From types to logical assertions : automatic or assisted proofs of property about functional programs

Yann Régis-Gianas

T H È S E

présentée à

l'Université Paris 7 – Denis Diderot

pour obtenir le titre de

Docteur en Informatique

Des types aux assertions logiques : Preuve automatique ou assistée de propriétés sur les programmes fonctionnels

soutenue par

Yann Régis-Gianas

le 29 novembre 2007, devant le jury composé de :

Monsieur	Roberto Di Cosmo	Président
Monsieur	François Pottier	Directeur de thèse
Messieurs	Gilles Barthe	
	Claude Marché	Rapporteurs
Monsieur	Simon Peyton-Jones	
Monsieur	Jean-Pierre Talpin	Examinateurs

Table des matières

Remerciements

Une thèse ? Quel programme ! L'analogie est frappante quand on prête attention à ceux qui y contribuent. Au commencement, les parents forment votre syntaxe en vous mettant tout petit un clavier sous les doigts quitte à casser leur tirelire. Ensuite, ils vous laissent entre les mains de professeurs qui donnent un sens à vos bidouilles et vous font rêver en vous montrant que la vérité se trouve là où les parallèles se croisent (Mme Tolsau et M. Hochart se reconnaîtront). Puis, un jour, un sage (Jacques Sakarovitch) vous lance avec confiance dans un environnement dangereux, hétérogène et complexe (UNIX) avec pour seul outil une hache très lourde (C++) en vous demandant de créer des canards artificiels (les automates de Vaucanson). L'initiation à la recherche débute alors sous l'égide de Shamans (Akim Demaille, Thierry Géraud et Didier Verna) qui manient la hache avec beaucoup de classes.

Après de nombreux patchs, on finit par remettre tout à plat et revenir aux questions fondamentales concernant sa syntaxe et sa sémantique. Rempli de doutes, on se retrouve en sûreté sur le dos d'un chameau rassurant conçu par les Touaregs de Rocquencourt à l'hospitalité et la bonne humeur rafraîchissantes (Sandrine Blazy, Damien Doligez, Alain Frisch, Gérard Huet, Xavier Leroy, Michel Mauny, Basile Starynkevitch et Didier Rémy). On y rencontre d'autres programmes en quête de sens avec qui on partage son bivouac (Boris Yakobowski) ou le café politico-philosophico-geeko-comique (Arthur Chargéraud, Zaynah Dargaye, Nadji Gauthier, Benoît Montagu, Keiko Nakata, Nicolas Pouillard, Tahina Ramananandro, Benoît Razet, Vincent Simonet et Jean-Baptiste Tristan).

Enfin, arrive le jour où on pense son programme fin prêt pour l'expertise de François Pottier. Avec lui, le verbe est clair et l'intention est juste, ses messages d'erreur humains et sa vérification des types sans faille. J'ai peur de ne pas être à la hauteur des perles que sont ses programmes habituels mais une chose est certaine, j'ai gagné grâce à lui d'innombrables numéros de version et je suis fier qu'il n'ait jamais hésité à me confronter à de nombreux utilisateurs (Simon Peyton-Jones lors de notre semaine à Cambridge chez Microsoft Research, les organisateurs des conférences POPL et ICFP, des séminaires du LRI, de PPS, du CEA, du LIX et du LORIA, les membres de mon jury, Roberto Di Cosmo, Jean-Pierre Talpin et les deux rapporteurs de cette thèse, Gilles Barthe et Claude Marché que je remercie pour leurs remarques enrichissantes et plus généralement d'avoir accepté de faire partie de mon jury).

Pour finir, qu'est-ce qu'un programme sans ses invariants ? Les miens sont précieux et m'ont rendu plus robuste. Olivier, Hélène, Pierre-Yves et Sam, mes amis, Gaëlle, ma soeur, et bien sûr le reste de la tribu, ont toujours su me rattrapper lorsque j'étais proche du segmentation fault. Reste Marine, à la fois proche et à l'infini, là-bas, là où les parallèles se croisent ...

Résumé

Cette thèse étudie deux approches fondées sur l'analyse statique pour augmenter la sûreté de fonctionnement et la correction des programmes informatiques.

La première approche est le typage qui permet de prouver automatiquement qu'un programme s'évalue sans échouer. Le langage fonctionnel ML possède un système de type très riche et un algorithme effectuant une synthèse automatique de ces types. On s'intéresse à l'adaptation de cet algorithme aux types algébriques généralisés (GADT), une forme restreinte des inductifs de Coq, qui ont été introduits par Hongwei Xi en 2003.

