsilon_2)\\
ftv(\mathbf{fix}\,x\colon\sigma\;\Rightarrow\varepsilon) &\triangleq ftv(\sigma) \cup ftv(\varepsilon)\\
ftv(C\,\{\ell_1=\varepsilon_1;\ldots;\ell_n=\varepsilon_n\}) &\triangleq ftv(\varepsilon_1) \cup \cdots \cup ftv(\varepsilon_n)\\
ftv(C\cdot\ell) &\triangleq \varnothing\\
ftv(a) &\triangleq \varnothing\\
ftv(f) &\triangleq \varnothing\\
ftv(\mathbf{meth}\colon\tau\to\tau'\,\{\pi_1\Rightarrow\varepsilon_1;\ldots;\pi_k\Rightarrow\varepsilon_k\}) &\triangleq ftv(\tau) \cup ftv(\tau') \cup ftv(\varepsilon_1) \cup \cdots \cup ftv(\varepsilon_k)\\
ftv(\forall\mathcal{V}\mid\kappa\,.\,\varepsilon) &\triangleq (ftv(\varepsilon) \cup ftv(\kappa)) \setminus \mathcal{V}
\end{aligned}
$$

5.3.2 Sémantique

La sémantique d'une expression annotée ε est celle de l'expression non annotée obtenue en effaçant toutes les annotations dans ε. Cette expression non-typée est notée $(\!|\varepsilon|\!)$. Cette opération rend bien compte du fait que les annotations de type ne servent que pour le typage mais ne sont pas utilisées à l'exécution.

5.4 Vérification de types semi-algébrique

Nous introduisons maintenant un système de types semi-algébrique pour les expressions annotées. Le système est essentiellement algébrique, mais il utilise un modèle du langage de types pour interpréter les annotations et diriger le système de types algébrique du chapitre 3.

Dans cette section, nous supposons donné un modèle $\mathcal{M}$.

Le jugement de typage s'écrit $\hat{\rho}; G \vdash \varepsilon : T$ et se lit « en utilisant la valuation $\hat{\rho}$ pour interpréter les annotations de types de ε, et dans l'environnement de typage G, ε a pour type T ». On impose que le domaine de $\hat{\rho}$ contienne au moins les variables de types libres de ε.

Examinons les règles de ce système. La règle de typage des expressions récursives est similaire à (FIX) excepté que le polytype utilisé doit correspondre à l'annotation donnée par le programmeur. L'interprétation de celle ci doit donc être non-vide et on écrit :

$$
(\textsc{Fix}_A)\qquad \frac{{}_{\hat{\rho}}[\![\sigma]\!] \neq \varnothing \qquad \hat{\rho}; G[x\colon {}_{\hat{\rho}}[\![\sigma]\!]] \vdash \varepsilon : {}_{\hat{\rho}}[\![\sigma]\!]}{\hat{\rho}; G \vdash (\mathbf{fix}\,x\colon\sigma\;\Rightarrow\varepsilon) : {}_{\hat{\rho}}[\![\sigma]\!]}
$$

La règle de typage des fonctions contraint le type de l'argument de la fonction à être celui spécifié par le programmeur :

$$
(\textsc{Fun}_A)\qquad \frac{\hat{\rho}; G[x\colon {}_{\hat{\rho}}[\![\tau]\!]] \vdash \varepsilon : t'}{\hat{\rho}; G \vdash \mathbf{fun}\,x\colon\tau\;\Rightarrow\varepsilon : {}_{\hat{\rho}}[\![\tau]\!] \to t'}
$$

De même, celle concernant les méthodes s'écrit naturellement :

$$(\text{METH}_A) \qquad \frac{{}_{\hat{\rho}}[\![\tau \to \tau']\!] \bowtie \pi_1; \ldots; \pi_k \qquad \hat{\rho}; G \vdash \pi_i \Rightarrow \varepsilon_i : {}_{\hat{\rho}}[\![\tau \to \tau']\!]}{\hat{\rho}; G \vdash \texttt{meth}\colon \tau \to \tau' \{\pi_1 \Rightarrow \varepsilon_1; \ldots; \pi_k \Rightarrow \varepsilon_k\} : {}_{\hat{\rho}}[\![\tau \to \tau']\!]}$$

Dans cette règle, le typage des cas se fait exactement comme dans le système algébrique, excepté qu'on transmet la valuation $\hat{\rho}$ pour le typage des corps des cas.

Examinons finalement la règle pour le typage de l'introduction de variables. Considérons une expression de la forme $\forall \mathcal{V} \mid \kappa \,.\, \varepsilon$. On demande tout d'abord que $\exists \mathcal{V} \,.\, \kappa$ soit satisfiable dans $\hat{\rho}$. Dans le cas contraire, on considère que l'annotation est tout simplement mal typée. Ensuite, on procède au typage de l'expression ε, ce qui produit un polytype T_ρ, forcément non-vide. On donne alors à l'expression annotée toute entière le type union de tous les T_ρ, ce qui est une forme particulière de généralisation :

$$(\text{INTRO}_A) \qquad \frac{{}_{\hat{\rho}}[\![\exists \mathcal{V} \,.\, \kappa]\!] \qquad \forall \rho, \left(\mathrm{dom}(\rho) = \mathcal{V} \wedge {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!]\right) \Rightarrow \hat{\rho} \oplus \rho; G \vdash \varepsilon : T_\rho}{\hat{\rho}; G \vdash \forall \mathcal{V} \mid \kappa \,.\, \varepsilon : \bigcup\limits_{\substack{\mathrm{dom}(\rho)=\mathcal{V} \\ {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!]}} T_\rho}$$

Remarque. Dans cette règle, il peut très bien y avoir un ensemble infini de solutions ρ et donc un infinité de polytypes $T_V al$ dont il faut prendre l'union. Cette infinité de prémisses ne pose pas plus de problème que pour la règle algébrique (GEN). ■

Les autres règles sont formellement identiques à celles du système algébrique. Pour une présentation complète de toutes les règles, voir en section 5.6, page 86.

5.5 Sûreté du système de vérification semi-algébrique

THÉORÈME 2 (SÛRETÉ DU TYPAGE SEMI-ALGÉBRIQUE). *Si* $\hat{\rho}; G \vdash \varepsilon : T$*, alors* $G \vdash (\!|\varepsilon|\!) : T$.

Démonstration. Chacune des nouvelles règles de typage semi-algébrique est une spécialisation des règles de typage algébriques, excepté (INTRO_A) qui est une instance de (GEN) précédée de (INST). ■

5.6 Récapitulatif

Voici la liste complète des règles du système de vérification de types semi-algébrique opérant sur les expressions annotées :

$$(\text{FIX}_A) \qquad \frac{{}_{\hat{\rho}}[\![\sigma]\!] \neq \varnothing \qquad \hat{\rho}; G[x : {}_{\hat{\rho}}[\![\sigma]\!]] \vdash \varepsilon : {}_{\hat{\rho}}[\![\sigma]\!]}{\hat{\rho}; G \vdash \texttt{fix}\, x : \sigma \;\Rightarrow \varepsilon : {}_{\hat{\rho}}[\![\sigma]\!]}$$

$$(\text{FUN}_A) \qquad \frac{\hat{\rho}; G[x : {}_{\hat{\rho}}[\![\tau]\!]] \vdash \varepsilon : t'}{\hat{\rho}; G \vdash \texttt{fun}\, x : \tau \;\Rightarrow \varepsilon : {}_{\hat{\rho}}[\![\tau]\!] \to t'}$$

(METH$_A$)
$$\frac{{}_{\hat\rho}[\![\tau \to \tau']\!] \bowtie \pi_1; \ldots; \pi_k \qquad \hat\rho; G \vdash \pi_i \Rightarrow \varepsilon_i : {}_{\hat\rho}[\![\tau \to \tau']\!]}{\hat\rho; G \vdash \mathbf{meth}\colon \tau \to \tau'\,\{\pi_1 \Rightarrow \varepsilon_1; \ldots; \pi_k \Rightarrow \varepsilon_k\} : {}_{\hat\rho}[\![\tau \to \tau']\!]}$$

(INTRO$_A$)
$$\frac{{}_{\hat\rho}[\![\exists \mathcal{V} \,.\, \kappa]\!] \qquad \forall \rho, \left(\mathrm{dom}(\rho) = \mathcal{V} \wedge {}_{\hat\rho\oplus\rho}[\![\kappa]\!]\right) \Rightarrow \hat\rho \oplus \rho; G \vdash \varepsilon : T_\rho}{\hat\rho; G \vdash \forall \mathcal{V} \mid \kappa \,.\, \varepsilon : \bigcup\limits_{\substack{\mathrm{dom}(\rho)=\mathcal{V} \\ {}_{\hat\rho\oplus\rho}[\![\kappa]\!]}} T_\rho}$$

(VAR$_A$)
$$\hat\rho; G \vdash x : G(x)$$

(APP$_A$)
$$\frac{\hat\rho; G \vdash \varepsilon_1 : t_2 \to t \qquad \hat\rho; G \vdash \varepsilon_2 : t_2}{\hat\rho; G \vdash e_1\, e_2 : t}$$

(OBJET$_A$)
$$\frac{t \in \overline{\#C} \qquad \hat\rho; G \vdash \varepsilon_1 : \overline{C\cdot\ell_1}(t) \qquad \ldots \qquad \hat\rho; G \vdash \varepsilon_n : \overline{C\cdot\ell_n}(t)}{\hat\rho; G \vdash C\,\{\ell_1 = \varepsilon_1; \ldots; \ell_n = \varepsilon_n\} : t}$$

(GET$_A$)
$$\frac{t' = \overline{C\cdot\ell}(t)}{\hat\rho; G \vdash C\cdot\ell : t \to t'}$$

(CONSTANTE$_A$)
$$\frac{}{\hat\rho; G \vdash a : \overline{a}}$$

(PRIMITIVE$_A$)
$$\frac{\forall a_1 \in \underline{t_1^0}, \ldots, a_n \in \underline{t_n^0},\ f(a_1, \ldots, a_n) \in \underline{t^0} \qquad \forall i, \overline{\mathbf{Tuple}_n\cdot i}(t) = t_i^0}{\hat\rho; G \vdash f : t \to t^0}$$

(INST)
$$\frac{\hat\rho; G \vdash \varepsilon : T \qquad t \in T}{\hat\rho; G \vdash \varepsilon : t}$$

(GEN$_A$)
$$\frac{T \neq \varnothing \qquad \text{pour tout } t \in T,\ \hat\rho; G \vdash \varepsilon : t}{\hat\rho; G \vdash \varepsilon : T}$$

(SUB$_A$)
$$\frac{\hat\rho; G \vdash \varepsilon : t \qquad t \leqslant t'}{\hat\rho; G \vdash \varepsilon : t'}$$

(LET$_A$)
$$\frac{\hat\rho; G \vdash \varepsilon_1 : T_1 \qquad \hat\rho; G[x : T_1] \vdash \varepsilon_2 : T_2}{\hat\rho; G \vdash \mathbf{let}\ x = \varepsilon_1\ \mathbf{in}\ \varepsilon_2 : T_2}$$

(CAS$_A$)
$$\frac{\forall t'' G' (t \vdash \pi : t''; G') \Rightarrow \hat\rho; G \oplus G' \vdash \varepsilon : t'}{\hat\rho; G \vdash \pi \Rightarrow \varepsilon : t \to t'}$$

Chapitre 6

De la vérification de types aux contraintes

Récapitulons notre parcours jusqu'à présent. Nous avons introduit un langage et sa sémantique non typée (chapitres 1 et 2). Puis, nous avons donné un système de types travaillant sur des monotypes pris dans une algèbre et vérifiant un petit nombre d'axiomes (chapitre 3). Nous avons ensuite montré la correction de ce système de types vis-à-vis de la sémantique opérationnelle à l'aide d'une preuve générique (chapitre 4). Ce système de types n'est pas directement utilisable en pratique, ne serait-ce que parce que certaines constructions du langage, comme la récursion polymorphe, ne peuvent être décidablement typées. En revanche, sa preuve de correction générique en fait un outil pour montrer la correction d'autres systèmes plus pratiques. Au chapitre 5, nous avons introduit un nouveau langage dont les expressions sont munies d'annotations de types exprimées dans un langage de types et dont la sémantique opérationnelle est obtenue par effacement des annotations. Nous en avons également donné un système de types dont nous avons prouvé la correction par réécriture dans le système algébrique. Ce système, que nous avons appelé semi-algébrique, est un système de vérification de types car le programmeur doit spécifier complètement le type des arguments des fonctions. Il se rapproche du système algébrique car il n'est pas dirigé par la syntaxe, le polymorphisme y est extensionnel et les dérivations peuvent être infinies. À partir de maintenant, nous allons considérer le système semi-algébrique comme la spécification d'un jugement de typage et nous allons étudier comment en donner une implémentation.

Dans ce chapitre, nous réalisons une partie du chemin dans cette voie. Le problème que nous traitons est de la forme « l'expression ε est-elle bien typée ou non ? ». Nous montrons qu'on peut le réduire à une série de problèmes plus simples, exprimés uniquement dans le langage de types et interprétés dans le modèle. Ces problèmes élémentaires ne font plus du tout référence au langage, mais sont des problèmes uniquement algébriques. Moyennant quelques hypothèses de représentabilité des éléments de base du langage, cette réduction peut avoir lieu indépendamment du modèle et de l'algèbre, c'est-à-dire de façon purement syntaxique. En d'autre termes, nous allons donner un algorithme qui parcourt l'expression à traiter et qui accumule des problèmes exprimés dans le langage de types (contraintes, implications de contraintes, tests de couverture), sans chercher à les résoudre et donc indépendamment du modèle. Ensuite, nous allons prouver que cet algorithme génère bien un ensemble de problèmes dont l'interprétation est équivalente au problème initial de typage.

Le fait d'être indépendant du modèle est particulièrement important pour traiter un monde qui peut être étendu (monde ouvert). Ce thème sera développé au prochain chapitre.

Le reste du chapitre est organisé comme suit : dans la section 6.1, nous décrivons les hypothèses

de représentabilité nécessaires ; dans la section 6.2, nous introduisons les problèmes de typage algébriques élémentaires ; en section 6.3, nous donnons l'énoncé du théorème et le reste du chapitre est consacré à la preuve.

6.1 Représentation et modèle d'une représentation

Un modèle permet de manipuler des monotypes et des polytypes algébriques en utilisant la syntaxe d'un langage de types. En général, le langage ne permet pas de les représenter tous. En particulier, un polytype algébrique est un ensemble quelconque de monotypes défini en extension, alors qu'un schéma de types décrit un ensemble de façon intensionnelle à l'aide d'une contrainte syntaxique. Un simple argument de cardinalité montre que, sauf cas trivial, de nombreux polytypes algébriques ne sont pas représentables par un schéma de types, car l'ensemble des schémas de types est toujours dénombrable alors que celui des polytypes algébriques est au moins continu dès que l'algèbre est infinie.

Pour pouvoir réduire le typage à des problèmes exprimés dans le langage de types, il est donc nécessaire de pouvoir y exprimer le type des éléments de base du langage d'expressions. De plus, pour que la réduction soit indépendante du modèle, il est nécessaire que la représentation des éléments de base en soit aussi indépendante.

Supposons donnés une structure primitive $\mathcal{C}^0$ et une structure de classes $\mathcal{C}$, c'est-à-dire tout ce qu'il faut pour écrire des programmes non-typés e : noms de classe et de champs et valeurs et opérations primitives. Fixons également un langage de types $\mathcal{L}$; nous avons maintenant de quoi écrire une expression annotée ε. Pour typer ε de manière purement syntaxique, il nous manque encore une représentation syntaxique du type des éléments de base du langage.

On appelle REPRÉSENTATION des structures $\mathcal{C}^0$ et $\mathcal{C}$ dans le langage de types $\mathcal{L}$ la donnée des éléments suivants :

- pour chaque constante a, un monotype syntaxique $\widetilde{a}$;
- pour chaque primitive f, un schéma de types $\widetilde{f}$;
- pour chaque motif primitif ϖ, un monotype syntaxique $\widetilde{\varpi}$
- pour chaque classe C, un schéma de types $\widetilde{\#C}$;
- enfin, pour chaque classe C et chaque champ ℓ de C, un schéma de types $\widetilde{C \cdot \ell}$.

Notons que la notion de représentation est purement syntaxique. Elle établit simplement un lien syntaxique entre structures de base du langage d'expression et du langage de types.

Considérons maintenant un modèle $\mathcal{M}$ du langage $\mathcal{L}$. On dit que $\mathcal{M}$ est un MODÈLE DE LA REPRÉSENTATION si l'interprétation des représentations syntaxiques est compatible avec les éléments algébriques correspondants dans l'algèbre sous-jacente au modèle. Nous allons maintenant définir plus formellement cette notion.

6.1.1 Type des constantes

Pour toute constante a, l'interprétation du type syntaxique $\widetilde{a}$ de la représentation doit être égal au type algébrique associé à la constante, ce que l'on a noté $\overline{a}$:[1]

[1] Petits rappel de notation pour décoder cette égalité : $[\![\widetilde{a}]\!]^{\mathcal{M}}$ désigne l'interprétation du symbole $\widetilde{a}$ dans le modèle $\mathcal{M}$. C'est donc un élément de l'algèbre $[\![\mathcal{M}]\!]$ support du modèle. Par ailleurs, la notation $\overline{a}^{[\![\mathcal{M}]\!]}$ désigne l'abstraction de la

(HYP-$\widetilde{a}$) $\qquad \forall \mathcal{M} \in \mathsf{Modèles},\ [\![\widetilde{a}]\!]^{\mathcal{M}} = \overline{a}^{[\![\mathcal{M}]\!]}$

Exemple. Le plus simple pour représenter le type des constantes est qu'à chaque type primitif correspond dans le langage de types une constante de type, c'est-à-dire un constructeur d'arité 0. On pourra ainsi supposer que le langage de types définit les constantes int, rat, bool, etc. ■

6.1.2 Type des primitives

Considérons une opération primitive f. Pour exprimer la condition de compatibilité concernant le type syntaxique $\widetilde{f}$, nous notons $\overline{f}$ l'ensemble des types possibles que la règle algébrique (PRIM) peut donner à la primitive f. Si celle-ci est d'arité n, on a donc :

$$t \to t^0 \in \overline{f} \quad \text{ssi} \quad \exists t_1^0, \ldots, t_N^0 \in \mathcal{A}^0, \begin{cases} \forall i, \overline{\mathtt{Tuple}_n \cdot i}(t) = t_i^0 \\ \forall a_1 \in \underline{t_1^0}, \ldots, a_n \in \underline{t_n^0},\ f(a_1, \ldots, a_n) \in \underline{t^0} \end{cases} \tag{6.1}$$

Muni de cette définition, nous pouvons formuler la condition que le polytype d'une primitive f est représentable par un schéma de types, noté $\widetilde{f}$. L'interprétation de ce schéma de types doit être équivalente au polytype $\overline{f}$ elle doit être non-vide, c'est-à-dire que la primitive doit être bien typée dans le modèle considéré :

(HYP-$\widetilde{f}$) $\qquad \forall \mathcal{M} \in \mathsf{Modèles},\ \uparrow [\![\widetilde{f}]\!]^{\mathcal{M}} = \uparrow \overline{f}^{[\![\mathcal{M}]\!]} \neq \varnothing$

Notons qu'on fait porter l'égalité sur la clôture supérieure des polytypes algébriques plutôt que sur les polytypes eux-mêmes. Comme deux polytypes équivalents par cette relation sont interchangeables par la règle (SUB$^\sharp$), la puissance de l'hypothèse est donc identique, mais cela donne un peu plus de liberté pour choisir le schéma $\widetilde{f}$.

Exemple. Le schéma de types de l'addition sera par exemple $\forall \alpha \mid \alpha \leqslant \mathsf{rat} \,.\, \langle \alpha, \alpha \rangle \to \alpha$. Celui de la comparaison peut être indifféremment $\forall \alpha \mid \alpha \leqslant \mathsf{rat} \,.\, \langle \alpha, \alpha \rangle \to \mathsf{bool}$ ou $\forall \varnothing \,.\, \langle \mathsf{rat}, \mathsf{rat} \rangle \to \mathsf{bool}$ qui sont équivalents. ■

6.1.3 Type des motifs primitifs

Pour chaque motif primitif ϖ, définissons donc l'ensemble des types des constantes acceptées par ϖ de la façon suivante :

$$\overline{\varpi}^{\mathcal{A}} \quad \triangleq \quad \left\{ \overline{a}^{\mathcal{A}} \mid a \in \varpi \right\} \tag{6.2}$$

Pour que $\mathcal{M}$ soit un modèle de la représentation, on exige alors l'égalité suivante :

constante a dans l'algèbre $[\![\mathcal{M}]\!]$.

(HYP-$\widetilde{\varpi}$)	$\forall \mathcal{M} \in \mathsf{Modèles},\ [\![\widetilde{\varpi}]\!]^{\mathcal{M}} = \overline{\varpi}^{[\![\mathcal{M}]\!]}$

Exemple. Si ϖ est le motif primitif **`rat`**, $\widetilde{\varpi}$ peut être choisi égal à $\forall \alpha \mid \alpha \leqslant \mathsf{rat}.\alpha$. L'interprétation de ce schéma correspond en effet au type de toutes les constantes acceptées par ϖ, c'est-à-dire $\{\mathsf{rat}, \mathsf{int}\}$. ■

6.1.4 Type des classes et des champs

Pour que $\mathcal{M}$ soit un modèle de la représentation, nous exigeons la propriété suivante :

(HYP-$\widetilde{\#C}$)	$\forall \mathcal{M} \in \mathsf{Modèles},\ [\![\widetilde{\#C}]\!]^{\mathcal{M}} = \overline{\#C}^{[\![\mathcal{M}]\!]} \neq \varnothing$

Notons l'emploi de l'égalité ici : le type d'une classe est en effet utilisé à la fois dans des contextes covariants (construction d'un objet) et contravariants (filtrage).

Finalement, l'interprétation de $\widetilde{C \cdot \ell}$ doit représenter tous les types possibles de l'accesseur $C \cdot \ell$ obtenus par la règle de typage algébrique (GET). On demande donc :

(HYP-$\widetilde{C \cdot \ell}$)	$\forall \mathcal{M} \in \mathsf{Modèles},\ [\![\widetilde{C \cdot \ell}]\!]^{\mathcal{M}} = \left\{ t \to^{[\![\mathcal{M}]\!]} t' \mid t' = \overline{C \cdot \ell}^{[\![\mathcal{M}]\!]}(t) \right\}$

Notons que l'hypothèse $\overline{\#C} \neq \varnothing$ implique que $[\![\widetilde{C \cdot \ell}]\!]$ est également non-vide, car $\overline{\#C}$ est supposé être le domaine de $\overline{C \cdot \ell}$.

Exemple. Quand l'algèbre est une algèbre de termes engendrés à partir des déclarations de classes du programmeur, les schémas $\widetilde{\#C}$ et $\widetilde{C \cdot \ell}$ peuvent être extraits automatiquement des déclarations. Par exemple :

```
class Cons⟨α⊕⟩ {
  head: α;
  tail: List⟨α⟩;
}
```

$$\Rightarrow \quad \begin{cases} \widetilde{\#\mathsf{Cons}} = \forall \alpha\ .\ \mathsf{Cons}\langle\alpha\rangle \\ \widetilde{\mathsf{Cons} \cdot \mathit{head}} = \forall \alpha\ .\ \mathsf{Cons}\langle\alpha\rangle \to \alpha \\ \widetilde{\mathsf{Cons} \cdot \mathit{tail}} = \forall \alpha\ .\ \mathsf{Cons}\langle\alpha\rangle \to \mathsf{List}\langle\alpha\rangle \end{cases}$$

■

6.2 Problèmes de typage élémentaires

Nous allons considérer deux types de problèmes que nous appelerons PROBLÈMES DE TYPAGE ÉLÉMENTAIRES. Il s'agit des problèmes d'implications de contraintes et des problèmes de test de couverture.

Nous supposons fixé un langage de types $\mathcal{L}$.

6.2.1 Implication de contraintes

Soit κ_1 et κ_2 deux contraintes dont toutes les variables libres sont contenues dans l'ensemble $\mathcal{V}$. On dit que κ_1 IMPLIQUE κ_2 dans un certain modèle $\mathcal{M}$ si et seulement si, pour toute valuation ρ de domaine $\mathcal{V}$, le fait que ${}_\rho[\![\kappa_1]\!]$ soit vrai implique ${}_\rho[\![\kappa_2]\!]$. On note alors $\mathcal{M} \models \forall\mathcal{V} \,.\, \kappa_1 \supset \kappa_2$ ou simplement $\forall\mathcal{V} \,.\, \kappa_1 \supset \kappa_2$ si le modèle est implicitement connu. On a donc formellement :

$$\mathcal{M} \models \forall\mathcal{V} \,.\, \kappa_1 \supset \kappa_2 \quad \text{ssi} \quad \forall\rho \in \left(\mathcal{V} \to [\![\mathcal{M}]\!]\right), {}_\rho[\![\kappa_1]\!]^{\mathcal{M}} \Rightarrow {}_\rho[\![\kappa_2]\!]^{\mathcal{M}} \tag{6.3}$$

Notons que rien ne force la contrainte $\exists\mathcal{V} \,.\, \kappa_1$ à être satisfiable. Si elle ne l'est pas, alors l'implication $\forall\mathcal{V} \,.\, \kappa_1 \supset \kappa_2$ est vraie indépendamment de κ_2.

6.2.2 Test de couverture

Soit $\pi_1, \ldots, \pi_k$ une liste de motifs et σ un schéma de type clos (sans variable libre). On dit que σ est COUVERT par la liste de motifs si et seulement si tout monotype algébrique élément de $[\![\sigma]\!]$ est couvert au sens algébrique par les motifs. On note alors $\mathcal{M} \models \sigma \bowtie \pi_1; \ldots; \pi_k$ ou simplement $\sigma \bowtie \pi_1; \ldots; \pi_k$ si le modèle est implicitement connu. Formellement, on a donc :

$$\mathcal{M} \models \sigma \bowtie \pi_1; \ldots; \pi_k \quad \text{ssi} \quad \forall t \in [\![\sigma]\!]^{\mathcal{M}}, t \bowtie^{[\![\mathcal{M}]\!]} \pi_1; \ldots; \pi_k \tag{6.4}$$

Remarque. On force donc tous les éléments de σ à être des types fonctionnels car la définition de $t \bowtie^{[\![\mathcal{M}]\!]} \ldots$ impose que t soit un type fonctionnel. ■

6.3 Énoncé du théorème

THÉORÈME 3. *Soient $\mathcal{C}^0$ une structure primitive, $\mathcal{C}$ une structure de classe, et $\mathcal{L}$ un langage de types. On fixe une représentation des structures $\mathcal{C}^0$ et $\mathcal{C}$ dans le langage $\mathcal{L}$.*

Pour toute expression annotée ε écrite dans la syntaxe définie par $\mathcal{C}^0$, $\mathcal{C}$ et $\mathcal{L}$, on peut calculer une liste de problèmes de typage élémentaires (implications de contrainte et tests de couverture) $P_1, \ldots, P_N$ et un schéma de types σ tels que, pour tout $\mathcal{M}$ modèle de la représentation considérée, on ait les propriétés suivantes :

- *si pour tout i, $\mathcal{M} \models P_i$, alors $\mathcal{M} \models \varnothing; \varnothing \vdash \varepsilon : [\![\sigma]\!]^{\mathcal{M}}$;*
- *s'il existe T tel que $\mathcal{M} \models \varnothing; \varnothing \vdash \varepsilon : T$, alors, $[\![\sigma]\!]^{\mathcal{M}} \leqslant^\sharp T$ et pour tout i, $\mathcal{M} \models P_i$.*

Le reste de ce chapitre est consacré à la démonstration de ce théorème.

Plan de la preuve

Le principe de la preuve consiste à introduire un nouveau système de types, dirigé par la syntaxe et opérant sur des types syntaxiques. Les règles de ce système utilisent les problèmes de typage pour assurer que l'interprétation des types qu'elles manipulent vérifient certaines propriétés algébriques. On vérifie ensuite que ces propriétés assurent la sûreté du typage en interprétant chaque règle dans le système semi-algébrique (lemme de correction). Inversement, on montre que les problèmes de typage engendrés par les règles syntaxiques sont juste suffisants pour assurer cette sûreté (lemme de complétude). On en conclut que le typage d'une expression annotée par le système algébrique est équivalent à la résolution des problèmes de typage engendrés par le système syntaxique. On vérifie enfin qu'on peut bien engendrer ces problèmes mécaniquement et indépendamment du modèle choisi pour les interpréter.

Voici comme est organisée la suite du chapitre : dans la section 6.4, on introduit des abréviations et définitions commodes pour exprimer le système de types syntaxique. Les règles du système sont introduites à la section 6.5. On en montre la correction à la section 6.6 puis la complétude en 6.7. La complétude ayant été obtenue pour une forme simplifiée des expressions (sans masquage de variables de type), on montre en section 6.8 comment lever cette hypothèse en renommant convenablement les variables de type. Enfin, on conclut en section 6.9.

6.4 Quelques définitions

Cette section regroupe les différentes définitions utiles pour écrire le système de types syntaxique de la section suivante.

6.4.1 Relations de sous-typage généralisées

Rappelons la définition de $\leqslant^\sharp$, relation qui opère sur les ensembles quelconques de monotypes :

$$T \leqslant^\sharp U \quad \text{ssi} \quad \forall u \in U, \exists t \in T, t \leqslant u$$

On définit de même la relation $\leqslant^\flat$ comme suit :

$$T \leqslant^\flat U \quad \text{ssi} \quad \forall t \in T, \exists u \in U, t \leqslant u$$

Ces deux relations sont transitives et réflexives et étendent $\leqslant$ si on identifie un monotype t au singleton $\{t\}$: les propositions $\{t\} \leqslant^\sharp \{u\}$, $\{t\} \leqslant^\flat \{u\}$ et $t \leqslant u$ sont équivalentes. Les relations d'équivalence associées aux préordres $\leqslant^\sharp$ et $\leqslant^\flat$ sont notées respectivement $\approx^\sharp$ et $\approx^\flat$:

$$T \approx^\sharp U \quad \text{ssi} \quad T \leqslant^\sharp U \text{ et } U \leqslant^\sharp T$$
$$T \approx^\flat U \quad \text{ssi} \quad T \leqslant^\flat U \text{ et } U \leqslant^\flat T$$

Il est commode de définir aussi la relation de compatibilité $\leqslant^\exists$:

$$T \leqslant^\exists U \quad \text{ssi} \quad \exists t \in T, \exists u \in U, t \leqslant u$$

Attention : $\leqslant^{\exists}$ n'est pas transitive. Mais, on a la propriété de transitivité avec $\leqslant^{\sharp}$ à gauche et $\leqslant^{\flat}$ à droite :

$$T' \leqslant^{\sharp} T \wedge T \leqslant^{\exists} U \wedge U \leqslant^{\flat} U' \Rightarrow T' \leqslant^{\exists} U'$$

6.4.2 Abréviations de contraintes

Soient $\sigma_1 = \forall \mathcal{V}_1 \mid \kappa_1 \,.\, \tau_1$ et $\sigma_2 = \forall \mathcal{V}_2 \mid \kappa_2 \,.\, \tau_2$ deux schémas de types ($\mathcal{V}_1$ et $\mathcal{V}_2$ ne sont pas forcément disjoints). La contrainte $\sigma_1 \leqslant^{\exists} \sigma_2$ est définie comme l'abréviation suivante où on choisit $\alpha \notin \mathit{ftv}(\sigma_1) \cup \mathcal{V}_1 \cup \mathit{ftv}(\sigma_2) \cup \mathcal{V}_2$:

$$\sigma_1 \leqslant^{\exists} \sigma_2 \quad \equiv \quad \exists \alpha \,.\, (\exists \mathcal{V}_1 \,.\, \kappa_1 \wedge \tau_1 \leqslant \alpha) \wedge (\exists \mathcal{V}_2 \,.\, \kappa_2 \wedge \alpha \leqslant \tau_2) \tag{6.5}$$

L'équivalence suivante est réalisée dans tous les modèles du langage et pour toute valuation $\hat{\rho}$ définie sur au moins les variables libres de σ_1 et σ_2 :

$$\hat{\rho}[\![\sigma_1 \leqslant^{\exists} \sigma_2]\!] \iff \hat{\rho}[\![\sigma_1]\!] \leqslant^{\exists} \hat{\rho}[\![\sigma_2]\!] \tag{6.6}$$

Démonstration. Supposons que $\hat{\rho}[\![\sigma_1 \leqslant^{\exists} \sigma_2]\!]$ soit vrai. Il existe donc ρ de domaine $\{\alpha\}$, ρ_1 de domaine $\mathcal{V}_1$ et ρ_2 de domaine $\mathcal{V}_2$ telles que $\hat{\rho}\oplus\rho\oplus\rho_1[\![\kappa_1 \wedge \tau_1 \leqslant \alpha]\!]$ et $\hat{\rho}\oplus\rho\oplus\rho_2[\![\kappa_2 \wedge \alpha \leqslant \tau_2]\!]$. Posons $t = \rho(\alpha)$ et $t_1 = \hat{\rho}\oplus\rho\oplus\rho_1[\![\tau_1]\!]$. Observons que $\hat{\rho} \oplus \rho \oplus \rho_1$ coïncide avec $\hat{\rho} \oplus \rho_1$ sur les variables libres de κ_1 et τ_1 car celles-ci sont ou bien dans $\mathcal{V}_1$, domaine de ρ_1, ou bien dans les variables libres de σ_1, c'est-à-dire différentes de α, domaine de ρ. Par ailleurs, remarquons que $\hat{\rho} \oplus \rho \oplus \rho_1(\alpha) = t$ car α n'est pas dans $\mathcal{V}_1$, domaine de ρ_1. On a donc $\hat{\rho}\oplus\rho_1[\![\kappa_1]\!]$, $\hat{\rho}\oplus\rho_1[\![\tau_1]\!] = t_1$ et $t_1 \leqslant t$. Le monotype t_1 est donc un élément de $\hat{\rho}[\![\sigma_1]\!]$. On montre de façon symétrique qu'il existe $t_2 \in \hat{\rho}[\![\sigma_2]\!]$ tel que $t \leqslant t_2$.

Inversement, si $\hat{\rho}[\![\sigma_1]\!] \leqslant^{\exists} \hat{\rho}[\![\sigma_2]\!]$, il existe $t_i \in \hat{\rho}[\![\sigma_i]\!]$ tels que $t_1 \leqslant t_2$. Il existe donc ρ_i tel que $\hat{\rho}\oplus\rho_i[\![\kappa_i]\!]$ et $t_i = \hat{\rho}\oplus\rho_i[\![\tau_i]\!]$. En posant $\rho = [\alpha \mapsto t_1]$, il est clairement possible de remonter le raisonnement ci-dessus pour conclure $\hat{\rho}\oplus\rho\oplus\rho_1[\![\kappa_1 \wedge \tau_1 \leqslant \alpha]\!]$ et $\hat{\rho}\oplus\rho\oplus\rho_2[\![\kappa_2 \wedge \alpha \leqslant \tau_2]\!]$, c'est-à-dire $\hat{\rho}[\![\sigma_1 \leqslant^{\exists} \sigma_2]\!]$. ■

Si τ_1 est un monotype et $\sigma_2 = \forall \mathcal{V}_2 \mid \kappa_2 \,.\, \tau_2$ un schéma de type. La contrainte $\tau_1 \in \sigma_2$ est définie comme l'abréviation suivante où $\alpha \notin \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\sigma_2) \cup \mathcal{V}_2$:

$$\tau_1 \in \sigma_2 \quad \equiv \quad \exists \alpha \,.\, \tau_1 \leqslant \alpha \wedge \alpha \leqslant \tau_1 \wedge (\exists \mathcal{V}_2 \,.\, \kappa_2 \wedge \tau_2 \leqslant \alpha \wedge \alpha \leqslant \tau_2) \tag{6.7}$$

Par un raisonnement similaire à celui développé ci-dessus et en tenant compte du fait que α est choisie « fraîche », on a la caractérisation suivante :

$$\hat{\rho}[\![\tau_1 \in \sigma_2]\!] \iff \hat{\rho}[\![\tau_1]\!] \in \hat{\rho}[\![\sigma_2]\!] \tag{6.8}$$

6.4.3 Contextes

On appelle CONTEXTE la donnée d'un ensemble de variables de type et d'une contrainte :

$$\Delta ::= \{\mathcal{V} \mid \kappa\}$$

L'ensemble des variables libres d'un contexte $\{\mathcal{V}|\kappa\}$ est égal à $ftv(\kappa)\setminus\mathcal{V}$. L'interprétation d'un contexte dans un modèle $\mathcal{M}$ et une valuation $\hat{\rho}$ définissant au moins les variables libres du contexte est l'ensemble des valuations ρ de domaine $\mathcal{V}$ qui satisfont la contrainte κ :

$$(6.9) \qquad {}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!]^{\mathcal{M}} \quad \triangleq \quad \left\{\rho \in \left(\mathcal{V} \to [\![\mathcal{M}]\!]\right) \mid {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!]\right\}$$

Le DOMAINE d'un contexte $\Delta = \{\mathcal{V} \mid \kappa\}$ est l'ensemble de variables $\mathcal{V}$ et est noté $\mathrm{dom}(\Delta)$.

Si Δ_1 est un contexte clos et Δ_2 un contexte dont toutes les variables libres sont dans le domaine de Δ_1, alors le PRODUIT du contexte Δ_1 par le contexte Δ_2 est défini comme l'abréviation suivante :

$$(6.10) \quad \{\mathcal{V}_1 \mid \kappa_1\} \otimes \{\mathcal{V}_2 \mid \kappa_2\} \quad \equiv \quad \{\mathcal{V}_1, \mathcal{V}_2 \mid (\exists(\mathcal{V}_1 \cap \mathcal{V}_2) \,.\, \kappa_1) \wedge \kappa_2\} \qquad \begin{pmatrix} \text{conditions :} \\ ftv(\kappa_1) \subseteq \mathcal{V}_1 \\ ftv(\kappa_2) \subseteq \mathcal{V}_1 \cup \mathcal{V}_2 \end{pmatrix}$$

Le produit de contextes est caractérisé par la propriété suivante :

$$(6.11) \qquad [\![\Delta_1 \otimes \Delta_2]\!] = \left\{\rho_1 \oplus \rho_2 \mid \rho_1 \in [\![\Delta_1]\!] \wedge \rho_2 \in {}_{\rho_1}[\![\Delta_2]\!]\right\}$$

Démonstration. Posons $\Delta_1 = \{\mathcal{V}_1 \mid \kappa_1\}$, $\Delta_2 = \{\mathcal{V}_2 \mid \kappa_2\}$ et $\mathcal{V}' = \mathcal{V}_1 \cap \mathcal{V}_2$.

• $(\subseteq)$: soit $\rho \in [\![\Delta_1 \otimes \Delta_2]\!]$. On a ${}_{\rho}[\![\kappa_2]\!]$ et ${}_{\rho}[\![\exists\mathcal{V}' \,.\, \kappa_1]\!]$. Il existe donc ρ' de domaine $\mathcal{V}'$ tel que ${}_{\rho\oplus\rho'}[\![\kappa_1]\!]$ est vrai. Posons alors $\rho_1 = (\rho \oplus \rho') \restriction \mathcal{V}_1$ et $\rho_2 = \rho \restriction \mathcal{V}_2$. On a bien $\rho = \rho_1 \oplus \rho_2$ car ρ' porte sur des variables masquées par ρ_2. Comme $\rho \oplus \rho'$ coïncide avec ρ_1 sur $\mathcal{V}_1$, ensemble qui contient toutes les variables libres de κ_1, Δ_1 étant clos, ${}_{\rho\oplus\rho_1}[\![\kappa_1]\!]$ équivaut à ${}_{\rho_1}[\![\kappa_1]\!]$. On a donc bien $\rho_1 \in [\![\Delta_1]\!]$.

• $(\supseteq)$: soit $\rho_1 \in [\![\Delta_1]\!]$ et $\rho_2 \in {}_{\rho_1}[\![\Delta_2]\!]$. Posons $\rho = \rho_1 \oplus \rho_2$. On sait déjà que ${}_{\rho}[\![\kappa_2]\!]$. On montre ${}_{\rho}[\![\exists\mathcal{V}' \,.\, \kappa_1]\!]$ en vérifiant que ρ et ρ_1 coïncident sur les variables libres de $\exists\mathcal{V}' \,.\, \kappa_1$. En effet, si $\alpha \in ftv(\exists\mathcal{V}' \,.\, \kappa_1)$, alors $\alpha \in \mathcal{V}_1$ et $\alpha \notin \mathcal{V}'$. ■

On note $\Delta \otimes \kappa$ le produit $\Delta \otimes \{\varnothing \mid \kappa\}$. Le produit n'est donc défini que si $ftv(\Delta) = \varnothing$ et $ftv(\kappa) \subseteq \mathrm{dom}(\Delta)$.

6.4.4 Problèmes sous contexte

Satisfiabilité

Soit $\hat{\Delta} = \{\hat{\mathcal{V}} \mid \hat{\kappa}\}$ un contexte clos. Soit κ une contrainte dont toutes les variables libres sont dans $\hat{\mathcal{V}}$. Le problème de la satisfiabilité de κ dans le contexte $\hat{\Delta}$ est le problème d'implication élémentaire défini par l'abréviation suivante :

$$(6.12) \qquad \hat{\Delta} \vdash \kappa \quad \equiv \quad \forall\hat{\mathcal{V}} \,.\, \hat{\kappa} \supset \kappa$$

Le problème de satisfiabilité sous contrainte est caractérisé par la propriété suivante :

$$(6.13) \qquad \mathcal{M} \models \hat{\Delta} \vdash \kappa \iff \forall\hat{\rho} \in [\![\hat{\Delta}]\!]^{\mathcal{M}}, {}_{\hat{\rho}}[\![\kappa]\!]^{\mathcal{M}}$$

Implication de contraintes

Soit $\hat{\Delta} = \{\hat{\mathcal{V}} \mid \hat{\kappa}\}$ un contexte clos. Soit $\mathcal{V}$ un ensemble de variables de type et κ_1 et κ_2 deux contraintes dont les variables libres sont toutes dans $\hat{\mathcal{V}} \cup \mathcal{V}$. Un problème d'implication de contraintes sous contexte est défini comme l'abréviation suivante :

$$\hat{\Delta} \vdash \forall \mathcal{V} \,.\, \kappa_1 \supset \kappa_2 \quad \equiv \quad \hat{\Delta} \otimes \{\mathcal{V} \mid \kappa_1\} \vdash \kappa_2 \tag{6.14}$$

L'implication de contraintes sous contexte est caractérisée par la propriété suivante :

$$\mathcal{M} \models \hat{\Delta} \vdash \forall \mathcal{V} \,.\, \kappa_1 \supset \kappa_2 \iff \forall \hat{\rho} \in [\![\hat{\Delta}]\!]^{[\![\mathcal{M}]\!]}, \forall \rho \in \left(\mathcal{V} \to [\![\mathcal{M}]\!]\right), {}_{\hat{\rho} \oplus \rho}[\![\kappa_1]\!]^{\mathcal{M}} \Rightarrow {}_{\hat{\rho} \oplus \rho}[\![\kappa_2]\!]^{\mathcal{M}} \tag{6.15}$$

Sous-typage

Soit $\hat{\Delta} = \{\hat{\mathcal{V}} \mid \hat{\kappa}\}$ un contexte clos. Soient σ_1 et σ_2 deux schémas de types dont toutes les variables libres sont dans $\hat{\mathcal{V}}$. Le problème de sous-typage de schémas est défini par l'abréviation suivante :

$$\hat{\Delta} \vdash \sigma_1 \leqslant^{\sharp} \sigma_2 \quad \equiv \quad \hat{\Delta} \vdash \forall \alpha \,.\, \sigma_2 \leqslant^{\exists} \alpha \supset \sigma_1 \leqslant^{\exists} \alpha \qquad \text{(on choisit } \alpha \notin \hat{\mathcal{V}}) \tag{6.16}$$

Le sous-typage sous contexte est caractérisé par la propriété suivante :

$$\mathcal{M} \models \hat{\Delta} \vdash \sigma_1 \leqslant^{\sharp} \sigma_2 \iff \forall \hat{\rho} \in [\![\hat{\Delta}]\!]^{\mathcal{M}}, {}_{\hat{\rho}}[\![\sigma_1]\!]^{\mathcal{M}} \leqslant^{\sharp} {}_{\hat{\rho}}[\![\sigma_2]\!]^{\mathcal{M}} \tag{6.17}$$

6.5 Système de types syntaxique

On définit le jugement $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ et les jugements auxiliaires $\hat{\Delta}; \Gamma \vdash \pi \Rightarrow \varepsilon : \tau \to \tau'$ et $\vdash \pi : \tau; \hat{\Delta}; \Gamma$. Le jugement $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ n'est valable que si $\mathit{ftv}(\varepsilon) \subseteq \mathrm{dom}(\hat{\Delta})$.