Dans ce cadre, le calcul efficace d'un type plus général est impossible. On propose une stratification qui maintient les caractéristiques habituelles sur le fragment ML et qui isole le traitement des GADT en explicitant leur utilisation. Le langage obtenu, MLGX, nécessite des annotations de type qui alourdissent les programmes. Une seconde strate, MLGI, offre au programmeur un mécanisme de synthèse locale, prédictible et efficace bien qu'incomplet, de la plupart de ces annotations. La première partie s'achève avec une démonstration de l'expressivité des GADT pour coder les invariants des automates à pile utilisés par l'analyse syntaxique LR.

La seconde approche augmente le langage de programmation par des assertions logiques permettant d'exprimer des spécifications de complexité arbitraire dans la logique d'ordre supérieur polymorphiquement typée. On vérifie statiquement la conformité de la sémantique du programme vis-à-vis de ces spécifications à l'aide d'une technique appelée logique de Hoare qui consiste à engendrer un ensemble d'obligations de preuves à partir d'un programme annoté. Une fois ces obligations de preuve traitées, si un programme est utilisé correctement et si il renvoie une valeur alors il est certain que celle-ci est correcte.

Habituellement, cette technique est employée sur les langages impératifs. Avec un langage fonctionnel pur, les problèmes liés à l'état de la mémoire d'évanouissent tandis que l'ordre supérieur et le polymorphisme en posent de nouveaux. Nos choix de conceptions cherchent à maximiser les opportunités d'utilisation de prouveurs automatiques en traduisant minutieusement les objets d'ordre supérieur en objets du premier ordre. Une implantation prototype du système en fournit une illustration dans la preuve presque totalement automatique d'un module CAML d'arbres équilibrés dénotant des ensembles finis.

Abstract

This work studies two approaches to improve the safety of computer programs using static analysis. The first one is typing which guarantees that the evaluation of program cannot fail. The functional language ML has a very rich type system and also an algorithm that infers automatically the types. We focus on its adaptation to generalized algebraic data types (GADTs). In this setting, efficient computation of a most general type is impossible. We propose a stratification of the language that retain the usual characteristics of the ML fragment and make explicit the use of GADTs. The resulting language, MLGX, entails a burden on the programmer who must annotate its programs too much. A second stratum, MLGI, offers a mechanism to infer locally, in a predictable and efficient way, incomplete yet, most of the type annotations. The first part concludes on an illustration of the expressiveness of GADTs to encode the invariants of pushdown automata used in LR parsing. The second approach augments the language with logic assertions that enables arbitrarily complex specifications to be expressed. We check the compliance of the program semantics with respect to these specifications thanks to a method called Hoare logic and thanks to semi-automatic computer-based proofs. The design choices permit to handle first-class functions. They are directed by an implementation which is illustrated by the certification of a module of trees that denote finite sets.

CHAPITRE UN

Introduction

Débusquer l'erreur, le léger détail qui nous a échappé, est souvent l'obsession première de la réalisation d'une idée. Que cette idée soit un objet physique et l'ingénieur dissèque sa réalisation, idéalement jusqu'au moindre atome, par le biais de machines, de processus ou de modèles. Que cette idée soit un objet logique et le mathématicien prouve sa validité, idéalement jusqu'au moindre axiome, par le moyen de règles de raisonnement, de théorèmes et de théories pré-existantes.

Ces deux approches ne sont pas si différentes : seul change le degré d'incertitude quant à la conformité de la réalisation à l'idée. En effet, si l'ingénieur évalue et teste, c'est qu'il ne contrôle qu'une partie des paramètres du monde physique. Le mathématicien n'est pas dans une situation beaucoup plus confortable lorsqu'il utilise un théorème dont il ne connaît pas la démonstration parce qu'elle est issue d'une théorie « admise » par tous.

Le programme informatique a un statut particulier dans ce tableau puisqu'il est la réalisation physique d'un objet logique : un ordinateur peut simuler un modèle mathématique en lui donnant la forme d'une machine interrogeable à volonté. Grâce à sa puissance de calcul, l'ordinateur manipule des objets dont la complexité peut prendre de court celui qui cherche à le vérifier.

Dès lors, quelle approche adopter pour vérifier l'absence d'erreurs dans un programme informatique ? La méthode du test peut sembler s'imposer compte tenu de la complexité des systèmes : il s'agit de trouver un ensemble de couples (argument, résultat attendu) et de vérifier que le système se comporte bien sur cet ensemble. Or, de par leur caractère logique, les programmes informatiques sont souvent très généraux et cette généralité empêche toute évaluation sérieuse de leur validité à l'aide d'un ensemble fini (même très important) de tests. Il faut donc s'attarder plus précisément sur la vision mathématique d'un programme informatique pour adapter la démarche du mathématicien.