$$(\textsc{Intro}_V) \qquad \frac{\mathrm{dom}(\hat{\Delta}) \# \mathcal{V} \# \alpha \qquad \hat{\Delta} \vdash \exists \mathcal{V} \,.\, \kappa \qquad \hat{\Delta} \otimes \{\mathcal{V} \mid \kappa\}; \Gamma \vdash \varepsilon : \sigma}{\hat{\Delta}; \Gamma \vdash (\forall \mathcal{V} \mid \kappa \,.\, \varepsilon) : \forall \mathcal{V}, \alpha \mid \kappa \wedge \sigma \leqslant^{\exists} \alpha \,.\, \alpha}$$

$$(\textsc{Var}_V) \qquad \frac{}{\hat{\Delta}; \Gamma \vdash x : \Gamma(x)}$$

$$(\textsc{Cst}_V) \qquad \frac{}{\hat{\Delta}; \Gamma \vdash a : \widetilde{a}}$$

$$(\textsc{Prim}_V) \qquad \frac{}{\hat{\Delta}; \Gamma \vdash f : \widetilde{f}}$$

$$(\text{APP}_V)\quad \frac{\hat{\Delta};\Gamma\vdash\varepsilon_1\colon\sigma_1 \qquad \hat{\Delta};\Gamma\vdash\varepsilon_2\colon\sigma_2 \qquad \sigma=\forall\alpha\beta\mid\sigma_1\leqslant^{\exists}\alpha\to\beta\wedge\sigma_2\leqslant^{\exists}\alpha\,.\,\beta \qquad \alpha\#\beta\#\mathrm{dom}(\hat{\Delta}) \qquad \hat{\Delta}\vdash\sigma}{\hat{\Delta};\Gamma\vdash\varepsilon_1\,\varepsilon_2\colon\sigma}$$

$$(\text{FUN}_V)\quad \frac{\hat{\Delta};\Gamma[x:\tau]\vdash\varepsilon\colon\sigma \qquad \alpha\#\mathrm{dom}(\hat{\Delta})}{\hat{\Delta};\Gamma\vdash\mathbf{fun}\,x:\tau\;\Rightarrow\varepsilon\colon\forall\alpha\mid\sigma\leqslant^{\exists}\alpha\,.\,\tau\to\alpha}$$

$$(\text{ACCÈS}_V)\quad \frac{}{\hat{\Delta};\Gamma\vdash C\cdot\ell\colon\widetilde{C\cdot\ell}}$$

$$(\text{OBJET}_V)\quad \frac{\hat{\Delta};\Gamma\vdash\varepsilon_1\colon\sigma_1 \qquad\ldots\qquad \hat{\Delta};\Gamma\vdash\varepsilon_n\colon\sigma_n \qquad \sigma=\forall\alpha\mid\alpha\in\widetilde{\#C}\wedge\textstyle\bigwedge_{i=1}^{n}\exists\beta\,.\,(\sigma_i\leqslant^{\exists}\beta\wedge\alpha\to\beta\leqslant^{\exists}\widetilde{C\cdot\ell})\,.\,\alpha \qquad \alpha\#\beta\#\mathrm{dom}(\hat{\Delta}) \qquad \hat{\Delta}\vdash\sigma}{\hat{\Delta};\Gamma\vdash C\,\{\ell_1=\varepsilon_1;\ldots;\ell_n=\varepsilon_n\}\colon\sigma}$$

$$(\text{LET}_V)\quad \frac{\hat{\Delta};\Gamma\vdash\varepsilon_1\colon\sigma_1 \qquad \hat{\Delta};\Gamma[x:\sigma_1]\vdash\varepsilon_2\colon\sigma_2}{\hat{\Delta};\Gamma\vdash\mathbf{let}\;x=\varepsilon_1\;\mathbf{in}\;\varepsilon_2\colon\sigma_2}$$

$$(\text{FIX}_V)\quad \frac{\hat{\Delta}\vdash\sigma \qquad \hat{\Delta};\Gamma[x:\sigma]\vdash\varepsilon\colon\sigma' \qquad \hat{\Delta}\vdash\sigma'\leqslant^{\sharp}\sigma}{\hat{\Delta};\Gamma\vdash(\mathbf{fix}\,x:\sigma\;\Rightarrow\varepsilon)\colon\sigma}$$

$$(\text{METH}_V)\quad \frac{\hat{\Delta}\vdash\tau\to\tau'\bowtie\pi_1;\ldots;\pi_k \qquad \text{pour tout } j,\ \hat{\Delta};\Gamma\vdash\pi_j\Rightarrow\varepsilon_j\colon\tau\to\tau'}{\hat{\Delta};\Gamma\vdash\left(\mathbf{meth}\colon\tau\to\tau'\left\{\pi_j\Rightarrow\varepsilon_j;\ldots;\pi_j\Rightarrow\varepsilon_j\right\}\right)\colon\tau\to\tau'}$$

$$(\text{CAS}_V)\quad \frac{\mathrm{dom}(\hat{\Delta})\#\mathrm{dom}(\Delta_j) \qquad \vdash\pi_j:\tau_j;\Delta_j;\Gamma_j \qquad \hat{\Delta}\otimes\Delta_j\otimes\tau_j\leqslant\tau;\Gamma\oplus\Gamma_j\vdash\varepsilon_j\colon\sigma_j \qquad \hat{\Delta}\otimes\Delta_j\otimes\tau_j\leqslant\tau\vdash\sigma_j\leqslant^{\sharp}\tau'}{\hat{\Delta};\Gamma\vdash\pi_j\Rightarrow\varepsilon_j\colon\tau\to\tau'}$$

$$(\pi\text{-UNIVERSEL}_V)\quad \frac{}{\vdash\_\colon\alpha;\{\alpha\mid\mathsf{vrai}\};[]}$$

$$(\pi\text{-PRIM}_V)\quad \frac{}{\vdash\varpi:\alpha;\{\alpha\mid\alpha\in\widetilde{\varpi}\};[]}$$

$$(\pi\text{-LIEUR}_V)\qquad \frac{\vdash \pi : \tau;\ \Delta;\ \Gamma}{\vdash \pi\ \mathbf{as}\ x : \tau;\ \Delta;\ \Gamma[x : \tau]}$$

$$(\pi\text{-CLASSE}_V)\qquad \frac{\begin{array}{c}\mathrm{dom}(\Delta_1) \# \ldots \mathrm{dom}(\Delta_n) \# \alpha \\ \vdash \pi_1 : \tau_1;\ \Delta_1;\ \Gamma_1 \qquad \ldots \qquad \vdash \pi_n : \tau_n;\ \Delta_n;\ \Gamma_n\end{array}}{\vdash \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : \alpha;\ \Delta_1 \otimes \cdots \wedge \Delta_n \otimes \{\alpha \mid \alpha \in \widetilde{\#C} \wedge \bigwedge_{i=1}^{n} \alpha \to \tau_i \leqslant^{\exists} \widetilde{C \cdot \ell_i}\};\ \Gamma_1 \oplus \cdots \oplus \Gamma_n}$$

$$(\pi\text{-SOUS-CLASSE}_V)\qquad \frac{\vdash \#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : \tau;\ \Delta;\ \Gamma \qquad \alpha\# \in \mathrm{dom}(\Delta)}{\vdash C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\} : \alpha;\ \Delta \otimes \{\alpha \mid \alpha \leqslant \tau\};\ \Gamma}$$

6.6 Correction

Pour tous environnements de typage algébriques G et G', on écrit $G \leqslant^\sharp G'$ si $\mathrm{dom}(G) = \mathrm{dom}(G')$ et $G(x) \leqslant^\sharp G'(x)$ pour tout $x \in \mathrm{dom}(G)$.

Pour tout environnement de typage syntaxique Γ, on écrit $\hat{\Delta} \vdash \Gamma$ quand $\hat{\Delta} \vdash \Gamma(x)$ pour tout $x \in \mathrm{dom}(\Gamma)$. Pour tout $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on note alors ${}_{\hat{\rho}}[\![\Gamma]\!]$ l'environnement de typage algébrique de même domaine que Γ qui envoie la variable x sur le polytype algébrique ${}_{\hat{\rho}}[\![\Gamma(x)]\!]$. Ce dernier ensemble est bien non-vide quand $\hat{\Delta} \vdash \Gamma$ est vrai.

LEMME 6.1. *Si $\hat{\rho}$; $G \vdash \varepsilon : T$ et si $G' \leqslant^\sharp G$, alors $\hat{\rho}$; $G' \vdash \varepsilon : T$*

Démonstration. Il suffit d'insérer une instance de la règle (SUB$_A$) après chaque occurrence de (VAR$_A$) dans la dérivation de $\hat{\rho}$; $G \vdash \varepsilon : T$. ■

LEMME 6.2. *Si $\vdash \pi : \tau;\ \Delta;\ \Gamma$ **(1)** et si $t \vdash \pi : t';\ G$ **(2)**, alors il existe $\rho \in [\![\Delta]\!]$ tel que $G \leqslant^\sharp {}_\rho[\![\Gamma]\!]$ **(3)** et $t' \leqslant {}_\rho[\![\tau]\!] \leqslant t$ **(4)**.*

Démonstration. Par induction sur la dérivation de (2). On considère la dernière règle R appliquée.

• Si $R = (\pi\text{-UNIVERSEL})$, π est le motif universel _. Δ est alors un contexte de la forme $\{\alpha \mid \mathsf{vrai}\}$, Γ est l'environnement vide et τ vaut α. Du côté algébrique, G est l'environnement vide, t est un monotype quelconque et $t' = t$. Il suffit donc de prendre $\rho = [\alpha \mapsto t]$. Cette valuation est bien dans l'interprétation de Δ et on a $t = {}_\rho[\![\tau]\!]$.

• Si $R = (\pi\text{-PRIM})$, π est un motif primitif ϖ, alors le contexte Δ est de la forme $\{\alpha \mid \alpha \in \widetilde{\varpi}\}$, l'environnement Γ est vide et τ est égal à α. Par ailleurs G est vide et t est un monotype de la forme $\overline{a}$ pour une certaine constante $a \in \varpi$ et $t' = t$. On sait donc que $t \in \overline{\varpi}$ et par l'hypothèse (HYP-$\widetilde{\varpi}$) que $t \in [\![\widetilde{\varpi}]\!]$. En prenant $\rho = [\alpha \mapsto t]$, on voit que ${}_\rho[\![\alpha \in \widetilde{\varpi}]\!]$ donc que $\rho \in [\![\Delta]\!]$.

• Si $R = (\pi\text{-SUB})$, on a alors $t'' \vdash \pi : t';\ G$ (**5**) pour un certain $t'' \leqslant t$. On peut appliquer l'hypothèse d'induction à (5) pour déduire qu'il existe $\rho \in [\![\Delta]\!]$ telle que $G \leqslant^\sharp {}_\rho[\![\Gamma]\!]$ et $t' \leqslant {}_\rho[\![\tau]\!] \leqslant t''$. On conclut par transitivité de $\leqslant$.

• Si $R = (\pi\text{-LIEUR})$, π est alors de la forme π' **as** x. On a $G = G'[x : t']$ et $t \vdash \pi' : t';\ G'$ (**6**). Du côté syntaxique, on a $\vdash \pi' : \tau;\ \Delta';\ \Gamma'$ et $\Gamma = \Gamma'[x : \tau]$. On peut appliquer l'hypothèse d'induction

à (6) pour déduire qu'il existe $\rho \in [\![\Delta]\!]$ telle que $G' \leqslant^\sharp {}_\rho[\![\Gamma']\!]$ et $t' \leqslant {}_\rho[\![\tau]\!] \leqslant t''$. On conclut par la transitivité de $\leqslant$ que $t' \leqslant {}_\rho[\![\tau]\!] \leqslant t$. D'autre part, on a $t' \leqslant {}_\rho[\![\tau]\!]$ donc $G \leqslant^\sharp {}_\rho[\![\Gamma]\!]$.

• Si $R = (\pi\text{-CLASSE})$, π est alors de la forme $\#C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$ et on a $t = t' \in \overline{\#C}$, $\overline{C\cdot\ell_i}(t) \vdash \pi_i : t_i'; G_i$ (**7**) et $G = G_1 \oplus \cdots \oplus G_n$. Du côté syntaxique, on a pour chaque i, $\vdash \pi_i : \tau_i; \Delta_i; \Gamma_i$ (**8**) avec $\mathrm{dom}(\Delta_1) \# \ldots \mathrm{dom}(\Delta_n) \# \alpha$ et $\Delta = \Delta_1 \otimes \cdots \otimes \Delta_n \otimes \{\alpha \mid \kappa\}$ avec $\kappa = \alpha \in \widetilde{\#C} \wedge \bigwedge_{i=1}^n \alpha \to \tau_i \leqslant^\exists \widetilde{C\cdot\ell_i}$. Appliquons l'hypothèse d'induction sur les jugements (7) et (8) pour chaque i entre 1 et n. Il existe donc $\rho_i \in [\![\Delta_i]\!]$ telle que $G_i \leqslant^\sharp {}_{\rho_i}[\![\Gamma_i]\!]$ (**9**) et $t_i' \leqslant {}_{\rho_i}[\![\tau_i]\!] \leqslant \overline{C\cdot\ell_i}(t)$ (**10**). Posons $\rho = \rho_1 \oplus \cdots \oplus \rho_n \oplus [\alpha \mapsto t]$. Comme les contextes Δ_i sont supposés porter sur des ensembles de variables de type disjoints, et que α est encore une variable différente, ρ coïncide avec chacune des ρ_i et avec $[\alpha \mapsto t]$ sur leur domaine respectif. Les équations (9) et (10) restent donc valables quand on remplace ρ_i par ρ. On a donc : $G_i \leqslant^\sharp {}_\rho[\![\Gamma]\!]$ et donc $G \leqslant^\sharp {}_\rho[\![\Gamma]\!]$. Montrons maintenant que ρ est dans l'interprétation de Δ. On a bien $t \in \overline{\#C}$, donc, par l'hypothèse (HYP-$\widetilde{\#C}$) ${}_\rho[\![\alpha \in \widetilde{\#C}]\!]$ est vrai. Pour chaque i, on sait, par la variance de la flèche appliquée à (10) que ${}_\rho[\![\tau_i]\!] \leqslant \overline{C\cdot\ell_i}(t)$ et donc $t \to {}_{\rho_i}[\![\tau_i]\!] \leqslant t \to \overline{C\cdot\ell_i}(t)$. Par l'hypothèse (HYP-$\widetilde{C\cdot\ell}$), le membre de droit de cette dernière inégalité est un élément de $[\![\widetilde{C\cdot\ell}]\!]$ et on a donc ${}_\rho[\![\alpha \to \tau_i \leqslant^\exists \widetilde{C\cdot\ell}]\!]$. On a donc montré que $\rho \in [\![\Delta]\!]$. On conclut en notant que $t = t' = {}_\rho[\![\tau]\!]$.

• Si $R = (\pi\text{-SOUS-CLASSE})$, alors π est de la forme $C\,\{\ell_1 = \pi_1; \ldots; \ell_n = \pi_n\}$, $t' = t$ et il existe t_1 tel que $t \leqslant t_1$ et $t_1 \vdash \#\pi : t_1; G$. Du côté syntaxique, on a : $\vdash \#\pi : \tau'; \Delta'; \Gamma, \tau = \alpha, \Delta = \Delta' \otimes \{\alpha \mid \alpha \leqslant \tau'\}$ et $\alpha \# \mathrm{dom}(\Delta')$. En appliquant l'hypothèse d'induction, on trouve qu'il existe $\rho' \in [\![\Delta']\!]$ telle que $G \leqslant^\sharp {}_{\rho'}[\![\Gamma]\!]$ et $t_1 \leqslant {}_\rho[\![\tau']\!] \leqslant \tau_1$, c'est-à-dire $t_1 = {}_\rho[\![\tau']\!]$. Posons alors $\rho = \rho' \oplus [\alpha \mapsto t]$. Comme $\alpha \# \mathrm{dom}(\Delta')$, ρ et ρ' coïncident sur $\mathrm{dom}(\rho')$. Donc $G \leqslant^\sharp {}_\rho[\![\Gamma]\!]$ et ${}_\rho[\![\tau']\!] = {}_{\rho'}[\![\tau']\!]$. On a aussi $t = {}_\rho[\![\alpha]\!] \leqslant t_1 \leqslant {}_\rho[\![\tau']\!]$. Donc $\rho \in [\![\Delta]\!]$ et on conclut en observant que $t = t' = {}_\rho[\![\tau]\!]$. ■

LEMME 6.3 (CORRECTION). *Si* $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ *et si* $\hat{\Delta} \vdash \Gamma$ *alors, pour tout* $\hat{\rho} \in [\![\hat{\Delta}]\!]$, *on a* $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : {}_{\hat{\rho}}[\![\sigma]\!]$.

Démonstration. Par induction sur la structure de l'expression ε. Considérons une dérivation de $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ (**1**). Celle-ci se termine forcément par la règle unique qui correspond à ε car le système de typage syntaxique est dirigé par la syntaxe.

• Si ε est une variable, le résultat est trivial.

• Si ε est une constante a, il découle de l'hypothèse (HYP-$\widetilde{a}$) : $[\![\widetilde{a}]\!] = \overline{a}$.

• Si ε est une primitive f, soit n l'arité de f. Fixons une valuation quelconque $\hat{\rho} \in [\![\hat{\Delta}]\!]$. Considérons un élément quelconque $t' \in [\![\widetilde{f}]\!]$. Par l'hypothèse (HYP-$\widetilde{f}$), il existe $t, t^0, t_1^0, \ldots, t_n^0$ tels que $t \to t^0 \leqslant t'$ (**2**), pour tout i, $\overline{\mathbf{Tuple}_n\cdot i}(t) = t_i^0$ (**3**) et pour tous $a_1 \in t_1^0, \ldots, a_n \in t_n^0$, $f(a_1, \ldots, a_n) \in \underline{t^0}$ (**4**). Les équations (4) et (3) constituent les prémisses de la règle $\overline{(\text{PRIMITIVE}_A)}$, dont on déduit le jugement $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash f : t \to t^0$ (**5**). On peut appliquer la règle (SUB$_A$) à (5) et (2) et déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash f : t'$. On a montré ce dernier jugement pour tout $t' \in [\![\widetilde{f}]\!]$. Comme l'hypothèse (HYP-$\widetilde{f}$) nous impose que $[\![\widetilde{f}]\!]$ soit non-vide, on peut appliquer la règle (GEN$_A$) pour conclure.

• Si ε est un accesseur $C\cdot\ell$, alors le jugement (1) est nécessairement de la forme $\hat{\Delta}; \Gamma \vdash \varepsilon : \widetilde{C\cdot\ell}$. Le schéma $\sigma = \widetilde{C\cdot\ell}$ est supposé clos et par l'hypothèse (HYP-$\widetilde{\#C}$), $[\![\sigma]\!] \neq \varnothing$. Par l'hypothèse (HYP-$\widetilde{C\cdot\ell}$), $[\![\sigma]\!] = \{t \to \overline{C\cdot\ell}(t)\}$. Fixons une valuation quelconque $\hat{\rho} \in [\![\hat{\Delta}]\!]$ et considérons un élément $t \to \overline{C\cdot\ell}(t) \in [\![\sigma]\!]$. Par la règle (GET$_A$), on a $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : t \to \overline{C\cdot\ell}(t)$. On a montré ce dernier jugement pour tout élément de l'ensemble non-vide $[\![\sigma]\!]$. On peut donc appliquer la règle (GEN$_A$) pour conclure $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : [\![\sigma]\!]$. On conclut en remarquant que σ étant clos, $[\![\sigma]\!] = {}_{\hat{\rho}}[\![\sigma]\!]$.

• Si ε est un objet de la forme $C\,\{\ell_i = \varepsilon_i\}$, alors la règle appliquée pour déduire le jugement (1) est forcément (OBJET_V). Les prémisses de cette règle nous assurent que pour tout i, $\hat{\Delta}; \Gamma \vdash \varepsilon_i \colon \sigma_i$ (**6**), qu'il existe deux variables $\alpha \# \beta \# \text{dom}(\hat{\Delta})$ telles que σ puisse s'ecrire :

$$\sigma = \forall\alpha \mid \alpha \in \widetilde{\#C} \wedge \bigwedge_{i=1}^{n} \exists\beta \,.\, (\sigma_i \leqslant^{\exists} \beta \wedge \alpha \to \beta \leqslant^{\exists} \widetilde{C\cdot\ell_i}) \,.\, \alpha \tag{7}$$

et que $\hat{\Delta} \vdash \sigma$ (**8**). Fixons $\hat{\rho} \in [\![\hat{\Delta}]\!]$. Par définition de (8), on sait que ${}_{\hat{\rho}}[\![\sigma]\!] \neq \varnothing$. Par ailleurs, on peut appliquer l'hypothèse d'induction à chacune des sous-expressions ε_i sachant (6). On déduit que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_i : {}_{\hat{\rho}}[\![\sigma_i]\!]$ (**9**). Considérons maintenant un élément quelconque $t \in {}_{\hat{\rho}}[\![\sigma]\!]$. D'après la définition de σ et le choix des variables α et β, on sait que $t \in [\![\widetilde{\#C}]\!]$ (**10**) et que pour tout i, il existe t_i tel que ${}_{\hat{\rho}}[\![\sigma_i]\!] \leqslant^{\exists} t_i$ (**11**) et $t \to t_i \leqslant^{\exists} [\![\widetilde{C\cdot\ell_i}]\!]$ (**12**). Appliquons la règle ($\text{SUB}^\sharp$) à (9) et (11). Il vient $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_i : t_i$ (**13**). Par (12), on sait par ailleurs qu'il existe un élément de $[\![\widetilde{C\cdot\ell_i}]\!]$ surtype de $t \to t_i$. Par l'hypothèse ($\text{HYP-}\widetilde{C\cdot\ell}$), on sait qu'il est nécessairement de la forme $u_i \to \overline{C\cdot\ell_i}(u_i)$, de sorte qu'on a $t \to t_i \leqslant u_i \to \overline{C\cdot\ell_i}(u_i)$ (**14**). L'axiome (FLÈCHE-VARIANCE) appliqué à (14) nous permet de déduire que $t_i \leqslant \overline{C\cdot\ell_i}(u_i)$ (**15**) et $u_i \leqslant t$ (**16**). Par l'hypothèse ($\text{HYP-}\widetilde{\#C}$) et en considérant (10), on sait que $t \in \overline{\#C}$ (**17**). t est donc dans le domaine de $\overline{C\cdot\ell_i}$. Sachant (16) et par l'axiome (CHAMPS-COVARIANTS), on déduit que $\overline{C\cdot\ell_i}(u_i) \leqslant \overline{C\cdot\ell_i}(t)$ (**18**). Par deux applications de la règle (SUB_A) à (13) et (15), puis à (18), on déduit que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_i : \overline{C\cdot\ell_i}(t)$ (**19**). On peut maintenant appliquer la règle (OBJET_A) à (17) et (19). Il vient donc $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : t$. On a montré ce dernier jugement pour tout $t \in {}_{\hat{\rho}}[\![\sigma]\!]$, ensemble non-vide. On peut donc appliquer la règle (GEN_A) pour conclure.

• Si ε est une application de la forme $\varepsilon_1\,\varepsilon_2$, la règle appliquée pour typer ε est (APP_V). Voici ses prémisses : pour chaque $i = 1, 2$, on a $\hat{\Delta}; \Gamma \vdash \varepsilon_i \colon \sigma_i$ (**20**), le schéma σ vaut $\forall\alpha\beta \mid \sigma_1 \leqslant^{\exists} \alpha \to \beta \wedge \sigma_2 \leqslant^{\exists} \alpha \,.\, \beta$ (**21**) avec $\alpha \# \beta \# \text{dom}(\hat{\Delta})$ et $\hat{\Delta} \vdash \sigma$ (**22**). Fixons-nous une valuation quelconque $\hat{\rho} \in [\![\hat{\Delta}]\!]$. En appliquant l'hypothèse d'induction à ε_i et sachant (20), il vient $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_i : {}_{\hat{\rho}}[\![\sigma_i]\!]$ (**23**). Par définition de (22), on sait par ailleurs que ${}_{\hat{\rho}}[\![\sigma]\!] \neq \varnothing$. Considérons alors un élément quelconque $t' \in {}_{\hat{\rho}}[\![\sigma]\!]$. La définition de σ et le choix des variables α et β nous assurent qu'il existe t, interprétation de α, tel que ${}_{\hat{\rho}}[\![\sigma_1]\!] \leqslant^{\exists} t \to t'$ (**24**) et ${}_{\hat{\rho}}[\![\sigma_2]\!] \leqslant^{\exists} t$ (**25**). En appliquant la règle ($\text{SUB}^\sharp$) à (23) pour $i = 1$ et (24), il vient $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_1 : t \to t'$ (**26**). De même sur (23) pour $i = 2$ et (25), on a $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_2 : t$ (**27**). On peut donc appliquer (APP) à (26) et (27) et déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_1\,\varepsilon_2 : t'$. On a montré ce dernier jugement pour tout monotype $t \in {}_{\hat{\rho}}[\![\sigma]\!]$, ensemble non-vide. On peut donc appliquer la règle (GEN_A) pour conclure.

• Si ε est une fonction de la forme $\texttt{fun}\ x \colon \tau\ \Rightarrow \varepsilon'$, la règle appliquée pour déduire (1) est forcément (FUN_V) et on a $\hat{\Delta}; \Gamma[x \colon \tau] \vdash \varepsilon' \colon \sigma'$ (**28**) et $\sigma = \forall\alpha \mid \sigma' \leqslant^{\exists} \alpha \,.\, \tau \to \alpha$ avec $\alpha \# \text{dom}(\hat{\Delta})$. Considérons une valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$. Comme $\Gamma[x \colon \tau]$ est toujours satisfiable dans $\hat{\Delta}$, on peut appliquer l'hypothèse d'induction sur ε' et (28). On sait donc que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x \colon {}_{\hat{\rho}}[\![\tau]\!]] \vdash \varepsilon' : {}_{\hat{\rho}}[\![\sigma']\!]$ (**29**). Considérons maintenant un élément quelconque $t \in {}_{\hat{\rho}}[\![\sigma]\!]$. Comme α est convenablement choisie pour éviter les captures, il existe donc t' (la valeur de α) tel que ${}_{\hat{\rho}}[\![\sigma']\!] \leqslant^{\exists} t'$ (**30**) et $t = {}_{\hat{\rho}}[\![\tau \to \alpha]\!]$, c'est-à-dire $t = {}_{\hat{\rho}}[\![\tau]\!] \to t'$ (**31**). En appliquant la règle ($\text{SUB}^\sharp$) à (29) et (30), il vient $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x \colon {}_{\hat{\rho}}[\![\tau]\!]] \vdash \varepsilon' : t'$. On peut alors appliquer la règle (FUN_A) à ce dernier jugement et déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : {}_{\hat{\rho}}[\![\tau]\!] \to t'$, c'est-à-dire, en utilisant l'égalité (31), $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : t$ (**32**). Par définition du système semi-algébrique, le jugement (29) ne peut être valable que si ${}_{\hat{\rho}}[\![\sigma']\!] \neq \varnothing$. La définition de σ implique clairement que ${}_{\hat{\rho}}[\![\sigma]\!] \neq \varnothing$. Comme on a montré le jugement (32) pour tout élément t de ${}_{\hat{\rho}}[\![\sigma]\!]$, ensemble non-vide, on peut appliquer la règle (GEN_A) pour conclure.

• Si ε est de la forme $\texttt{let}\ x = \varepsilon_1\ \texttt{in}\ \varepsilon_2$, la règle appliquée pour déduire le jugement (1) est

forcément (LET_V) et on a $\hat{\Delta}; \Gamma \vdash \varepsilon_1 : \sigma_1$ (**33**) et $\hat{\Delta}; \Gamma[x : \sigma_1] \vdash \varepsilon_2 : \sigma$ (**34**). On peut appliquer l'hypothèse d'induction à ε_1 sachant (33). On en déduit que pour toute valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on a ${}_{\hat{\rho}}[\![\Gamma]\!]; \varepsilon_1 \vdash {}_{\hat{\rho}}[\![T\, t_1]\!] :$ (**35**). Ce jugement implique nécessairement que ${}_{\hat{\rho}}[\![\sigma_1]\!]$ est non-vide, ce qu'on a montré pour tout $\hat{\rho} \in [\![\hat{\Delta}]\!]$. Par définition, on a donc $\hat{\Delta} \vdash \sigma_1$ et donc $\hat{\Delta} \vdash \Gamma[x : \sigma_1]$ (**36**). Fixons maintenant une valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$. On peut appliquer l'hypothèse d'induction à ε_2 sachant (34) et (36). On en déduit que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma[x : \sigma_1]]\!] \vdash \varepsilon_2 : {}_{\hat{\rho}}[\![\sigma]\!]$, c'est-à-dire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x : {}_{\hat{\rho}}[\![\sigma_1]\!]] \vdash \varepsilon_2 : {}_{\hat{\rho}}[\![\sigma]\!]$. On conclut, en appliquant la règle (LET_A), que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : {}_{\hat{\rho}}[\![\sigma]\!]$.

• Si ε est de la forme $\mathbf{fix}\ x : \sigma \Rightarrow \varepsilon'$, la règle appliquée pour déduire le jugement (1) est forcément (FIX_V) et on a : $\hat{\Delta} \vdash \sigma$ (**37**), $\hat{\Delta}; \Gamma[x : \sigma] \vdash \varepsilon' : \sigma'$ (**38**) et $\hat{\Delta} \vdash \sigma' \leqslant^\sharp \sigma$ (**39**). Par (37) et comme on a déjà $\hat{\Delta} \vdash \Gamma$, on peut écrire $\hat{\Delta} \vdash \Gamma[x : \sigma]$. On peut donc appliquer l'hypothèse d'induction sur ε'. Pour toute valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on déduit que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma[x : \sigma]]\!] \vdash \varepsilon' : {}_{\hat{\rho}}[\![\sigma']\!]$, c'est-à-dire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x : {}_{\hat{\rho}}[\![\sigma]\!]] \vdash \varepsilon' : {}_{\hat{\rho}}[\![\sigma']\!]$ (**40**). Par définition de (39), on sait que ${}_{\hat{\rho}}[\![\sigma']\!] \leqslant^\sharp {}_{\hat{\rho}}[\![\sigma]\!]$ (**41**). On peut alors appliquer la règle ($\text{SUB}^\sharp$) à (40) et (41) pour déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x : {}_{\hat{\rho}}[\![\sigma]\!]] \vdash \varepsilon' : {}_{\hat{\rho}}[\![\sigma]\!]$. On conclut en appliquant (FIX_A).

• Si ε est de la forme $\forall \mathcal{V} \mid \kappa \,.\, \varepsilon'$, alors la règle appliquée pour déduire (1) est nécessairement (INTRO_V). Les prémisses de cette règle sont $\hat{\Delta} \vdash \exists \mathcal{V} \,.\, \kappa$ (**42**), $\hat{\Delta} \otimes \{\mathcal{V} \mid \kappa\}; \Gamma \vdash \varepsilon' : \sigma'$ (**43**), $\hat{\mathcal{V}} \# \mathcal{V}$ et $\sigma = \forall \mathcal{V}, \alpha \mid \kappa \wedge \sigma' \leqslant^\exists \alpha \,.\, \alpha$ (**44**) avec $\alpha \# \text{dom}(\hat{\Delta}) \cup \mathcal{V}$. Fixons nous une valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$ et considérons un élément quelconque $t_{\hat{\rho}}[\![\sigma]\!]$. Par la propriété (44) et compte-tenu du choix de la variable α, on sait qu'il existe $\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!]$ telle que ${}_{\hat{\rho}\oplus\rho}[\![\sigma']\!] \leqslant^\exists t$ (**45**). Par la caractérisation du produit des contextes $\hat{\Delta}$ et $\{\mathcal{V} \mid \kappa\}$, on a d'autre part $\hat{\rho} \oplus \rho \in [\![\hat{\Delta} \otimes \{\mathcal{V} \mid \kappa\}]\!]$. Par ailleurs, ${}_{\hat{\rho}\oplus\rho}[\![\Gamma]\!] = {}_{\hat{\rho}}[\![\Gamma]\!]$ car $\hat{\mathcal{V}} \# \mathcal{V}$. On a donc toujours $\hat{\Delta} \otimes \{\mathcal{V} \mid \kappa\} \vdash \Gamma$. On peut donc appliquer l'hypothèse d'induction à ε' sachant (43). Il vient $\hat{\rho} \oplus \rho; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon' : {}_{\hat{\rho}\oplus\rho}[\![\sigma']\!]$ (**46**). En appliquant la règle ($\text{SUB}^\sharp$) à (46) et (45), on sait que $\hat{\rho} \oplus \rho; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon' : t$ (**47**). Par définition de (42), on sait d'autre part que ${}_{\hat{\rho}}[\![\exists \mathcal{V} \,.\, \kappa]\!]$ est vrai. On peut alors appliquer la règle (INTRO_A) à (46) et tous les jugements (47) pour déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : t$. On a montré ce dernier jugement pour tout $t \in {}_{\hat{\rho}}[\![\sigma]\!]$. Enfin, la condition (42) assurant que ${}_{\hat{\rho}}[\![\sigma]\!]$ est non-vide, on peut appliquer la règle (GEN_A) pour conclure.

• Si ε est une méthode $\mathbf{meth}\colon \tau \to \tau' \{\pi_j \Rightarrow \varepsilon_j\}$, c'est la règle ($\text{METH}_V$) qui a été appliquée pour déduire le jugement (1). Ses prémisses sont $\hat{\Delta} \vdash \tau \to \tau' \bowtie \pi_1; \ldots; \pi_k$ (**48**) et, pour tout $j \in [1, k]$, $\hat{\Delta}; \Gamma \vdash \pi_j \Rightarrow \varepsilon_j : \tau \to \tau'$ (**49**). Ce dernier jugement est nécessairement obtenu par la règle (CAS_V) et on a les prémisses suivantes : $\vdash \pi_j : \tau_j; \Delta_j; \Gamma_j$, $\text{dom}(\hat{\Delta}) \# \text{dom}(\Delta_j)$, $\hat{\Delta}_j = \hat{\Delta} \otimes \Delta_j \otimes \tau_j \leqslant \tau$, $\hat{\Delta}_j; \Gamma \oplus \Gamma_j \vdash \varepsilon_j : \sigma_j$ (**50**) et $\hat{\Delta}_j \vdash \sigma_j \leqslant^\sharp \tau'$ (**51**). Fixons nous une valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$ et considérons un jugement quelconque de ${}_{\hat{\rho}}[\![\tau]\!] \vdash \pi_j : t''; G_j$ (**52**). Par le lemme 6.2, on sait qu'il existe $\rho_j \in [\![\Delta_j]\!]$ telle que $G_j \leqslant^\sharp {}_{\rho_j}[\![\Gamma_j]\!]$ (**53**) et $t'' \leqslant {}_{\rho_j}[\![\tau_j]\!] \leqslant {}_{\hat{\rho}}[\![\tau]\!]$. Posons maintenant $\hat{\rho}_j = \hat{\rho} \oplus \rho_j$. Comme $\text{dom}(\hat{\Delta}) \# \text{dom}(\Delta_j)$, $\hat{\rho}_j$ coïncide avec $\hat{\rho}$ sur $\hat{\mathcal{V}}$ et avec ρ_j sur $\mathcal{V}_j$. On a donc ${}_{\hat{\rho}_j}[\![\tau_j]\!] \leqslant {}_{\hat{\rho}_j}[\![\tau]\!]$ et comme $\hat{\rho} \in [\![\hat{\Delta}]\!]$ et $\rho_j \in [\![\Delta_j]\!]$, on a finalement $\hat{\rho}_j \in [\![\hat{\Delta}_j]\!]$. Par le même argument, ${}_{\hat{\rho}_j}[\![\Gamma]\!] = {}_{\hat{\rho}}[\![\Gamma]\!]$, ce qui est en fait valable pour toute $\hat{\rho}_j \in [\![\hat{\Delta}_j]\!]$, ce qui signifie que $\hat{\Delta}_j \vdash \Gamma$. Comme par ailleurs, Γ_j est uniquement composé de monotype, on a aussi $\hat{\Delta}_j \vdash \Gamma \oplus \Gamma_j$ (**54**). On peut maintenant appliquer l'hypothèse d'induction à ε_j sachant (50) et (54). Il vient $\hat{\rho}_j; {}_{\hat{\rho}_j}[\![\Gamma \oplus \Gamma_j]\!] \vdash \varepsilon_j : {}_{\hat{\rho}_j}[\![\sigma_j]\!]$ (**55**). L'environnement de typage ${}_{\hat{\rho}_j}[\![\Gamma \oplus \Gamma_j]\!]$ vaut ${}_{\hat{\rho}_j}[\![\Gamma]\!] \oplus {}_{\hat{\rho}_j}[\![\Gamma_j]\!]$. Compte-tenu du choix des variables, il vaut aussi ${}_{\hat{\rho}}[\![\Gamma]\!] \oplus {}_{\rho_j}[\![\Gamma_j]\!]$. De (53), on déduit alors ${}_{\hat{\rho}}[\![\Gamma]\!] \oplus G_j \leqslant^\sharp {}_{\hat{\rho}}[\![\Gamma]\!] \oplus {}_{\rho_j}[\![\Gamma_j]\!]$ (**56**). Par le lemme 6.1 (subsomption sur les environnements de typage) appliqué à (55) et (56), on sait $\hat{\rho}_j; {}_{\hat{\rho}}[\![\Gamma]\!] \oplus G_j \vdash \varepsilon_j : {}_{\hat{\rho}_j}[\![\sigma_j]\!]$ (**57**). D'après la définition de (51), on sait que ${}_{\hat{\rho}_j}[\![\sigma_j]\!] \leqslant^\sharp {}_{\hat{\rho}_j}[\![\tau']\!]$ (**58**). On peut alors appliquer ($\text{SUB}^\sharp$) à (57) et (58) pour déduire $\hat{\rho}_j; {}_{\hat{\rho}}[\![\Gamma]\!] \oplus G_j \vdash \varepsilon_j : {}_{\hat{\rho}_j}[\![\tau']\!]$ (**59**). Pour être valable, le jugement (1) nécessite que $ftv(\varepsilon) \subseteq \text{dom}(\hat{\Delta})$. Donc τ' ne peut contenir que des variables de $\hat{\Delta}$ et comme $\hat{\rho}_j$ coïncide avec $\hat{\rho}$ sur $\hat{\mathcal{V}}$, on a ${}_{\hat{\rho}_j}[\![\tau']\!] = {}_{\hat{\rho}}[\![\tau']\!]$. Par ailleurs, $ftv(\varepsilon_j) \subseteq ftv(\varepsilon)$, donc le

jugement (59) est aussi valable dans la valuation $\hat{\rho}$. On a finalement $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \oplus G_j \vdash \varepsilon_j : {}_{\hat{\rho}}[\![\tau']\!]$. On a montré ce dernier jugement pour toute instance du jugement (52), ce qui nous permet d'appliquer la règle (CAS_A) et déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \pi_j \Rightarrow \varepsilon_j : {}_{\hat{\rho}}[\![\tau]\!] \to {}_{\hat{\rho}}[\![\tau']\!]$ (**60**). Par ailleurs, la définition de (48) implique ${}_{\hat{\rho}}[\![\tau]\!] \to {}_{\hat{\rho}}[\![\tau']\!] \bowtie \pi_1; \ldots; \pi_k$ (**61**). Les jugements (60) et la propriété (61) permettent finalement de conclure $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : {}_{\hat{\rho}}[\![\tau \to \tau']\!]$. ■

6.7 Complétude

LEMME 6.4. *Pour tout motif π et tout ensemble de variables $\hat{\mathcal{V}}$, on peut calculer un monotype τ, un contexte Δ et un environnement Γ tels que $\vdash \pi : \tau; \Delta; \Gamma$ et $\hat{\mathcal{V}} \# \text{dom}(\Delta)$*

Démonstration. Aucun test algébrique n'intervient dans les règles de typage syntaxique des motifs. Il suffit donc de parcourir mécaniquement la structure du motif π, en choisissant toujours les variables de type « fraîches », c'est-à-dire hors de $\hat{\mathcal{V}}$ et du domaine de tous les contextes Δ engendrés jusqu'au moment où on choisit la variable. ■

LEMME 6.5. *Si $\vdash \pi : \tau; \Delta; \Gamma$, alors, pour tout $\rho \in [\![\Delta]\!]$, ${}_{\rho}[\![\tau]\!] \vdash \pi : {}_{\rho}[\![\tau]\!]; {}_{\rho}[\![\Gamma]\!]$.*

Démonstration. Par induction sur la dérivation de $\vdash \pi : \tau; \Delta; \Gamma$. Considérons la dernière règle R appliquée :

• Si $R = (\pi\text{-UNIVERSEL}_V)$, alors π est le motif universel _ et $\Gamma = \varnothing$. On peut appliquer directement la règle (π-UNIVERSEL) pour déduire directement ${}_{\rho}[\![\tau]\!] \vdash \pi : {}_{\rho}[\![\tau]\!]; \varnothing$.

• Si $R = (\pi\text{-PRIM}_V)$, alors π est un motif primitif ϖ, on a $\Delta = \{\alpha \mid \alpha \in \widetilde{\varpi}\}$, $\tau = \alpha$ et $\Gamma = \varnothing$. Toute valuation $\rho \in [\![\Delta]\!]$ est donc de la forme $[\alpha \mapsto t]$ avec $t \in [\![\widetilde{\varpi}]\!]$. Par l'hypothèse (HYP-$\widetilde{\varpi}$), on sait donc que $t \in \overline{\varpi}$. Par définition de ce dernier ensemble, il existe donc une constante $a \in \varpi$ (**1**) telle que $t = \overline{a}$. En appliquant la règle (π-PRIM) à (1), on obtient $\overline{a} \vdash \varpi : \overline{a}; \varnothing$, c'est-à-dire $t \vdash \varpi : t; \varnothing$. Par ailleurs, comme $\tau = \alpha$, on remarque que $t = {}_{\rho}[\![\tau]\!]$ et on a donc bien ${}_{\rho}[\![\tau]\!] \vdash \varpi : {}_{\rho}[\![\tau]\!]; \varnothing$.

• Si $R = (\pi\text{-LIEUR}_V)$, alors π est de la forme π' **as** x et on a $\Gamma = \Gamma'[x : \tau]$ avec $\vdash \pi' : \tau; \Delta; \Gamma'$. Considérons alors une valuation quelconque $\rho \in [\![\Delta]\!]$. Par l'hypothèse d'induction, ${}_{\rho}[\![\tau]\!] \vdash \pi' : {}_{\rho}[\![\tau]\!]; {}_{\rho}[\![\Gamma']\!]$. En appliquant ($\pi$-LIEUR) à ce dernier jugement, il vient ${}_{\rho}[\![\tau]\!] \vdash \pi : {}_{\rho}[\![\tau]\!]; {}_{\rho}[\![\Gamma']\!][x : {}_{\rho}[\![\tau]\!]]$. On conclut en remarquant que ${}_{\rho}[\![\Gamma']\!][x : {}_{\rho}[\![\tau]\!]] = {}_{\rho}[\![\Gamma]\!]$.