Le *medium* de transformation d'un objet logique en machine est le langage de programmation. Il s'agit d'un langage, en général adapté à l'homme, qui décrit, de manière plus ou moins abstraite, un calcul, c'est-à-dire la production d'une sortie à partir d'une entrée. Ces entrées et ces sorties sont des données structurées stockées dans la mémoire de l'ordinateur (des listes, des arbres, des dictionnaires ...). En première approximation, les théorèmes de correction des programmes informatiques sont de la forme :

Pour toute entrée I vérifiant la propriété P, le programme calcule une sortie O vérifiant Q.

ou en termes mathématiques :

$$\forall I.P(I) \Rightarrow \exists O.Q(I,O)$$

Le couple de propriétés (P,Q) constitue la *spécification* du programme, c'est-à-dire la dénotation logique de l'idée en vue de sa réalisation. Pour prouver ce genre de théorème, la difficulté est d'interpréter la description du calcul faite dans le langage de programmation pour vérifier que toutes

les exécutions possibles de la machine obtenue seront conformes aux propriétés énoncées par cette spécification.

Donner un sens aux programmes informatiques, donner une *sémantique* à un langage de programmation, est la condition *sine qua none* pour prouver des théorèmes de correction de programmes. La définition du sens des opérations calculatoires est établie à un niveau si élémentaire qu'elle rappelle celles des règles de raisonnement de la logique (comme le *modus ponens* par exemple). Cette axiomatisation fournit un système de preuves pour les programmes. Avec une armée de mathématiciens et suffisamment de temps, on doit donc pouvoir vérifier n'importe quel programme.

Malheureusement, de nouveau, la complexité des programmes informatiques rentre en scène : le nombre d'opérations atomiques dont est constitué un programme moderne est de l'ordre d'une dizaine voire d'une centaine de millions. Prouver plusieurs millions de théorèmes n'est envisageable pour personne même avec beaucoup de bonne volonté. La méthode consiste alors, comme en mathématiques, à abstraire le plus possible les opérations atomiques en fonctions réutilisables (écrire des fonctions les plus générales possibles est l'une des motivations des langages de programmation fonctionnelle qui nous intéressent plus particulièrement dans cette thèse). On factorise ainsi le travail de preuve.

Néanmoins, le nombre de preuves à effectuer reste encore très grand. Une idée à la base du domaine de cette thèse, *l'analyse statique*, est d'utiliser la puissance de calcul de la machine pour prouver automatiquement la plupart de ces théorèmes de bon fonctionnement des programmes. Pour cela, il suffit de remarquer que l'énoncé d'un théorème est une donnée comme une autre et qu'un ordinateur peut très bien vérifier, voire même trouver dans une certaine mesure, la règle de raisonnement à employer pour passer d'une étape de démonstration à une autre.

La problématique générale de cette thèse est la conception de langages de programmation facilitant la preuve, la plus automatique possible, de propriétés sur les programmes informatiques. La méthode choisie consiste à augmenter le langage de programmation par des indications, dite statiques, qui sont annexes au calcul et qui renseignent le programme de vérification de programmes sur les propriétés attendues. On étudie deux variantes de cette méthode.

La première consiste à restreindre l'ensemble des propriétés prouvables en les exprimant sous la forme d'objets syntaxiques simples, appelés « types ». En général, dans les langages de programmation classiques, les types ne capturent qu'une approximation de la forme des données traitées et fournissent des propriétés de *sûreté*. Ils permettent la suppression d'un grand nombre d'erreurs très communes en programmation comme le mélange de données incompatibles (un entier qui serait fourni à une fonction attendant une liste par exemple). Cette restriction permet au programme de vérification d'effectuer l'ensemble des preuves automatiquement. Lorsque la syntaxe de ces types et le langage de programmation sont bien choisis, il est même possible de *synthétiser* ces types, c'est-à-dire *inférer*, à partir du seul programme, les types de données qu'il manipule. Cette approche occupera la première partie de cette thèse dans laquelle on contribue à la recherche dans le domaine du *typage des langages de programmation* en fournissant une solution pratique au problème de la synthèse des types pour une extension du langage fonctionnel ML par des types algébriques généralisés (notés GADT dans la suite de ce manuscrit).

La seconde variante ne restreint pas l'expressivité des propriétés prouvables. On cherche plutôt à déterminer la quantité minimale d'informations statiques à insérer dans le programme, sous la forme d'assertions logiques, pour permettre une vérification semi-automatique des programmes : le programme de vérification attend une preuve manuelle des théorèmes non triviaux et s'occupe des théorèmes les plus simples. On obtient ainsi des programmes corrects par construction. La seconde partie de cette thèse contribue à la recherche dans le domaine de la *programmation certifiée* en formalisant un système de preuves pour un langage de programmation fonctionnelle et pure.