• Si $R = (\pi\text{-CLASSE}_V)$, alors π est de la forme $\#C\,\{\ell_i = \pi_i\}$. Pour tout i, on a $\vdash \pi_i : \tau_i; \Delta_i; \Gamma_i$ (**2**), le monotype τ est une variable α, le contexte Δ vaut $\Delta = \Delta_1 \otimes \cdots \otimes \Delta_n \otimes \{\alpha \mid \kappa\}$ (**3**) avec $\kappa = \alpha \in \widetilde{\#C} \wedge \bigwedge_{i=1}^{n} \alpha \to \tau_i \leqslant^{\exists} \widetilde{C \cdot \ell_i}$ et on a finalement la condition $\text{dom}(\Delta_1) \# \ldots \text{dom}(\Delta_n) \# \alpha$ (**4**). Considérons une valuation quelconque $\rho \in [\![\Delta]\!]$. La définition (3) et la caractérisation des produits de contextes impose que cette valuation soit de la forme $\rho = \rho_1 \oplus \cdots \oplus \rho_n \oplus [\alpha \mapsto t]$ avec chacune des ρ_i dans $[\![\Delta_i]\!]$ et t satisfaisant la contrainte κ, c'est-à-dire ${}_{\rho}[\![\kappa]\!]$ (**5**). Appliquons maintenant l'hypothèse d'induction à chacune des dérivations (2) et des valuations ρ_i. Pour chaque i, on obtient : ${}_{\rho_i}[\![\tau_i]\!] \vdash \pi_i : {}_{\rho_i}[\![\tau_i]\!]; \Gamma_i$. Grâce à la condition (4), on constate que la valuation ρ coïncide avec chaque ρ_i sur le domaine de celle-ci. On a donc aussi ${}_{\rho}[\![\tau_i]\!] \vdash \pi_i : {}_{\rho_i}[\![\tau_i]\!]; {}_{\rho}[\![\Gamma_i]\!]$ (**6**). Par ailleurs, le fait que κ est vraie (interprétation (5)) signifie que $t \in {}_{\rho}[\![\widetilde{\#C}]\!]$ (**7**) et que pour chaque i, $t \to {}_{\rho_i}[\![\tau_i]\!] \leqslant^{\exists} [\![\widetilde{C \cdot \ell_i}]\!]$ (**8**). L'hypothèse (HYP-$\widetilde{\#C}$) appliquée à (7) nous dit que $t \in \overline{\#C}$ (**9**). Par ailleurs, l'hypothèse (HYP-$\widetilde{C \cdot \ell}$) appliqué à (8) nous dit que pour chaque i, il existe un monotype algébrique u_i tel que $t \to {}_{\rho_i}[\![\tau_i]\!] \leqslant u_i \to \overline{C \cdot \ell_i}(u_i)$ (**10**). En appliquant l'axiome (FLÈCHE-VARIANCE) à (10),

il vient ${}_{\rho_i}[\![t_i]\!] \leqslant \overline{C\cdot\ell_i}(u_i)$ (**11**) et $u_i \leqslant t$ (**12**). D'autre part, par (9), on sait que t est dans $\overline{\#C}$, qui est le domaine de $\overline{C\cdot\ell_i}$. Donc $\overline{C\cdot\ell_i}(t)$ est bien défini. En appliquant l'axiome (CHAMPS-COVARIANTS) à (12), il vient alors $\overline{C\cdot\ell_i}(u_i) \leqslant \overline{C\cdot\ell_i}(t)$ (**13**). Par transitivité de la relation de sous-typage appliquée à (11) et (13), on obtient ${}_{\rho_i}[\![t_i]\!] \leqslant \overline{C\cdot\ell_i}(t)$ (**14**). On peut alors appliquer la règle (π-SUB) à (6) et (14) pour déduire $\overline{C\cdot\ell_i}(t) \vdash \pi_i : {}_{\rho_i}[\![\tau_i]\!]; {}_{\rho}[\![\Gamma_i]\!]$ (**15**). Enfin, en appliquant (π-CLASSE) à (9) et à chacun des jugements (15), on déduit $t \vdash \pi : t; {}_{\rho}[\![\Gamma_1]\!] \oplus \cdots \oplus {}_{\rho}[\![\Gamma_n]\!]$. On conclut en remarquant que t vaut ${}_{\rho}[\![\tau]\!]$ et que ${}_{\rho}[\![\Gamma_1]\!] \oplus \cdots \oplus {}_{\rho}[\![\Gamma_n]\!] = {}_{\rho}[\![\Gamma]\!]$.

• Si $R = (\pi\text{-SOUS-CLASSE}_V)$, alors π est de la forme $\#\pi'$ où π' est un motif de la forme $C\{\ldots\}$. Les prémisses de la règle nous disent que le monotype τ est une variable α, qu'il existe un contexte Δ' tel que $\alpha \# \mathrm{dom}(\Delta')$, $\vdash \pi' : \tau; \Delta'; \Gamma$ (**16**) et $\Delta = \Delta' \otimes \{\alpha \mid \alpha \leqslant \tau\}$. Considérons une valuation $\rho \in [\![\Delta]\!]$. Par caractérisation du produit des contextes, ρ est nécessairement de la forme $\rho' \oplus [\alpha \mapsto t]$ où $\rho' \in [\![\Delta']\!]$ et $t \leqslant {}_{\rho'}[\![\tau]\!]$ (**17**). Appliquons l'hypothèse d'induction à (16) et à la valuation ρ'. Il vient ${}_{\rho'}[\![\tau]\!] \vdash \pi' : {}_{\rho'}[\![\tau]\!]; {}_{\rho'}[\![\Gamma]\!]$. Comme ρ et ρ' coïncident sur le domaine de ρ', on a aussi ${}_{\rho}[\![\tau]\!] \vdash \pi' : {}_{\rho}[\![\tau]\!]; {}_{\rho}[\![\Gamma]\!]$ (**18**). Appliquons alors la règle (π-SOUS-CLASSE) à (17) et (18) pour déduire $t \vdash \pi' : t; {}_{\rho}[\![\Gamma]\!]$. On conclut en remarquant que $t = {}_{\rho}[\![\tau]\!]$. ■

On dit que ε est $\hat{\mathcal{V}}$-NORMALE si quand ε est une expression de la forme $\forall\mathcal{V} \mid \kappa \,.\, \varepsilon'$, on a $\hat{\mathcal{V}} \# \mathcal{V}$ et ε' est $\hat{\mathcal{V}} \cup \mathcal{V}$-normale et dans les autres cas toutes les sous-expressions de ε sont également $\hat{\mathcal{V}}$-normale. Intuitivement, une expression ε est $\hat{\mathcal{V}}$-normale s'il n'y a pas de masquage de variables de types dans ε et si toutes les variables introduites sont en dehors de $\hat{\mathcal{V}}$.

LEMME 6.6 (COMPLÉTUDE). *Soit $\hat{\Delta} = \{\hat{\mathcal{V}} \mid \hat{\kappa}\}$ un contexte et ε une expression $\hat{\mathcal{V}}$-normale. Si pour tout $\hat{\rho} \in [\![\hat{\Delta}]\!]$, il existe un polytype algébrique $T_{\hat{\rho}}$ tel que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon : T_{\hat{\rho}}$ (**1**), alors il existe un schéma de type σ tel que $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ et pour tout $\hat{\rho}$, ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^\sharp T_{\hat{\rho}}$.*

Démonstration. Notons que l'ensemble $[\![\hat{\Delta}]\!]$ peut très bien être vide. On procède par induction sur la structure de ε. Remarquons d'abord qu'on peut se limiter au cas où chacun jugement (1) admet une dérivation simple. Si ce n'est pas le cas, on peut en effet appliquer le lemme de simplification 4.2 et trouver, pour chaque $\hat{\rho}$, une dérivation simple de $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash^0 \varepsilon : T'_{\hat{\rho}}$ telle que $T'_{\hat{\rho}} \leqslant^\sharp T_{\hat{\rho}}$. En supposant qu'on puisse en déduire qu'il existe σ tel que $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$ et ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^\sharp T'_{\hat{\rho}}$, alors on a aussi ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^\sharp T_{\hat{\rho}}$, par transitivité de $\leqslant^\sharp$.

Supposons donc que tous les jugements (1) admettent une dérivation simple, ce qu'on note $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash^0 \varepsilon : T_{\hat{\rho}}$ (**2**). La dernière règle appliquée dans ces dérivations est alors déterminée par la nature de l'expression ε et est donc identique pour toutes les valuations $\hat{\rho}$. On examine alors tous les cas possibles pour ε.

• Si ε est de la forme $\forall\mathcal{V} \mid \kappa \,.\, \varepsilon'$, l'hypothèse que ε est $\hat{\mathcal{V}}$-normale signifie que $\hat{\mathcal{V}} \# \mathcal{V}$ et que ε' est $\hat{\mathcal{V}} \cup \mathcal{V}$-normale. Les dérivations simples (2) terminent forcément toutes par la règle (INTRO$_A$) et donc, pour tout $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on a ${}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!] \neq \varnothing$ (**3**). Pour tout $\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!]$, il existe de plus un polytype algébrique $T'_{\hat{\rho},\rho}$ tel que $\hat{\rho} \oplus \rho; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon' : T'_{\hat{\rho},\rho}$ (**4**) et l'ensemble de tous les $T'_{\hat{\rho},\rho}$ est tel que $T_{\hat{\rho}} = \bigcup_{\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V}\mid\kappa\}]\!]} T'_{\hat{\rho},\rho}$ (**5**). Considérons maintenant le contexte $\hat{\Delta}' = \hat{\Delta} \otimes \{\mathcal{V} \mid \kappa\}$. Par la caractérisation des produits de contextes, tout élément $\hat{\rho}' \in [\![\hat{\Delta}']\!]$ est de la forme $\hat{\rho} \oplus \rho$ avec $\hat{\rho} \in [\![\hat{\Delta}]\!]$ et $\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V}\mid\kappa\}]\!]$. Comme $\hat{\rho}$ et $\hat{\rho}'$ coïncident sur $\hat{\mathcal{V}}$, on a d'autre part ${}_{\hat{\rho}}[\![\Gamma]\!] = {}_{\hat{\rho}'}[\![\Gamma]\!]$. On peut donc réécrire le jugement (4) sous la forme $\hat{\rho}'; {}_{\hat{\rho}'}[\![\Gamma]\!] \vdash \varepsilon : T'_{\hat{\rho},\rho}$. Ce dernier jugement a été montré quel que soit $\hat{\rho}' \in [\![\hat{\Delta}']\!]$. On peut donc appliquer l'hypothèse d'induction à ε' et déduire qu'il existe un schéma de type σ' tel que $\hat{\Delta}'; \Gamma \vdash \varepsilon' : \sigma'$ (**6**) et tel que, pour toute $\hat{\rho} \oplus \rho \in [\![\hat{\Delta}']\!]$, on ait ${}_{\hat{\rho}\oplus\rho}[\![\sigma']\!] \leqslant^\sharp T_{\hat{\rho},\rho}$ (**7**). D'autre part, la

propriété (3) signifie, par définition, $\hat{\Delta} \vdash \exists \mathcal{V} \,.\, \kappa$ (**8**). Appliquons maintenant la règle (INTRO_V) à (8) et (6). On déduit $\hat{\Delta}; \Gamma \vdash \varepsilon\colon \sigma$ avec $\sigma = \forall \mathcal{V}, \alpha \mid \kappa \wedge \sigma' \leqslant^{\exists} \alpha \,.\, \alpha$ où α est choisie telle que $\alpha \notin \hat{\mathcal{V}} \cup \mathcal{V}$. Considérons maintenant un élément quelconque $t \in T_{\hat{\rho}}$. Par définition de $T_{\hat{\rho}}$ (équation (5)), on sait qu'il existe $\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!]$ telle que $t \in T'_{\hat{\rho},\rho}$. Par (7), il s'en suit qu'il existe $t' \in {}_{\hat{\rho}\oplus\rho}[\![\sigma']\!]$ tel que $t' \leqslant t$, c'est-à-dire que ${}_{\hat{\rho}\oplus\rho}[\![\sigma']\!] \leqslant^{\exists} t$ (**9**). Posons maintenant $\rho'' = \rho \oplus [\alpha \mapsto t]$. On a $t = {}_{\hat{\rho}\oplus\rho''}[\![\alpha]\!]$, l'inégalité (9) signifie que ${}_{\hat{\rho}\oplus\rho''}[\![\sigma' \leqslant^{\exists} \alpha]\!]$ est vrai et enfin, ${}_{\hat{\rho}\oplus\rho''}[\![\kappa]\!]$ est vrai car $\rho \in {}_{\hat{\rho}}[\![\{\mathcal{V} \mid \kappa\}]\!]$. En fin de compte, on a donc $t \in {}_{\hat{\rho}}[\![\sigma]\!]$. On a donc prouvé que $T_{\hat{\rho}} \subseteq {}_{\hat{\rho}}[\![\sigma]\!]$, ce qui implique ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^{\sharp} T_{\hat{\rho}}$, ce qu'il fallait démontrer.

• Si ε est une application de la forme $\varepsilon_1\,\varepsilon_2$, alors toutes les dérivations simples de (2) terminent forcément par la règle (APP_A). Le polytype $T_{\hat{\rho}}$ est alors un monotype $T_{\hat{\rho}} = t'_{\hat{\rho}}$ (**10**) et on a les jugements $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_1 : t_{\hat{\rho}} \to t'_{\hat{\rho}}$ (**11**) et $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_2 : t_{\hat{\rho}}$ (**12**). Ces deux derniers jugements étant valables quel que soit $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on peut appliquer l'hypothèse d'induction à ε_1 et ε_2. D'une part, il existe donc un schéma de type σ_1 tel que $\hat{\Delta}; \Gamma \vdash \varepsilon_1\colon \sigma_1$ (**13**) et pour tout $\hat{\rho} \in [\![\hat{\Delta}]\!]$, ${}_{\hat{\rho}}[\![\sigma_1]\!] \leqslant^{\sharp} t_{\hat{\rho}} \to t'_{\hat{\rho}}$ (**14**). D'autre part, il existe σ_2 tel que $\hat{\Delta}; \Gamma \vdash \varepsilon_2\colon \sigma_2$ (**15**) et pour tout $\hat{\rho}$, ${}_{\hat{\rho}}[\![\sigma_2]\!] \leqslant^{\sharp} t_{\hat{\rho}}$ (**16**). Choisissons maintenant deux variables de type α et β fraîches , c'est-à-dire telles que $\alpha \# \beta \# \hat{\mathcal{V}}$. Posons $\rho = [\alpha \mapsto t_{\hat{\rho}}; \beta \mapsto t'_{\hat{\rho}}]$. Les inéquations (14) et (16) montrent que l'interprétation ${}_{\hat{\rho}\oplus\rho}[\![\sigma_1 \leqslant^{\exists} \alpha \to \beta \wedge \sigma_2 \leqslant^{\exists} \alpha]\!]$ (**17**) est vraie. Posons $\sigma = \forall \alpha\beta \,|\, \sigma_1 \leqslant^{\exists} \alpha \to \beta \wedge \sigma_2 \leqslant^{\exists} \alpha \,.\, \beta$. D'après (17), on sait que $t'_{\hat{\rho}} \in {}_{\hat{\rho}}[\![\sigma]\!]$, ce qu'on peut aussi écrire ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^{\sharp} t'_{\hat{\rho}}$ (**18**). L'interprétation du schéma σ dans $\hat{\rho}$ est donc non-vide (**19**). En appliquant la règle (APP_V) à (13), (15) et (19), on conclut $\hat{\Delta}; \Gamma \vdash \varepsilon_1\,\varepsilon_2\colon \sigma$.

• Si ε est une fonction de la forme $\texttt{fun}\ x\colon \tau\ \Rightarrow \varepsilon'$, alors la dernière règle appliquée dans toutes les dérivation de (2) est forcément (FUN_A). Pour toute $\hat{\rho} \in [\![\hat{\Delta}]\!]$, le polytype $T_{\hat{\rho}}$ est donc de la forme ${}_{\hat{\rho}}[\![\tau]\!] \to t_{\hat{\rho}}$ (**20**) et on a $\hat{\rho}; G[x\colon {}_{\hat{\rho}}[\![\tau]\!]] \vdash \varepsilon' : t_{\hat{\rho}}$ (**21**). On peut appliquer l'hypothèse d'induction à ε' et à (21) et déduire qu'il existe un schéma de type σ' tel que $\hat{\Delta}; \Gamma[x\colon \tau] \vdash \varepsilon'\colon \sigma'$ (**22**) et tel que pour toute $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on ait ${}_{\hat{\rho}}[\![\sigma']\!] \leqslant^{\sharp} t_{\hat{\rho}}$ (**23**). Choisissons une variable de type α fraîche, c'est-à-dire telle que $\alpha \# \hat{\mathcal{V}}$. En appliquant la règle (FUN_V) à (22) et en posant $\sigma = \forall \alpha \mid \sigma' \leqslant^{\exists} \alpha \,.\, \tau \to \alpha$, il vient $\hat{\Delta}; \Gamma \vdash \varepsilon\colon \sigma$ (**24**). Prenons maintenant une valuation quelconque $\hat{\rho} \in [\![\hat{\Delta}]\!]$. (23) peut aussi s'écrire ${}_{\hat{\rho}}[\![\sigma']\!] \leqslant^{\exists} t_{\hat{\rho}}$ (**25**). En utilisant la valuation $\rho = [\alpha \mapsto t_{\hat{\rho}}$, on constate que $t_{\hat{\rho}} \in {}_{\hat{\rho}}[\![\sigma]\!]$, c'est-à-dire que ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^{\sharp} T_{\hat{\rho}}$.

• Si ε est une variable, le résultat est trivial.

• Si ε est un accesseur de la forme $C\cdot\ell$, alors on a toujours la dérivation syntaxique $\hat{\Delta}; \Gamma \vdash C\cdot\ell\colon \widetilde{C\cdot\ell}$. Par ailleurs, la dernière règle appliquée dans toutes les dérivations (2) étant forcément (GET_A), on a $T_{\hat{\rho}} = t \to \overline{C\cdot\ell}(t)$ pour un certain monotype t. Par l'hypothèse ($\text{HYP-}\widetilde{C\cdot\ell}$), on a alors $T_{\hat{\rho}} \in [\![\sigma]\!]$ et donc ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^{\sharp} T_{\hat{\rho}}$.

• Si ε est de la forme $\texttt{let}\ x = \varepsilon_1\ \texttt{in}\ \varepsilon_2$, alors les dérivations simples de (2) terminent forcément par (LET_A). Pour toute valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$, il existe donc $T^1_{\hat{\rho}}$ tel que $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_1 : T^1_{\hat{\rho}}$ (**26**) et $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!][x\colon T^1_{\hat{\rho}}] \vdash \varepsilon_2 : T_{\hat{\rho}}$. Appliquons l'hypothèse d'induction à ε_1 et à (26). On peut donc trouver un schéma σ_1 tel que $\hat{\Delta}; \Gamma \vdash \varepsilon_1\colon \sigma_1$ (**27**) et ${}_{\hat{\rho}}[\![\sigma_1]\!] \leqslant^{\sharp} T^1_{\hat{\rho}}$ (**28**). De cette dernière inégalité, on déduit que ${}_{\hat{\rho}}[\![\Gamma[x\colon \sigma_1]]\!] \leqslant^{\sharp} {}_{\hat{\rho}}[\![\Gamma]\!][x\colon T^1_{\hat{\rho}}]$ (**29**). On peut alors appliquer le lemme 6.1 à (27) et (29), ce qui nous donne $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma[x\colon \sigma_1]]\!] \vdash \varepsilon_2 : T_{\hat{\rho}}$ (**30**). Appliquons maintenant l'hypothèse d'induction à ε_2 et (30). On peut donc trouver un schéma σ_2 tel que $\hat{\Delta}; \Gamma[x\colon \sigma_1] \vdash \varepsilon_2\colon \sigma_2$ (**31**) et tel que pour toute $\hat{\rho}$, on ait ${}_{\hat{\rho}}[\![\sigma_2]\!] \leqslant^{\sharp} T_{\hat{\rho}}$. On conclut finalement en appliquant la règle (LET_V) à (27) et (31).

• Si ε est une expression récursive de la forme $\texttt{fix}\ x\colon \sigma\ \Rightarrow \varepsilon'$, alors toutes les dérivations (2) se terminent forcément par la règle (FIX_A). On a donc $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma[x\colon \sigma]]\!] \vdash \varepsilon' : {}_{\hat{\rho}}[\![\sigma]\!]$ (**32**). Appliquons

l'hypothèse d'induction à ε' et à (32). Il existe donc un schéma σ' tel que $\hat{\Delta}; \Gamma[x:\sigma] \vdash \varepsilon' : \sigma'$ (**33**) et ${}_{\hat{\rho}}[\![\sigma']\!] \leqslant^\sharp {}_{\hat{\rho}}[\![\sigma]\!]$ (**34**). Du jugement (32), on déduit que ${}_{\hat{\rho}}[\![\sigma]\!]$ n'est jamais vide. On a donc $\hat{\Delta} \vdash \sigma$ (**35**). D'autre part, la conjonction de toutes les inéquations (34) est par définition équivalente à $\hat{\Delta} \vdash \sigma' \leqslant^\sharp \sigma$ (**36**). On peut finalement appliquer la règle (FIX$_V$) à (33), (35) et (36) pour conclure $\hat{\Delta}; \Gamma[x:\sigma] \vdash \varepsilon : \sigma$.

• Si ε est un objet de la forme $C\,\{\ell_i = \varepsilon_i\}$, alors toutes les dérivations simples (2) se terminent forcément par la règle (OBJET$_A$). Pour toute valuation $\hat{\rho} \in [\![\hat{\Delta}]\!]$, le polytype $T_{\hat{\rho}}$ est un monotype $t_{\hat{\rho}} \in \overline{\#C}$ et pour tout i, on a $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash \varepsilon_i : \overline{C\cdot\ell_i}(t_{\hat{\rho}})$ (**37**). Appliquons l'hypothèse d'induction à chacune des sous-expressions ε_i et à (37). Il existe donc pour chaque i un schéma σ_i tel que $\hat{\Delta}; \Gamma \vdash \varepsilon_i : \sigma_i$ (**38**) et ${}_{\hat{\rho}}[\![\sigma_i]\!] \leqslant^\sharp \overline{C\cdot\ell_i}(t_{\hat{\rho}})$. Choisissons deux variables α et β fraîches, c'est-à-dire telles que $\alpha \# \beta \# \hat{\mathcal{V}}$ et considérons le schéma suivant :

$$\sigma = \forall\alpha \mid \alpha \in \widetilde{\#C} \wedge \bigwedge_{i=1}^{n} \exists\beta\,.\,(\sigma_i \leqslant^\exists \beta \wedge \alpha \to \beta \leqslant^\exists \widetilde{C\cdot\ell})\,.\,\alpha \qquad \textbf{(39)}$$

En posant $\rho = [\alpha \mapsto t_{\hat{\rho}}]$, on a clairement ${}_{\hat{\rho}\oplus\rho}[\![\alpha \in \widetilde{\#C}]\!]$ et, pour chaque i, $t_{\hat{\rho}} \to \overline{C\cdot\ell_i}(t_{\hat{\rho}})$ est par l'hypothèse (HYP-$\widetilde{C\cdot\ell}$) dans $[\![\widetilde{C\cdot\ell_i}]\!]$. La valuation $\rho' = [\beta \mapsto \overline{C\cdot\ell_i}(t_{\hat{\rho}})]$ nous montre donc que l'interprétation ${}_{\hat{\rho}}[\![\sigma_1 \leqslant^\exists \beta \wedge \alpha \to \beta \leqslant^\exists \widetilde{C\cdot\ell_i}]\!]$ est vraie. En fin de compte, on a donc montré que $t_{\hat{\rho}} \in {}_{\hat{\rho}}[\![\sigma]\!]$ et donc que ${}_{\hat{\rho}}[\![\sigma]\!] \leqslant^\sharp t_{\hat{\rho}}$ (**40**). Cela prouve que σ est bien non-vide dans toute valuation $\hat{\rho}$, c'est-à-dire que $\hat{\Delta} \vdash \sigma$ (**41**). En appliquant la règle (OBJET$_V$) à (38) et (41), on conclut $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$.

• Si ε est une constante a, les dérivations simples de (2) terminent forcément par (CONSTANTE$_A$) et on a $T_{\hat{\rho}} = \overline{a}$. Par ailleurs, on a toujours $\hat{\Delta}; \Gamma \vdash \varepsilon : \widetilde{a}$, par la règle (CST$_V$). On peut conclure en remarquant que, par l'hypothèse (HYP-$\widetilde{a}$), $[\![\widetilde{a}]\!] = \overline{a}$.

• Si ε est une primitive f, les dérivations simples de (2) terminent forcément par (PRIMITIVE$_A$). $T_{\hat{\rho}}$ est donc un monotype élément de $\overline{f}$. Par ailleurs on a toujours $\hat{\Delta}; \Gamma \vdash \varepsilon : \widetilde{f}$, par la règle (PRIM$_V$). Par l'hypothèse (HYP-$\widetilde{f}$), on conclut que $[\![\widetilde{f}]\!] \leqslant^\sharp T_{\hat{\rho}}$.

• Il reste le cas où ε est une méthode de la forme $\texttt{meth}: \tau \to \tau' \{\pi_j \Rightarrow \varepsilon_j; \ldots\}$. Les dérivations simples de (2) se terminent forcément par la règle (METH$_A$). Pour toute $\hat{\rho} \in [\![\hat{\Delta}]\!]$, le polytype $T_{\hat{\rho}}$ est donc égal au monotype ${}_{\hat{\rho}}[\![\tau]\!] \to {}_{\hat{\rho}}[\![\tau']\!]$ (**42**), on a le test de couverture ${}_{\hat{\rho}}[\![\tau \to \tau']\!] \bowtie \pi_1; \ldots; \pi_k$ (**43**) et pour tout j, on a $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \vdash (\pi_j \Rightarrow \varepsilon_j) : {}_{\hat{\rho}}[\![\tau]\!] \to {}_{\hat{\rho}}[\![\tau']\!]$ (**44**) Fixons maintenant un indice j. Par la règle (CAS$_A$), le jugement (44) est vrai si et seulement si :

$$\forall t'_j, G_j, \quad (\hat{\rho}[\![\tau]\!] \vdash \pi_j : t'_j; G_j) \Rightarrow (\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \oplus G_j \vdash \varepsilon_j : {}_{\hat{\rho}}[\![\tau']\!]) \qquad \textbf{(45)}$$

Par le lemme 6.4, on sait par ailleurs qu'il existe τ_j, Δ_j et Γ_j tels que $\Delta_j \# \hat{\Delta}$ (**46**) et $\vdash \pi_j : \tau_j; \Delta_j; \Gamma_j$ (**47**). Posons $\hat{\Delta}_j = \hat{\Delta} \otimes \Delta_j \otimes \tau_j \leqslant \tau$ et considérons une valuation $\hat{\rho}_j$ quelconque dans $[\![\hat{\Delta}_j]\!]$. Par la caractérisation des produits, $\hat{\rho}_j$ est nécessairement de la forme $\hat{\rho} \oplus \rho_j$ avec $\hat{\rho} \in [\![\hat{\Delta}]\!]$, $\rho_j \in [\![\Delta_j]\!]$ et ${}_{\rho_j}[\![\tau_j]\!] \leqslant {}_{\hat{\rho}}[\![\tau]\!]$ (**48**). Appliquons maintenant le lemme 6.5 à la valuation $\rho_j \in [\![\Delta_j]\!]$ et au jugement (47). On obtient ${}_{\rho_j}[\![\tau_j]\!] \vdash \pi_j : {}_{\rho_j}[\![\tau_j]\!]; {}_{\rho_j}[\![\Gamma_j]\!]$ (**49**). En appliquant la règle (π-SUB) à (48) et à (49), il vient ${}_{\hat{\rho}}[\![\tau]\!] \vdash \pi_j : {}_{\rho_j}[\![\tau_j]\!]; {}_{\rho_j}[\![\Gamma_j]\!]$ (**50**). L'implication (45) appliquée à (50) nous permet alors de déduire $\hat{\rho}; {}_{\hat{\rho}}[\![\Gamma]\!] \oplus {}_{\rho_j}[\![\Gamma_j]\!] \vdash \varepsilon_j : {}_{\hat{\rho}}[\![\tau']\!]$ (**51**) Comme $\hat{\rho} \# \rho_j$, l'environnement dans ce dernier jugement est égal à ${}_{\hat{\rho}_j}[\![\Gamma \oplus \Gamma_j]\!]$. Par ailleurs, le jugement (51) est également valide dans la valuation $\hat{\rho}_j$ car celle-ci coïncide avec $\hat{\rho}$ sur le domaine de $\hat{\rho}$. Finalement, on peut donc écrire $\hat{\rho}_j; {}_{\hat{\rho}_j}[\![\Gamma \oplus \Gamma_j]\!] \vdash \varepsilon_j : {}_{\hat{\rho}_j}[\![\tau']\!]$ (**52**). Le jugement (52) a été montré pour toute valuation $\hat{\rho}_j \in [\![\hat{\Delta}_j]\!]$. On peut appliquer l'hypothèse d'induction à ε_j et à (52). Il existe donc un schéma de type σ_j tel que $\hat{\Delta}_j; \Gamma \oplus \Gamma_j \vdash \varepsilon_j : \sigma_j$ (**53**) et tel que pour toute valuation $\hat{\rho}_j \in [\![\hat{\Delta}_j]\!]$, on ait ${}_{\hat{\rho}_j}[\![\sigma_j]\!] \leqslant^\sharp {}_{\hat{\rho}_j}[\![\tau']\!]$ (**54**). Les inégalités (54) étant valables

pour toute valuation $\hat{\rho}_j$, on a par définition $\hat{\rho}_j \vdash \sigma_j \leqslant^\sharp \tau'$ (**55**). On peut maintenant appliquer la règle syntaxique (CAS_V) à (46), (47), (53) et (55) pour déduire $\hat{\Delta}; \Gamma \vdash \pi_j \Rightarrow \varepsilon_j : \tau \to \tau'$ (**56**). Par ailleurs, le test de couverture (43) étant valable pour toute les valuations $\hat{\rho} \in [\![\hat{\Delta}]\!]$, on a donc par définition $\hat{\Delta} \vdash \tau \to \tau' \bowtie \pi_1; \ldots; \pi_k$ (**57**). En appliquant la règle (METH_V) à (57) et à tous les jugements (56), on conclut $\hat{\Delta}; \Gamma \vdash \varepsilon : \sigma$. ■

6.8 Traitement des masquages

Dans le lemme de complétude 6.5, on a supposé que l'expression ε est $\hat{\mathcal{V}}$-normale, ce qui permet de traiter commodémment l'introduction de variables de type. Nous allons maintenant montrer comment toute expression peut être normalisée sans que cela change son typage dans le système semi-algébrique. L'idée de la preuve consiste simplement à parcourir l'expression en renommant toutes les variables introduites de façon à ce qu'un même nom de variable ne soit jamais utilisé pour référencer deux variables différentes. Un compilateur effectuerait naturellement cette opération dans une première passe d'analyse syntaxique, quand il remplace les identificateurs dans la syntaxe concrète par des références vers des structures internes différentes pour chaque variable dans la syntaxe abstraite. Il convient simplement de vérifier que cette opération ne change rien au typage, ce qui est fastidieux, mais sans aucune difficulté.

Dans cette section, on appelera ÉLÉMENT SYNTAXIQUE DE TYPAGE, qu'on notera θ, un monotype syntaxique, une contrainte, un schéma de types ou une expression annotée. Il est commode de définir l'interprétation d'une expression annotée ε relativement à une certaine valuation $\hat{\rho}$ comme l'ensemble des couples (G, T) tels que ε ait le type σ dans l'environnement G et la valuation $\hat{\rho}$:

$$(6.18) \qquad {}_{\hat{\rho}}[\![\varepsilon]\!] = \left\{(G, T) \mid \hat{\rho}; G \vdash \varepsilon : T\right\}$$

On appelle RENOMMAGE DE VARIABLES DE TYPES une fonction partielle injective à domaine finie de VarsType vers VarsType. Si $\mathcal{V}_1$ est le domaine d'un renommage R et $\mathcal{V}_2$ son codomaine, alors R^{-1} est un renommage de domaine $\mathcal{V}_2$ et de codomaine $\mathcal{V}_1$. Si $\mathcal{V}_1$ et $\mathcal{V}_2$ sont deux ensembles finis de variables de types, alors il existe un renommage de domaine $\mathcal{V}_1$ et de codomaine disjoint de $\mathcal{V}_2$, car VarsType est supposé infini. On peut de plus imposer que ce renommage laisse inchangées les variables qui sont dans $\mathcal{V}_1 \setminus \mathcal{V}_2$. Un de ces renommage est choisi arbitrairement et noté $\mathcal{R}(\mathcal{V}_1, \mathcal{V}_2)$. On a donc :

$$\begin{aligned} \text{dom}(\mathcal{R}(\mathcal{V}_1, \mathcal{V}_2)) &= \mathcal{V}_1 \\ \text{codom}(\mathcal{R}(\mathcal{V}_1, \mathcal{V}_2)) &\# \mathcal{V}_2 \\ \mathcal{R}(\mathcal{V}_1, \mathcal{V}_2)(\alpha) &= \alpha \qquad \text{si } \alpha \in \mathcal{V}_1 \text{ et } \alpha \notin \mathcal{V}_2 \end{aligned}$$

Un renommage s'applique à tout élément de typage. Pour simplifier, on supposera quand on applique le renommage R à l'élément θ qu'on a toujours $\mathit{ftv}(\theta) \subseteq \text{dom}(R)$. L'application du renommage

R à l'élément θ est notée $[R]\theta$ et est définie par les équations suivantes :

$$
\begin{array}{rcl}
[\hat{R}]\alpha & \triangleq & \hat{R}(\alpha) \\
[\hat{R}](\tau \to \tau') & \triangleq & [\hat{R}]\tau \to [\hat{R}]\tau' \\
[\hat{R}](c\langle\tau_1, \ldots, \tau_n\rangle) & \triangleq & c\langle[\hat{R}]\tau_1, \ldots, [\hat{R}]\tau_n\rangle \\
[\hat{R}]\mathsf{vrai} & \triangleq & \mathsf{vrai} \\
[\hat{R}](\tau \leqslant \tau') & \triangleq & [\hat{R}]\tau \leqslant [\hat{R}]\tau' \\
[\hat{R}](p\langle\tau_1, \ldots, \tau_n\rangle) & \triangleq & p\langle[\hat{R}]\tau_1, \ldots, [\hat{R}]\tau_n\rangle \\
[\hat{R}](\kappa \wedge \kappa') & \triangleq & [\hat{R}]\kappa \wedge [\hat{R}]\kappa' \\
[\hat{R}](\exists\mathcal{V} \,.\, \kappa) & \triangleq & \exists R(\mathcal{V}) \,.\, [\hat{R} \oplus R]\kappa \qquad \text{où } R = \mathcal{R}(\mathcal{V}, \mathrm{codom}(\hat{R})) \\
[\hat{R}](\forall\mathcal{V} \mid \kappa \,.\, \tau) & \triangleq & \forall R(\mathcal{V}) \mid [\hat{R} \oplus R]\kappa \,.\, [\hat{R} \oplus R]\tau \qquad \text{où } R = \mathcal{R}(\mathcal{V}, \mathrm{codom}(\hat{R}))
\end{array}
$$

$$
\begin{array}{rcl}
[\hat{R}]x & \triangleq & x \\
[\hat{R}](\varepsilon_1\, \varepsilon_2) & \triangleq & [R]\varepsilon_1\, [\hat{R}]\varepsilon_2 \\
[\hat{R}](\mathbf{fun}\, x : \tau \Rightarrow \varepsilon) & \triangleq & \mathbf{fun}\, x : [\hat{R}]\tau \Rightarrow [\hat{R}]\varepsilon \\
[\hat{R}](\mathbf{let}\, x = \varepsilon_1\, \mathbf{in}\, \varepsilon_2) & \triangleq & \mathbf{let}\, x = [\hat{R}]\varepsilon_1\, \mathbf{in}\, [\hat{R}]\varepsilon_2 \\
[\hat{R}](\mathbf{fix}\, x : \sigma \Rightarrow \varepsilon) & \triangleq & \mathbf{fix}\, x : [\hat{R}]\sigma \Rightarrow [\hat{R}]\varepsilon \\
[\hat{R}](C\,\{\ell_i = \varepsilon_i\}) & \triangleq & C\left\{\ell_i = [\hat{R}]\varepsilon_i\right\} \\
[\hat{R}]C\cdot\ell & \triangleq & C\cdot\ell \\
[\hat{R}]a & \triangleq & a \\
[\hat{R}]f & \triangleq & f \\
[\hat{R}](\mathbf{meth} : \tau \to \tau'\left\{\pi_j \Rightarrow \varepsilon_j\right\}) & \triangleq & \mathbf{meth} : [\hat{R}]\tau \to [\hat{R}]\tau'\left\{\pi_j \Rightarrow [\hat{R}]\varepsilon_j\right\} \\
[\hat{R}](\forall\mathcal{V} \mid \kappa \,.\, \varepsilon) & \triangleq & \forall R(\mathcal{V}) \mid [\hat{R} \oplus R]\kappa \,.\, [\hat{R} \oplus R]\varepsilon \qquad \text{où } R = \mathcal{R}(\mathcal{V}, \mathrm{codom}(\hat{R}))
\end{array}
$$

Les lemme suivant expriment la propriété d'invariance attendue de l'application d'un renommage :

LEMME 6.7. *Pour tout élément de typage θ, tout renommage R tel que $ftv(\theta) \subseteq \mathrm{dom}(R)$, et toute valuation ρ telle que* $\mathrm{codom}(R) \subseteq \mathrm{dom}(\rho)$*, alors* ${}_{\rho\circ R}[\![\theta]\!]$ *et* ${}_{\rho}[\![[R]\theta]\!]$ *sont tous les deux bien définis et égaux.*

Démonstration. Si θ est un monotype, la preuve se fait par induction sur sa structure. Dans le cas où c'est une variable α, on vérifie simplement que, d'après les hypothèses, $\rho(R(\alpha))$ est bien défini et est égal aux deux interprétations qu'on veut comparer. Les autres cas sont triviaux.

Si θ est une contrainte, on procède de même par induction sur sa structure. Le seul cas non-trivial est celui où la contrainte vaut $\exists\mathcal{V} \,.\, \kappa$. Notons alors $\hat{R} = \mathcal{R}(\mathcal{V}, \mathrm{codom}(R))$. Prenons une valuation quelconque $\hat{\rho}$ de domaine $\hat{\mathcal{V}} = \hat{R}(\mathcal{V})$. On vérifie facilement que $ftv(\kappa) \subseteq \mathrm{dom}(R \oplus \hat{R})$ et que $\mathrm{codom}(R \oplus \hat{R}) \subseteq \mathrm{dom}(\rho \oplus \hat{\rho})$. Par l'hypothèse d'induction, ${}_{\rho\oplus\hat{\rho}}[\![[R \oplus \hat{R}]\kappa]\!]$ est donc équivalent à ${}_{(\rho\oplus\hat{\rho})\circ(R\oplus\hat{R})}[\![\kappa]\!]$. Montrons maintenant que la valuation $\rho_1 = (\rho \oplus \hat{\rho}) \circ (R \oplus \hat{R})$ coïncide avec

$\rho_2 = (\rho \circ R) \oplus (\hat{\rho} \circ \hat{R})$ sur toutes les variables $\alpha \in ftv(\kappa)$. En effet, si $\alpha \in \mathcal{V}$, alors α est dans le domaine de $\hat{R}$, $\hat{R}(\alpha)$ dans celui de $\hat{\rho}$ et ρ_1 et ρ_2 envoient toutes deux α sur $\hat{\rho}(\hat{R}(\alpha))$. Si en revanche $\alpha \notin \mathcal{V}$, alors α n'est pas dans le domaine de $\hat{R}$, mais $\alpha \in ftv(\exists \mathcal{V} \,.\, \kappa)$. Par hypothèse, on a donc $\alpha \in \mathrm{dom}(R)$ et $R(\alpha) \in \mathrm{dom}(\rho)$. $\rho_2(\alpha)$ vaut donc $\rho(\hat{R}(\alpha))$. Par ailleurs $R(\alpha)$ ne peut être dans $\mathrm{dom}(\hat{\rho})$ car ce dernier ensemble est aussi le codomaine de $\hat{R}$ et celui-ci est supposé disjoint de $\mathrm{codom}(R)$. Donc $\rho_1(\alpha)$ vaut également $\rho(\hat{R}(\alpha))$. Les valuations ρ_1 et ρ_2 coïncidant sur les variables libres de κ, on sait que ${}_{\rho_1}[\![\kappa]\!]$ est équivalent à ${}_{\rho_2}[\![\kappa]\!]$. On a donc montré que pour toute valuation $\hat{\rho}$ de domaine $\hat{\mathcal{V}}$, la valeur de vérité de ${}_{\rho \oplus \hat{\rho}}[\![[R \oplus \hat{R}]\kappa]\!]$ (**1**) est équivalente à celle de ${}_{(\rho \circ R) \oplus (\hat{\rho} \circ \hat{R})}[\![\kappa]\!]$ (**2**). Maintenant, si ${}_{\rho}[\![[R](\exists \mathcal{V} \,.\, \kappa)]\!]$ est vrai, alors il existe $\hat{\rho}$ qui vérifie (1), donc qui vérifie aussi (2). Alors, $\hat{\rho} \circ \hat{R}$ est une solution qui montre que ${}_{\rho \circ R}[\![\exists \mathcal{V} \,.\, \kappa]\!]$ est vrai. Réciproquement, si ${}_{\rho \circ R}[\![\exists \mathcal{V} \,.\, \kappa]\!]$ est vrai, il existe ρ' de domaine $\mathcal{V}$ tel que ${}_{(\rho \circ R) \oplus \rho'}[\![\kappa]\!]$ est vrai. En posant $\hat{\rho} = \rho' \circ \hat{R}^{-1}$, on obtient (2), donc (1) et donc ${}_{\rho}[\![[R](\exists \mathcal{V} \,.\, \kappa)]\!]$ est vrai.

Si θ est un schéma de types, on utilise le même argument que pour les contraintes de la forme $\exists \mathcal{V} \,.\, \kappa$.

Si θ est une expression annotée, il s'agit de montrer que pour tout G et T, le jugement $\rho; G \vdash [R]\varepsilon : T$ est équivalent au jugement $\rho \circ R; G \vdash \varepsilon : T$. Montrons d'abord qu'on peut se contenter de vérifier l'équivalence pour les dérivations simples de ces jugements. En effet, si on considère une dérivation quelconque du premier jugement, alors il existe une dérivation simple de la forme $\rho; G \vdash [R]\varepsilon : T'$ avec $T' \leqslant^\sharp T$. En supposant que cela nous permette de déduire $\rho \circ R; G \vdash \varepsilon : T'$, il s'en suit alors le jugement $\rho \circ R; G \vdash \varepsilon : T$. La réciproque est clairement vraie également. On procède ensuite par induction sur la structure de l'expression ε et on montre l'équivalence des deux jugements pour pour toute valuation ρ et tout (G, T). Quand ε ne contient pas d'annotation de type, la récursion est triviale. Quand ε contient une annotation de type, on peut utiliser les résultats prouvés plus haut pour conclure. Le seul cas non-évident qui reste est donc quand l'expression est de la forme $\forall \mathcal{V} \mid \kappa \,.\, \varepsilon$. Posons alors $\hat{R} = \mathcal{R}(\mathcal{V}, \mathrm{codom}(R))$ et $\hat{\mathcal{V}} = \hat{R}(\mathcal{V})$. Par un argument identique à celui utilisé pour le traitement de la contrainte $\exists \mathcal{V} \,.\, \kappa$, les deux jugements suivants sont toujours équivalents si $\hat{\rho}$ est une valuation de domaine $\hat{\mathcal{V}}$:

$$\rho \oplus \hat{\rho}; G \vdash [R \oplus \hat{R}]\varepsilon : T_{\hat{\rho}} \iff (\rho \circ R) \oplus (\hat{\rho} \circ \hat{R}); G \vdash \varepsilon : T_{\hat{\rho}} \tag{3}$$

De même, pour toute telle valuation $\hat{\rho}$, on a aussi :

$$\hat{\rho} \in {}_{\rho}[\![\{\hat{\mathcal{V}} \mid [R \oplus \hat{R}]\kappa\}]\!] \iff \hat{\rho} \circ \hat{R} \in {}_{\rho \circ R}[\![\{\mathcal{V} \mid \kappa\}]\!] \tag{4}$$

Supposons maintenant que le jugement $\rho; G \vdash [R](\forall \mathcal{V} \mid \kappa \,.\, \varepsilon) : T$ soit vrai et qu'il admette une dérivation simple. Cette dérivation termine forcément par la règle ($\textsc{Intro}_A$) et on a :

$$ {}_{\rho}[\![\{\hat{\mathcal{V}} \mid [R \oplus \hat{R}]\kappa\}]\!] \neq \varnothing \tag{5}$$

$$\forall \hat{\rho} \in {}_{\rho}[\![\{\hat{\mathcal{V}} \mid [R \oplus \hat{R}]\kappa\}]\!], \rho \oplus \hat{\rho}; G \vdash [R \oplus \hat{R}]\varepsilon : T_{\hat{\rho}} \tag{6}$$

$$T = \bigcup T_{\hat{\rho}} \tag{7}$$

Par (4), on sait que l'ensemble ${}_{\rho \circ R}[\![\{\mathcal{V} \mid \kappa\}]\!]$ n'est pas vide car il contient $\hat{\rho} \circ \hat{R}$. Par ailleurs, tout élément $\rho'' \in {}_{\rho \circ R}[\![\{\mathcal{V} \mid \kappa\}]\!]$ peut s'écrire sous la forme $\hat{\rho} \circ \hat{R}$ en posant $\hat{\rho} = \rho'' \circ \hat{R}^{-1}$. Donc, pour toute telle ρ'', on a, d'après (6) et (3), $(\rho \circ R) \oplus \rho''; G \vdash \varepsilon : T_{\rho'' \circ \hat{R}^{-1}}$. En appliquant la règle ($\textsc{Intro}_A$), il vient que $\rho \circ; G \vdash \forall \mathcal{V} \mid \kappa \,.\, \varepsilon : T$. La réciproque se traite de la même façon. ■

Comme son nom l'indique, un renommage a pour effet de renommer les variables libres d'un élément syntaxique, mais, appliqué à une expression annotée, il peut également servir à normaliser l'expression :

LEMME 6.8. *Si $\hat{R}$ est un renommage de domaine $\hat{\mathcal{V}}$ et ftv$(\varepsilon) \subseteq \hat{\mathcal{V}}$, alors $[R]\varepsilon$ est une expression* codom$(\hat{R})$*-normale.*

Démonstration. Découle de la définition de l'application de $\hat{R}$ à une expression de la forme $\forall\mathcal{V} \mid \kappa \,.\, \varepsilon$. Celle-ci est transformée en $\forall R(\mathcal{V}) \mid [\hat{R} \oplus R]\kappa \,.\, [\hat{R} \oplus R]\varepsilon$ et le codomaine $R(\mathcal{V})$ est supposé disjoint de codom$(\hat{R})$. Pour une preuve complète, il suffit de procéder par induction sur la structure de ε. ■

6.9 Conclusion

Démonstration du théorème 3. Nous composons maintenant l'ensemble des lemmes pour prouver le théorème et décrire l'algorithme de réduction.

Étant donnée une expression ε close (sans variable ni variable de type libre), on commence par calculer l'expression $\varepsilon_0 = []\varepsilon$ (l'application du renommage vide à ε). Les définitions du renommage montrent que cette opération est bien calculable et le lemme 6.8 que ε_0 est $\varnothing$-normale.

On peut ensuite parcourir la structure de ε_0 et collecter des problèmes de typage en consultant pour chaque nœud de ε_0 la règle du système de types syntaxique correspondante. Par exemple, si ε_0 est de la forme $\varepsilon_1\,\varepsilon_2$, on ajoute $\hat{\Delta} \vdash \forall\alpha\beta \mid \sigma_1 \leqslant^{\exists} \alpha \to \beta \wedge \sigma_2 \leqslant^{\exists} \alpha\,.\,\beta$ dans la liste des problèmes à résoudre. Durant cette phase, on maintient un environnement formé d'un contexte $\hat{\Delta}$ et d'un environnement syntaxique Γ. On se contente de maintenir cette environnement et de générer les problèmes mécaniquement sans chercher à les interpréter et indépendamment d'un modèle. Dans la règle (INTRO$_V$), l'exigence dom$(\hat{\Delta}) \# \mathcal{V}$ est toujours réalisée car ε_0 est normale. Quand on doit traiter un motif, on utilise le lemme 6.4 pour assurer que les variables de type choisies comme domaine du contexte Δ_j ne posent pas de problèmes, par exemple en demandant qu'elles soient en dehors de l'ensemble de toutes les variables (libres ou liées) de ε_0. Finalement, on se convainc donc aisément qu'il est possible de calculer à partir de ε_0 de façon complètement mécanique et syntaxique une liste de problèmes de typage $P_1, \ldots, P_N$ et un schéma de type σ tels que pour tout $\mathcal{M}$, modèle de la représentation, la satisfiabilité de tous ces problèmes est équivalente au jugement $\mathcal{M} \models \varnothing; \varnothing \vdash \varepsilon_0 \colon \sigma$. Notons que cette opération est peut être exponentielle par rapport à la profondeur d'imbrication des **`let`**.

On montre ensuite l'équivalence sémantique attendue : si dans un certain modèle $\mathcal{M}$, tous les problèmes P_I sont satisfaits, c'est-à-dire que $\mathcal{M} \models P_I$ pour tout I, alors on a $\mathcal{M} \models \varnothing; \varnothing \vdash \varepsilon_0 \colon \sigma$, comme expliqué ci-dessus. On peut donc appliquer le lemme de correction 6.3 et déduire que $\varnothing; \varnothing \vdash \varepsilon_0 : [\![\sigma]\!]$. Puis, par le lemme 6.7 on trouve que $\varnothing; \varnothing \vdash \varepsilon : [\![\sigma]\!]$. Inversement, si dans le modèle $\mathcal{M}$, il existe T tel que $\varnothing; \varnothing \vdash \varepsilon : \sigma$, alors par ce même lemme 6.7, on sait que $\varnothing; \varnothing \vdash \varepsilon_0 : \sigma$. L'expression ε_0 étant normale, on peut appliquer le lemme de complétude 6.6 pour déduire que $\varnothing; \varnothing \vdash \varepsilon_0 \colon \sigma$ et $[\![\sigma]\!] \leqslant^{\sharp} T$. Dans l'application de ce lemme, on prend soin de procéder aux mêmes choix de variables que lors de la phase de génération des problèmes, ce qui assure que le schéma de types produit par le lemme est bien σ et que tous les problèmes P_I sont bien satisfaits dans $\mathcal{M}$. ■

Chapitre 7

Discussion

Dans les chapitres précédents, nous avons développé une formalisation du typage polymorphe d'un langage avec multi-méthodes. Successivement, nous avons défini le langage (chapitre 1), sa sémantique (chapitre 2), un système de types purement algébrique (chapitre 3) et la preuve de correction associée (chapitre 4), un langage d'annotation de types et la spécification du typage des expressions annotées (chapitre 5) et enfin, nous avons montré que le typage des expressions annotées est réductible à des problèmes de contraintes (chapitre 6).

Pour utiliser cette formalisation dans un vrai langage de programmation, de nombreux aspects du typage devraient encore être traités. Il faudrait fixer une algèbre particulière, étudier la résolution des problèmes de contraintes, en fournir des implémentations efficaces, s'occuper de la simplification des types et de l'affichage compréhensible des erreurs de typage, etc. Cela dépasserait largement le cadre de cette thèse et des objectifs plus modestes que nous lui avons fixés.

Dans ce chapitre, nous allons discuter d'une manière plus informelle quelques uns des problèmes posés par la mise en œuvre pratique du typage. Nous montrerons également comment la présentation de certains aspects essentiels du typage d'un langage à objets profite avantageusement de l'approche algébrique adoptée dans cette thèse. Nous nous intéresserons particulièrement au typage en « monde ouvert », c'est-à-dire quand un module est typée indépendamment de l'algèbre de types dans laquelle l'exécution finit par avoir lieu.

Ce chapitre sera organisé comme suit : à la section 7.1, nous montrerons comment la formalisation peut être mise en œuvre concrètement dans un langage de programmation spécifique. En section 7.2, nous aborderons le problème du monde ouvert. En section 7.3, nous dirons quelques mots de l'inférence de types et des difficultés supplémentaires qu'elle pose par rapport à la simple vérification de types envisagée jusqu'à présent.

Ce chapitre se terminera par la comparaison de cette thèse avec des travaux apparentés (section 7.4).

7.1 Mise en œuvre de la formalisation

Dans chacun des chapitres précédents, nous avons introduit des structures laissées ouvertes et qui sont autant de paramètres de la formalisation. Le tableau suivant récapitule ces différents paramètres :

Aspect ou système	Paramètres
Syntaxe des expressions	– Symboles des constantes, opérations et motifs primitifs – Symboles des classes et ensemble des champs d'une classe
Sémantique opérationnelle	– Valeurs des opérations primitives et motifs primitifs – Relation de sous-classement
Système de types algébrique	Algèbre de types
Syntaxe des expressions annotées	Langage de types
Typage des expressions annotées (système semi-algébrique)	Modèle du langage de types dans l'algèbre de types
Réduction du typage à des problèmes de contraintes	Représentation syntaxique des éléments de base du typage

L'intérêt de ces multiples paramétrages est de mettre en évidence, à chaque étape de la formalisation, ce qui est strictement nécessaire. En pratique, dans un vrai langage de programmation, la définition du langage lui-même ne fixe typiquement que les paramètres primitifs (types de base, opérations primitives, etc.). Le reste des paramètres est dérivé des déclarations du programmeur. La puissance d'expression de ces déclarations ainsi que la façon dont on en dérive les paramètres de typage dépendent entièrement du langage de programmation considéré et relèvent uniquement de la conception de langage. Nous allons maintenant en donner un exemple simple qui montre déjà la grande variété de conceptions possibles et la façon dont notre approche peut les modéliser. En reprenant la syntaxe illustrative déjà utilisée dans les chapitres précédents, voici quelques exemples de déclarations de classes :

```
abstract class Couleur {
  r, g, b: [int];
}
abstract class ObjetColoré {
  c: Couleur;
}
class Point {
  x, y: int;
}
class PointColoré ⊑ Point, ObjetColoré {}
```

```
class List⟨α⊕⟩ {}
class Cons⟨α⊕⟩ ⊑ List {
  head: α;
  tail: List⟨α⟩;
}
class Nil⟨α⊕⟩ ⊑ List {}
class Prédicat⟨α⊖⟩ {
  test: α → bool;
}
```

L'ensemble des informations contenues dans les déclarations de classes d'un programme sera collectivement appelé HIÉRARCHIE DE CLASSES.

À partir d'une hiérarchie $\mathcal{H}$, on peut dériver de façon évidente une structure de classes $\mathcal{C}$, c'est-à-dire un ensemble de symboles de classes, une relation de sous-classement et un ensemble de champs

pour chaque classe. On peut également dériver naturellement un langage de types $\mathcal{L}$: chaque symbole de classe C joue le rôle d'un constructeur de type d'arité égale au nombre de variables de types qui paramètrent la classe. Par exemple List est un constructeur d'arité 1 car cette classe accepte une variable de type α.

Il y a beaucoup plus de choix pour dériver l'algèbre de types utilisée $\mathcal{A}$. Comme mentionné au chapitre 3, on peut choisir de travailler avec des termes finis ordonné structurellement, avec des termes infinis, ou bien rationnels. On peut saturer l'ordre partiel obtenu pour en faire un treillis, ou un demi-treillis, ou bien garder l'ordre partiel arbitraire, etc.

Après avoir choisir l'algèbre de types à utiliser, il convient de vérifier qu'elle permet bien de construire un modèle de $\mathcal{L}$ et des représentations syntaxiques des éléments de base. En pratique ce n'est pas un problème car toutes ces algèbres sont des algèbres de termes.

Le choix de l'algèbre de types est donc l'élément crucial. Il est important de comprendre que chacune de ces algèbres aboutit à un système de types *a priori différent*, c'est-à-dire qu'il va rejeter plus ou moins de programmes. Par exemple, dans l'algèbre des termes finis structurels, la contrainte $\exists\alpha \,.\, \alpha \leqslant \alpha \to \alpha$ n'est pas satisfiable. En revanche, elle est satisfiable dans l'algèbre structurelle avec termes infinis ou récursifs. Elle est aussi satisfiable si l'algèbre est un treillis (prendre $\alpha = \bot$). Par ailleurs, les systèmes de type issus d'algèbres différentes auront également une complexité *a priori* différente. Par exemple, considérons simplement le problème de satisfiabilité d'une contrainte $\exists\alpha \,.\, \kappa$. Dans une algèbre treillis, ce problème est typiquement traitable en temps polynomial $O(n^3)$ [Pal95]. En revanche, dans l'algèbre des termes finis ordonnés structurellement, le problème est **PSPACE**-complet [Tiu92, Fre97].

Au final, on se rend compte que l'un des avantages de notre approche algébrique est qu'elle permet de bien séparer trois aspects du systèmes de type : la correction, l'implémentation et la conception du langage. En particulier, il est possible de faire varier l'algèbre de types et de prouver très facilement la correction du système qui en résulte. Cela permet de se concentrer sur la conception du langage (quelles déclarations offrir ? quelles programmes rejeter ?) et sur l'implémentation du typage (quelles algorithmes pour résoudre les contraintes ?).

7.2 Monde ouvert

Jusqu'à présent, nous avons fait comme si le langage était un tout fermé : une fois fixé un ensemble de déclarations de classes, on peut écrire, typer et exécuter une expression. En réalité, un langage à objets réaliste est ouvert, c'est-à-dire qu'il permet d'écrire des modules réutilisables. En pratique, cela signifie qu'un module peut être typé et compilé dans une certaine hiérarchie de classes, mais qu'il est finalement exécuté dans une autre hiérarchie.

Prenons l'exemple d'un module M spécialisé dans le traitement des points. Il est typiquement compilé dans une hiérarchie $\mathcal{H}_1$ définissant les classes Point et Point3D. Cependant, au moment où le module est exécuté, le programmeur a pu ajouter les classes ObjetColoré, PointColoré et Point3DColoré pour former la hiérarchie $\mathcal{H}_2$ qui est de ce fait une EXTENSION de $\mathcal{H}_1$.

Notons que les expressions dans le module M, les définitions de méthodes par exemple, sont syntaxiquement bien formées dans la structure de classes dérivée de la hiérarchie $\mathcal{H}_2$ car celle-ci se contente de rajouter des classes à $\mathcal{H}_1$ et ne modifie pas les champs des classes existantes. Si ces expressions sont annotées, remarquons de même que les annotations restent syntaxiquement bien

formées quand on passe de $\mathcal{H}_1$ à $\mathcal{H}_2$ car on a simplement de nouveaux constructeurs de type et on ne touche pas à l'arité des constructeurs existants.

Du point de vue de l'exécution, on remarque que la sémantique opérationnelle de ces expressions est plus générale dans $\mathcal{H}_2$ que dans $\mathcal{H}_1$ car elles peuvent traiter des objets construits avec des classes inconnues de $\mathcal{H}_1$, ce qui est bien le but recherché du « polymorphisme à objets ».

Qu'en est-il du typage ? Si M est bien typé dans $\mathcal{H}_1$, il est en général faux qu'il soit également bien typé dans $\mathcal{H}_2$. Par exemple, plaçons-nous dans l'algèbre des termes finis structurels et supposons que le typage de M selon l'algorithme du chapitre 6 produise entre autres le problème d'implication suivant :

$$\forall \alpha \,.\, \alpha \leqslant \mathsf{Point3D} \supset \mathsf{Point3D} \leqslant \alpha$$

Dans le modèle $\mathcal{M}_1$ dérivé de $\mathcal{H}_1$, cette implication de contrainte est vraie, parce que Point3D n'a pas de sous-type strict et donc la seule solution de la contrainte de gauche est $\alpha \mapsto$ Point3D. En revanche, dans le modèle $\mathcal{M}_2$ dérivé de $\mathcal{H}_2$, l'implication n'est plus vraie car α peut prendre la valeur Point3DColoré qui est bien un sous-type de Point3D mais n'en est pas un super-type.

Il semble donc qu'il ne soit pas possible de typer le module M sans savoir dans quelle hiérarchie il va être exécuté. Cela va à l'encontre d'un principe de la programmation modulaire et de la compilation séparée qui voudrait qu'on puisse toujours typer un module indépendamment de son utilisation.

Pour traiter ce « typage ouvert » ou typage en « monde ouvert », remarquons d'abord que notre approche algébrique fournit un moyen très simple pour exprimer le résultat attendu. Pour typer un module M écrit dans une hiérarchie $\mathcal{H}_1$, on commence par extraire une liste de problèmes de typage $P_1, \ldots, P_N$ selon l'algorithme du chapitre 6. Il suffit alors de résoudre les problèmes P_i *simultanément dans toutes les extensions de* $\mathcal{H}_1$.

Précisons maintenant ce qu'on entend exactement par extension. Pour que $\mathcal{H}_2$ étende $\mathcal{H}_1$, un certain nombre de conditions sont nécessaires : $\mathcal{H}_2$ peut ajouter des classes, mais ne doit pas modifier le caractère concret ou abstrait d'une classe existant. Elle ne doit pas non plus modifier les relations de sous-classement existantes. Elle ne peut ajouter des champs à des classes existantes, ni modifier leur type.

En pratique, ces conditions sont nécessaires mais elles sont souvent trop générales pour modéliser les extensions qui sont effectivement *exprimables* dans un langage de programmation donné. Les extensions exprimables peuvent être limitées par la forme syntaxique des déclarations de classes. Par exemple, dans la syntaxe suggérée ci-dessus, on peut rajouter des classes uniquement *par le bas*. C'est-à-dire qu'une nouvelle classe ne peut être rajoutée comme super-classe d'une classe existante. Dans d'autres syntaxes et donc dans d'autres langages, l'extension par le haut peut être autorisée. Par exemple, si on introduit la classe abstraite Copiable des objets copiables, c'est-à-dire qui peuvent être passés à une méthode *copy*, alors on peut vouloir autoriser *après coup* la classe Point à être une sous-classe de Copiable.[1]

De la même façon que le choix de l'algèbre de types conditionne le système de typage fermé obtenu, on peut remarquer que le choix de la notion d'extension de hiérarchie détermine également la puissance du système ouvert qu'on peut en dériver. Par exemple, considérons le problème d'implica-

[1] Un langage qui autoriserait cette possibilité devrait probablement prévoir de la restreindre aux classes comme Copiable qui n'ont pas de champs.

tion suivant :

$$\forall\alpha\ .\ \mathsf{Point} \leqslant \alpha \supset \alpha \leqslant \mathsf{Point}$$

Ce problème est faux quand la relation d'extension de hiérarchie n'est pas limitée. En effet, il suffit de considérer une hiérarchie $\mathcal{H}_2$ qui ajoute une super-classe à Point, par exemple la classe Objet. Dans $\mathcal{H}_2$, il est bien clair que Point n'est plus une classe maximale de sorte que que l'implication n'est pas vrai dans toutes les hiérarchies qui étendent $\mathcal{H}_1$. Le système de typage ouvert va donc rejeter un module qui génèrerait cette implication. Cependant, si le langage de programmation restreint la forme des déclarations de classe et n'autorise que les extensions par le bas, alors $\mathcal{H}_2$ n'est pas une extension de $\mathcal{H}_1$. De plus, il est clair qu'une extension par le bas va toujours préserver la maximalité de Point, de sorte que, dans un tel langage, l'implication est en fait vraie dans toutes les extensions. Ainsi, il convient d'accepter un programme qui génère cette implication.

Au final, l'approche algébrique fournit donc un outil pour la formalisation fine du monde ouvert mais n'impose rien quant au choix précis de la relation d'extension, de la même façon que l'algèbre de types n'est pas imposée. Encore une fois, ces choix relèvent de la conception du langage de programmation et de l'équilibre entre la puissance du système de typage et son caractère pratique.

7.3 Inférence de types

Dans cette thèse, nous avons seulement abordé le typage du point de vue de la vérification de types (« *type-checking* »), c'est-à-dire quand le programmeur doit indiquer au compilateur le type des variables et des paramètres des fonctions.

Dans les langages fonctionnels comme ML, certaines de ces déclarations peuvent être omises par le programmeur et le système se charger d'*inférer* le type des variables (« *type inference* »). Notons toutefois que certaines déclarations de type doivent toujours être entrées par le programmeur comme par exemple le type des éléments publics d'un module (sa « signature ») ou bien le contenu des alternatives d'un type concret.

Dans un langage avec sous-typage et ordre supérieur, les types des variables peuvent devenir particulièrement compliqués à exprimer. Il semble donc indispensable de fournir une forme d'inférence, au moins pour les variables locales. Voyons comment l'inférence pourrait être intégrée à notre formalisation et les problèmes que cela pose.

Conformément à notre démarche, nous allons séparer la spécification de l'inférence de types de son implémentation avec des contraintes. Nous proposons d'ajouter la production suivante dans la grammaire des expressions annotées :

$$\begin{array}{ll} \varepsilon ::= \ \ldots & \\ \quad \mid \exists\mathcal{V} \mid \kappa\ .\ \varepsilon & \text{(Introduction de variables inférées)} \end{array}$$

Dans cette expression, $\mathcal{V}$ représente un ensemble de variables dont le système de types doit trouver une valeur compatible avec la contrainte κ et avec l'expression ε. La règle de typage associée est simplement :

$$\frac{\mathrm{dom}(\rho) = \mathcal{V} \qquad {}_{\hat{\rho}\oplus\rho}[\![\kappa]\!] \qquad \hat{\rho} \oplus \rho;\ G \vdash \varepsilon : T}{\hat{\rho};\ G \vdash (\exists\mathcal{V} \mid \kappa\ .\ \varepsilon) : T}$$

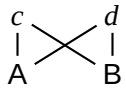
Par exemple, la fonction $\exists\alpha \mid \mathsf{vrai}\ .\ (\mathbf{fun}\ x : \alpha\ \Rightarrow\ x + 1)$ est bien typée car on peut prendre $\alpha = \mathsf{float}$ ou $\alpha = \mathsf{int}$. Par contraste, notons que $\forall\alpha \mid \mathsf{vrai}\ .\ (\mathbf{fun}\ x : \alpha\ \Rightarrow\ x + 1)$ n'est pas bien typée car il n'est pas vrai que $\mathbf{fun}\ x : \alpha\ \Rightarrow\ x + 1$ soit bien typée pour tout α (contrexemple : $\alpha = \mathsf{Point}$). Le programmeur donc est forcé d'expliciter les valeurs possibles de α par exemple en écrivant $\forall\alpha \mid \alpha \leqslant \mathsf{float}\ .\ (\mathbf{fun}\ x : \alpha\ \Rightarrow x + 1)$.

La spécification de l'inférence ne pose donc pas de problème particulier et s'intègre bien dans notre formalisation. En revanche, la réduction de l'inférence aux contraintes est problématique. La difficulté provient de la présence d'une alternance entre des quantificateurs universels (annotations polymorphes sur les définitions récursives et sur les méthodes) et existentiels (inférence de type des paramètres de fonction). Dans le cas général, nous allons maintenant montrer qu'il est nécessaire d'avoir un langage de contraintes plus riche, autorisant des quantificateurs universels. Les problèmes de contraintes obtenus sont alors plus complexes (formules $(\forall\exists)^*$ plutôt que $\forall\exists$).

Pour faire comprendre la nécessité de contraintes riches, examinons un exemple d'inférence où le type principal d'une expression ne peut pas être exprimé avec les contraintes simples. Plaçons-nous par exemple dans une structure contenant les classes A et B et considérons l'expression suivante :

$$\varepsilon = \exists\alpha\ .\ \mathsf{vrai}(\mathbf{fun}\ x : \alpha\ \Rightarrow\ (\mathbf{fix}\ y : \sigma\ \Rightarrow x)) \text{ où } \sigma = \forall\beta \mid \mathsf{A} \leqslant \beta \wedge \mathsf{B} \leqslant \beta\ .\ \beta$$

La définition récursive a ici pour effet d'imposer une contrainte polymorphe sur la variable à inférer α. Dans le cas général, ceci n'est pas exprimable par une contrainte simple. Considérons en effet une algèbre structurelle atomique construite sur un ordre partiel de base où A et B possède deux super-types communs incomparables c et d :



Notons bien que c et d sont des monotypes algébriques qui ne sont pas directement représentables par un monotype syntaxique. Cette situation peut typiquement survenir quand on se place dans un monde ouvert et si c et d sont des classes définies dans une extension de la hiérarchie courante.

Dans cette algèbre, remarquons d'abord que l'expression ε est bien typée. Le polytype σ dénote en effet l'ensemble des super-types communs à A et B, c'est-à-dire l'ensemble $\{c, d\}$. Les monotypes qui sont des sous-types de σ sont donc uniquement A et B. On vérifie alors que $[\alpha \mapsto \mathsf{A}]$ et $[\alpha \mapsto \mathsf{B}]$ sont bien les solutions possibles pour le typage de ε et les types de ε en résultant sont donc l'ensemble des $t \to u$ où t vaut A ou B et u vaut c ou d. Nous allons maintenant montrer qu'il n'existe pas de schéma de types décrivant exactement cet ensemble.

Démonstration. Supposons qu'il existe un schéma tel que :

$$[\![\forall\mathcal{V} \mid \kappa\ .\ \tau]\!] = \{t \to u \mid t \in \{\mathsf{A}, \mathsf{B}\}, u \in \{c, d\}\}$$

Considérons alors le contexte $\Delta = \{\mathcal{V}, \alpha, \beta \mid \kappa \wedge \tau = \alpha \to \beta\}$. On constate que l'ensemble des monotypes $\rho(\alpha)$ quand ρ parcourt l'interprétation $[\![\Delta]\!]$ est exactement égal à $\{\mathsf{A}, \mathsf{B}\}$. Montrons maintenant que cela est impossible quel que soit le contexte Δ.

Considérons donc un contexte quelconque $\Delta = \{\alpha_1, \ldots, \alpha_n \mid \kappa\}$. Sans perte de généralité, nous supposons que κ ne contient pas de quantificateur existentiel, ce qui est toujours possible en faissant passer les variables en tête. Enfin, nous supposons que κ ne contient que des contraintes entre termes atomiques, c'est-à-dire entre une variable α_i et les deux constantes A et B. Comme nous travaillons

dans une algèbre structurelle atomique, cela est toujours possible en décomposant les contraintes entre termes.

Considérons maintenant l'opérateur binaire $\vee$ défini sur l'algèbre comme suit :

$$\begin{aligned} \mathsf{A} \vee \mathsf{A} &= \mathsf{A} \\ \mathsf{B} \vee \mathsf{B} &= \mathsf{B} \\ t \vee u &= c \quad \text{dans tous les autres cas} \end{aligned}$$

Notons que l'opérateur $\vee$ satisfait à la propriété suivante :

$$t \leqslant u \wedge t' \leqslant u' \Rightarrow t \vee t' \leqslant u \vee u'$$

Par induction sur la structure de κ, on en déduit que pour toute paire de solutions $\rho, \rho' \in [\![\Delta]\!]$, on a aussi $\rho \vee \rho' \in [\![\Delta]\!]$.

S'il existe une solution de ρ telle que $\rho(\alpha_i) = \mathsf{A}$ et une autre telle que $\rho'(\alpha_i) = \mathsf{B}$, alors $\rho \vee \rho'$ est aussi une solution dans $[\![\Delta]\!]$. Mais, comme $\rho \vee \rho'(\alpha_i) = \mathsf{A} \vee \mathsf{B} = c$. il est donc impossible que $\rho(\alpha_i)$ parcoure exactement l'ensemble $\{\mathsf{A}, \mathsf{B}\}$. ■

Les contraintes du chapitre 5 sont donc trop faibles pour traiter l'inférence de type et exprimer le type principal d'une expression. Pour traiter l'inférence dans toute sa généralité, il faudrait essentiellement pouvoir exprimer des inégalités entre schémas dans les contraintes. Le type principal inféré pour ε serait alors de la forme :

$$\forall \alpha, \beta \mid \{\alpha \leqslant (\forall \beta \mid \mathsf{A} \leqslant \beta \wedge \mathsf{B} \leqslant \beta \,.\, \beta) \wedge \mathsf{A} \leqslant \beta \wedge \mathsf{B} \leqslant \beta\} \,.\, \alpha \to \beta$$

Nous pensons qu'en utilisant ce genre de contraintes étendues, il doit être possible de transposer assez facilement la preuve du chapitre 6 pour réduire l'inférence de types, et non plus seulement la vérification, aux contraintes.

Toutefois, la résolution des contraintes étendues est *a priori* plus difficile que celle des contraintes simples. Notons cependant que les contraintes les plus générales ne sont nécessaires que si on autorise une définition récursive polymorphe ou une méthode à figurer sous un **`fun`**. Pour simplifier le problème, on peut donc également envisager de n'autoriser ces expressions polymorphes qu'au niveau le plus haut du programme. On est alors ramené à des problèmes d'implications de contraintes simples. Enfin, si on interdit complètement les définitions récursives polymorphes et les méthodes, on peut alors vérifier que les problèmes qu'on obtient sont uniquement des problèmes de satisfiabilité. Cette situation correspond au langage ML de base (core-ML) avec sous-typage et on retrouve alors le résultat bien connu que le typage de ce langage est simplement équivalent à la satisfiabilité d'une contrainte de sous-typage.

Pour terminer, le tableau suivant résume la complexité logique des problèmes de typage obtenus pour les différents sous-langages envisagés :

Langage	Typage	Problèmes à résoudre	Complexité logique
core-ML	inférence	satisfiabilité	$\exists$
méthodes et **`fix`** à *toplevel*	inférence	implications	$\forall\exists$
méthodes et **`fix`** profonds	vérification	implications	$\forall\exists$
méthodes et **`fix`** profonds	inférence	formules quelconques	$(\forall\exists)^*$

7.4 Travaux apparentés

Ce travail de thèse constitue une simplification et une généralisation considérables de $ML_{\leqslant}$ [BM96, BM97]. La simplification porte d'abord sur la sémantique opérationnelle qui est typée et très non-standard dans [BM96], alors qu'elle est non-typée et par réduction chez nous. Nous introduisons également le filtrage profond à la ML et non pas seulement le dispatch superficiel. Notre présentation du typage est systématique, modulaire et conçue pour être extensible alors que celle de [BM96] s'est révélée figée et ses preuves difficiles à lire et à étendre. L'implication de contrainte dans [BM96] est basée sur une desription de nature algorithmique (axiomatisation complète), alors que notre approche algébrique permet de bien séparer la correction du typage de la résolution des contraintes. En revanche, notre travail n'aborde pas du tout la question de la résolution des problèmes de typage pour lesquels [BM96] fournit des algorithmes.

Bonniot [Bon02a] a également utilisé une approche algébrique et modulaire pour typer $ML_{\leqslant}$, ce qui lui permet de séparer la preuve de correction et son instantiation sur une algèbre concrète. Le langage utilisé est moins riche : le filtrage profond, les méthodes imbriquées ou les éléments primitifs ne sont pas traités. Par ailleurs, la notion d'algèbre dans [Bon02a] est très proche du langage source. Une algèbre doit notamment définir une opération de substitution qui est une abstraction de la substitution opérationnelle et qui fait intervenir les variables du programme. La preuve de correction du typage s'en trouve simplifiée mais, en plaçant le niveau d'abstraction aussi haut, une grande partie du typage doit être traité au coup par coup dans les algèbres spécifiques. Notre formalisation est un peu moins générale, mais a l'avantage de traiter de façon générique une plus grande partie du typage, les contraintes par exemple. La forme *#C* des motifs a été introduite dans [Bon02a].

Dans un autre article, Bonniot [Bon02b] a également décrit une extension de $ML_{\leqslant}$ avec des « *kinds* » qui permettent d'écrire des contraintes particulièrement expressives pour traiter des méthodes partiellement polymorphes. Nous pensons que cette extension devrait s'intégrer parfaitement dans le cadre de cette thèse. En particulier, les kinds pourraient être codées en utilisant les prédicats du langage de types. De manière générale, toutes les extensions du typage de ML basées sur des types contraints avec des prédicats (par exemple les *type classes* [WB89]) devraient pouvoir utiliser notre présentation très facilement.

Le système HM(X) introduit par Sulzmann *et al.* [SOW96, Sul00] est actuellement le « standard » en matière de typage modulaire. La présntation du typage y est paramétrée par un système de contraintes X, qui correspond *grosso modo* dans notre formalisation à la donnée d'un langage de type et d'un modèle. Moyennant quelques hypothèses sur X, la preuve de correction de typage peut être rendue générique. Notre travail se distingue de HM(X) par plusieurs aspects. D'abord, nous traitons un langage considérablement plus riche, avec des multi-méthodes, des définitions récursives polymorphes ainsi que des éléments primitifs. Ensuite, la présentation de HM(X) reste très syntaxique : les règles de typage correspondent essentiellement à l'algorithme utilisé au chapitre 6. Notre présentation a l'avantage de permettre l'écriture d'une spécification du typage qui n'utilise pas les contraintes. Les règles de typage algébriques (chapitre 3) ou semi-algébrique dans le cas où les expressions sont annotées (chapitre 5) sont alors bien plus simples à lire et la preuve de correction s'en trouve facilitée. Les contraintes ne sont introduites que lors de l'implémentation de la spécification et le système qui en résulte (chapitre 6) n'apparaît que comme un intermédiaire de preuve. Les règles de ce système ne sont pas choisies arbitrairement mais construites pour que le système syntaxique corresponde exactement à la spécification.

Le passage par un système purement algébrique pour prouver la correction d'un système à contraintes

a été esquissé par Pottier [Pot01] dans un travail qui s'inspire d'une version préliminaire de cette thèse. Cependant, ce passage n'est utilisé que comme une technique facilitant la preuve de correction du système syntaxique HM(X) en le réécrivant dans un système purement algébrique. Dans notre thèse, cette technique est considérablement étendue de sorte que le système algébrique (plus précisément semi-algébrique) forme une spécification du typage et que le système syntaxique en constitue une implémentation correcte et complète et pas seulement correcte. Ainsi, l'approche algébrique peut permettre de guider la construction du système syntaxique.

Pottier et Simonet [SP03] ont étudié le filtrage profond dans un langage à la ML avec sous-typage et ont remarqué que les contraintes simples sont insuffisantes. Leurs contraintes « gardées » sont similaires aux contraintes étendues que nous avons décrites à la section 7.3.

Castagna, Ghelli et Longo [CGL92, Cas97] ont étudié une extension du λ-calcul simplement typé avec des multi-méthodes où les méthodes sont des fonctions de première classe composables avec un opérateur &. Ce travail a inauguré une approche nouvelle des objets qui se démarquait de la vision « objets=enregistrements extensibles ». Plus récemment, le langage CDuce [BCF03] s'est inspiré de ce λ&-calcul pour autoriser les fonctions surchargées. Le typage de celles-ci demandent au programmeur de complètement spécifier le type de la fonction sous la forme d'un ensemble explicite de types fonctionnels syntaxiques, c'est-à-dire en extension. Un avantage de $\mathrm{ML}_{\leqslant}$, et donc aussi du système que nous présentons dans cette thèse, est qu'on peut donner le type d'une fonction surchargée en compréhension, c'est-à-dire sous la forme d'une schéma de types contraint, ce qui permet de passer plus commodément à un monde ouvert.

Le langage Cecil [Ct02, CL95] est un langage de recherche avec multi-méthodes suffisamment stable pour permettre d'écrire son propre compilateur. Cecil fournit un système de vérification de type monomorphe décidable [CL95] et un autre système polymorphe avec inférence [Lit98] basé sur le système F et dont la décidabilité est inconnue. Dans l'ensemble, ces systèmes de type s'éloignent du typage à la Hindley-Milner auquel nous nous référons dans cette thèse.

Finalement, on ne peut pas ne pas mentionner le langage OCAML [LDG$^+$03], seule extension de ML avec objets actuellement réellement déployée. L'inférence des types y est très efficace car elle est basée sur de l'unification et les variables de rangée. Certaines coercions de sous-typage doivent alors être exprimées explicitement par le programmeur. En pratique, ce compromis donne de très bons résultats. Cependant, OCAML souffre des inconvénients du modèle à enregistrements (méthodes binaires notamment). Par ailleurs, le langage OCAML ne fournit pas de sous-typage primitif, comme $\mathsf{int} \leqslant \mathsf{float}$, ce qui oblige d'avoir deux jeux d'opérateurs mathématiques et limite le polymorphisme des fonctions arithmétiques. Il serait intéressant d'étudier l'intégration d'un système avec sous-typage utilisant les techniques de cette thèse avec le système à objets de OCAML.

Conclusions

Ce travail est issu de l'implémentation dans le langage Jazz du système de types $ML_{\leqslant}$. Celui-ci présentait des limitations gênantes : absence d'inférence, langage de contraintes rigide, présentation non-standard et peu extensible. Lors de la phase de conception de Jazz et plus encore lorsque le langage commença à être utilisé, de nombreuses idées d'extension surgirent, mais il manquait clairement un outil théorique permettant de les valider rapidement.

Dans cette thèse, nous avons présenté une formalisation générique du typage. À la manière de HM(X), le système de types y est paramétré par une algèbre de types, et, moyennant des hypothèses minimales sur cette algèbre, la correction du typage peut être prouvée une fois pour toutes. Cette généralité n'est pas gratuite : notre expérience de Jazz nous a convaincu qu'une telle approche générique est indispensable pour un langage en développement, particulièrement quand le typage est riche (polymorphisme, sous-typage) et le langage complexe (multi-méthodes, objets).

Un aspect original de la formalisation est le traitement extensionnel du polymorphisme : dans la *spécification* du typage, un type polymorphe est simplement un ensemble quelconque de types monomorphes. Les schémas de type et les contraintes ne sont introduits que lors de l'*implémentation*, celle-ci consistant en une réduction du typage à des problèmes de contraintes. La correction globale du typage découle alors de deux preuves indépendantes : 1) correction de la spécification par rapport à la sémantique opérationnelle et 2) correction de l'implémentation par rapport à la spécification.

Cette séparation est d'autant plus utile que l'implémentation est également prouvée *complète* par rapport à la spécification. On peut donc considérer que seule la spécification définit vraiment le système de types. Contrairement aux présentations habituelles des systèmes de type avec contraintes, les règles syntaxiques ne constituent pas la définition du système, mais sont un simple intermédiaire dans son implémentation. Or ces règles sont bien plus compliquées que la spécification car elles travaillent exclusivement au niveau de la syntaxe sur les contraintes, variables de types et schémas et doivent donc prendre en compte de nombreux détails syntaxiques. Notre approche contribue donc à simplifier considérablement la preuve de correction du typage et assure que les règles syntaxiques ne sont pas par mégarde arbitrairement restrictives, ce qui représente une « sécurité théorique » appréciable pour un langage aussi riche et complexe.

Dans cette thèse, nous n'avons pas du tout abordé la résolution concrète des problèmes de contraintes obtenus. Celle-ci dépend en effet du domaine d'interprétation et de la forme des contraintes, deux paramètres spécifiques qui définissent le système de types. De plus, il existe aujourd'hui une vaste littérature sur la résolution des contraintes de typage.

Cependant, nous avons esquissé une modélisation originale du monde ouvert selon laquelle pour typer un programme dans un monde ouvert, il suffit d'extraire les problèmes de contraintes associés et de les résoudre *simultanément dans toutes les extensions futures du monde présent*. Au même titre

que l'algèbre de types, la notion d'extension est ici elle-même un paramètre du système de types qui dépend du langage de programmation concret considéré.

En soi, cette modélisation du monde ouvert ne permet naturellement pas de progresser dans la résolution du typage mais elle fournit une formulation simple et claire du problème. Alors que de très nombreux travaux ont été consacrés aux contraintes de typage, on s'aperçoit que, sauf exception, il s'agit toujours de la résolution de certaines formes de contraintes dans une structure d'interprétation fixée. Il serait donc intéressant de reconsidérer ce corpus de résultats quand on doit traiter un problème de contraintes simultanément dans tous les modèles d'interprétation issus d'une relation d'extension donnée.

Bibliographie

[AC93] Roberto M. AMADIO et Luca CARDELLI : Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, septembre 1993.

[Bar84] Hendrik Pieter BARENDREGT : *The Lambda Calculus – Its Syntax and Semantics*, volume 103 de *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[BCC+95] Kim B. BRUCE, Luca CARDELLI, Giuseppe CASTAGNA, Jonathan EIFRIG, Scott F. SMITH, Valery TRIFONOV, Gary T. LEAVENS et Benjamin C. PIERCE : On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BCF03] Véronique BENZAKEN, Giuseppe CASTAGNA et Alain FRISCH : Cduce : an xml-centric general-purpose language. Dans *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63. ACM Press, 2003.

[Ber78] G. BERRY : Stable models of typed lambda-calculi. *Automata, Languages and Programming, Fifth Colloquium, Udine*, pages 72–89, 1978. Lecture Notes in Computer Science 62.

[BM96] François BOURDONCLE et Stephan MERZ : On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Paris, mars 1996.

[BM97] François BOURDONCLE et Stephan MERZ : Type checking higher-order polymorphic multi-methods. Dans *Conference Record of the 24th Annual ACM Symposium on Principles of Programming Languages*, pages 302–315, Paris, janvier 1997. ACM.

[Bon02a] Daniel BONNIOT : Type-checking multi-methods in ML (a modular approach). Dans *The Ninth International Workshop on Foundations of Object-Oriented Languages, FOOL 9*, Portland, Oregon, USA, janvier 2002.

[Bon02b] Daniel BONNIOT : Using kinds to type partially polymorphic multi-methods. Dans *Workshop on Types in Programming (TIP'02)*, Dagstuhl, Germany, juillet 2002.

[BVB94] François BOURDONCLE, Jean VUILLEMIN et Gérard BERRY : The 2z reference manual, 1994.

[Cas97] Giuseppe CASTAGNA : *Object-Oriented Programming : A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

[CDT01] The Coq Development TEAM : *The Coq Proof Assistant Reference Manual Version 7.2*. INRIA-Rocquencourt, December 2001.
http ://coq.inria.fr/doc-eng.html.

[CGL92] Giuseppe CASTAGNA, Giorgio GHELLI et Giuseppe LONGO : A calculus for overloaded functions with subtyping. Dans *Proceedings of the ACM Conference on Lisp and Functional Programming*, volume 5, pages 182–192, 1992.

[CL95] Craig CHAMBERS et Gary T. LEAVENS : Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, novembre 1995.

[CP90] T. COQUAND et C. PAULIN : Inductively defined types. Dans *Proceedings of the international conference on Computer logic*, pages 50–66. Springer-Verlag New York, Inc., 1990.

[Ct02] Craig CHAMBERS et THE CECIL GROUP : The cecil language : Specification & rationale, version 3.1. http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-ceci 2002.

[FBB+] Alexandre FREY, Gérard BERRY, Patrice BERTIN, François BOURDONCLE et Jean VUILLEMIN : The jazz home page. http://www.exalead.com/jazz/.

[FM88] You-Chin FUH et Prateek MISHRA : Type inference with subtypes. Dans H. GANZINGER, éditeur : *Proceedings of the European Symposium on Programming*, volume 300 de *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[FM89] You-Chin FUH et Prateek MISHRA : Polymorphic subtype inference : Closing the theory-practice gap. Dans J. DÍAZ et F. OREJAS, éditeurs : *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, pages 167–183. LNCS 352. Springer, mars 1989.

[Fre97] Alexandre FREY : Satisfying subtype inequalities in polynomial space. Dans Pascal Van HENTENRYCK, éditeur : *Proceedings of the Forth International Symposium on Static Analysis (SAS'97)*, numéro 1302 dans Lecture Notes in Computer Science, pages 265–277, Paris, France, septembre 1997. Springer Verlag.

[GJSB00] James GOSLING, Bill JOY, Guy STEELE et Gilad BRACHA : *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[Hin69] J. R. HINDLEY : The principal type-scheme of an object in combinatory logic. *Trans. American Math. Soc*, 146:29–60, 1969.

[KR03] Viktor KUNCAK et Martin RINARD : On the theory of structural subtyping. Rapport technique 879, MIT Laboratory for Computer Science, janvier 2003.

[KTU93] A. J. KFOURY, J. TIURYN et P. URZYCZYN : Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, avril 1993.

[LDG+03] Xavier LEROY, Damien DOLIGEZ, Jacques GARRIGUE, Didier RÉMY et Jérôme VOUILLON : *The Objective Caml system, release 3.07*. INRIA, September 2003.

[Lit98] Vassily LITVINOV : Contraint-based polymorphism in cecil : towards a practical and static type system. Dans *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–411. ACM Press, 1998.

[Mar94] Luc MARANGET : Two techniques for compiling lazy pattern matching. Rapport technique 2385, INRIA, 1994.

[Mey92] Bertrand MEYER : *Eiffel : the language*. Prentice-Hall, Inc., 1992.

[Mil78] Robin MILNER : A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, décembre 1978.

[Mit84] J. MITCHELL : Coercion and type inference (summary). Dans *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185. ACM, ACM, janvier 1984.

[MTHM97] Robin MILNER, Mads TOFTE, Robert HARPER et David MACQUEEN : *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Pae93] Andreas PAEPCKE, éditeur. *Object-Oriented Programming : The CLOS Perspective*. MIT Press, 1993.

[Pal95] Jens PALSBERG : Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[Pot98] Francois POTTIER : *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. Thèse de doctorat, Université Paris VII, juillet 1998.

[Pot01] François POTTIER : A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, mars 2001.

[Rém94] Didier RÉMY : Programming objects with ML-ART : An extension to ML with abstract and record types. Dans *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer Verlag, avril 1994. ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/tacs94.ps.gz.

[Sha96] Andrew SHALIT : *The Dylan reference manual : the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., 1996.

[Sim03] Vincent SIMONET : Type inference with structural subtyping : A faithful formalization of an efficient constraint solver. Dans Atsushi OHORI, éditeur : *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 de *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, novembre 2003. Springer-Verlag. ©Springer-Verlag.

[SOW96] Martin SULZMANN, Martin ODERSKY et Martin WEHR : Type inference with constrained types. Technical Report iratr-1996-28, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1996.

[SP03] Vincent SIMONET et François POTTIER : Constraint-based type inference for guarded algebraic data types. Submitted for publication, juillet 2003.

[Ste90] Guy L. STEELE JR. : *Common Lisp : The Language (second edition)*. Digital Press, 1990.

[Str67] Christopher STRACHEY : Fundamental Concepts in Programming Languages. Dans *Lecture Notes for International Summer School in Computer Programming*. Copenhague, August 1967.

[Str91] B. STROUSTRUP : *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[Str00] Christopher STRACHEY : Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. First print of an unpublished manuscript written 1967.

[Sul00] Martin SULZMANN : *A General Framework for Hindley/Milner Type Systems with Constraints*. Thèse de doctorat, Yale University, Department of Computer Science, May 2000.

[Tiu92] Jerzy TIURYN : Subtype inequalities. Dans *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, Santa Cruz, California, 22–25 juin 1992. IEEE Computer Society Press.

[TS96] Valery TRIFONOV et Scott SMITH : Subtyping constrained types. Dans *Static Analysis Symposium (SAS)*, volume 1145 de *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, septembre 1996.

[Ver01] *IEEE 1364 Verilog Language Reference Manual (LRM)*. 2001.

[VHD93] *IEEE 1164 Standard VHDL Langage Reference Manual*. 1993.

[Vui94] Jean E. VUILLEMIN : On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, Aug 1994.

[WB89] P. WADLER et S. BLOTT : How to make ad-hoc polymorphism less ad-hoc. Dans *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, janvier 1989.

[WM02] Lyndon WHILE et Greg MILDENHALL : An implementation of parallel pattern-matching via concurrent haskell. Dans Michael J. OUDSHOORN, éditeur : *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.