L'étude de ces deux variantes n'est pas un hasard. La volonté d'intégrer des preuves de propriétés sophistiquées (au delà des propriétés de sûreté) apparaît dès la première partie de cette thèse avec les

types algébriques généralisés. Nous allons montrer dans les deux sections qui suivent ce qui nous a fait progresser de l'approche centrée sur les types vers une approche centrée sur les assertions logiques. Pour ce qui suit, une connaissance du langage O'Caml (Leroy *et al*, 2004) est conseillée.

1.1 Types algébriques généralisés en ML

Dans le langage de programmation O'Caml, on peut définir des types de données pour des structures de données non bornées grâce aux types algébriques. Par exemple, on peut définir un type de données pour les listes d'entiers ainsi :

```
type list =
| Nil                  (* la liste vide. *)
| Cons of int * list   (* la liste formée d'un entier et suivie d'une liste. *)
```

La valeur *v* définie par *Cons (42, Cons (1, Nil))* est alors de type *list* et dénote une liste composée de l'entier 42 suivi de l'entier 1.

On peut manipuler ces listes à l'aide d'une construction du langage, *l'analyse de motifs*, capable de discriminer une valeur de type *list* suivant le cas *Nil* ou le cas *Cons*. La fonction suivante permet d'extraire le premier élément d'une liste :

```
let hd ℓ =
  match ℓ with
  | Nil → failwith "La liste est vide."
  | Cons (x, xs) → x
```

Lorsque cette fonction est appelée sur la liste *v*, elle renvoie *42* ce qui est le comportement attendu. Lorsque cette fonction est appelée sur la liste *Nil*, une erreur se produit : on ne peut évidemment pas extraire un élément d'une liste qui n'en contient pas !

On aimerait que le programme de vérification nous empêche d'évaluer la fonction *hd* sur une liste vide. Pour l'aider à détecter cette erreur, on peut utiliser l'extension des GADT, inventée par Hongwei Xi (Xi *et al*, 2003), pour coder la longueur de la liste à l'intérieur des types.

On se donne tout d'abord deux constructeurs de type pour dénoter les entiers :

```
type zero      (* Encode l'entier 0. *)
type 'n succ   (* Encode le successeur de n. *)
```

Le type *zero succ succ* correspond alors à l'entier 2. On redéfinit ensuite le type des listes à l'aide d'un type algébrique généralisé :

```
type 'n list =
| Nil : zero list
| Cons : forall 'm. int * 'm list → 'm succ list
```

Nous verrons les détails de cette déclaration dans le chapitre 3. Le point important, c'est qu'elle permet de donner un type plus précis à chaque constructeur : on sait maintenant que le constructeur *Nil* est de type *zero list* qui dénote une liste de taille nulle ; on sait aussi corréler la taille d'une liste et de sa sous-liste grâce à la fonction « successeur » sur les types.

Dès lors, la fonction *hd* peut être écrite différemment :

```
let hd (ℓ : 'n succ list) =
  match ℓ with
  | Cons (x, xs) → x
```

Deux changements sont apparus. Le type de *l* a été spécifié : *l* doit être une liste de taille *'n succ*. Une liste vide, nécessairement de type *zero list*, est donc exclue. Grâce à ce type plus précis, il n'est plus nécessaire d'écrire un cas d'analyse pour la liste vide. Le programme de vérification s'occupe de

vérifier que les appels à cette fonction sont toujours faits avec des entrées de type *'n succ list*. Nous avons donc corrigé le problème de la fonction *hd* en embarquant la longueur de la liste à l'intérieur du type *list*.

Malheureusement, le langage des types n'est pas assez riche pour qu'on puisse y encoder des propriétés arbitraires. Si on cherche par exemple à concaténer deux listes à l'aide de la fonction O'Caml suivante :

```
let rec concat ℓ1 ℓ2 =
  match ℓ1 with
  | Nil → ℓ2
  | Cons (x, xs) → Cons (x, concat xs ℓ2)
```

Avec la première déclaration de type algébrique, cette fonction a le type $list \rightarrow list \rightarrow list$. Avec le type algébrique généralisé, on aimerait donner à la fonction *concat* le type

$$'a\ list \rightarrow 'b\ list \rightarrow ('a + 'b)\ list$$

pour être renseigné sur la longueur de la liste construite. Seulement, la fonction d'addition entre les entiers codés par des types n'est pas définissable.

Nous avons choisi l'exemple d'un type des listes indexé par un entier de manière à montrer rapidement les limites des types algébriques généralisés pour prouver des invariants arbitrairement complexes. Cependant, de nombreux invariants sont codables à l'aide de GADT et le fait d'obtenir de nombreuses preuves automatiquement par cette méthode mérite déjà toute notre attention. Dans le chapitre 9 de cette thèse par exemple, nous utilisons les types algébriques généralisés pour encoder des invariants sur la forme de la pile d'un automate en fonction de son état courant. Nous recenserons aussi les applications des GADT qui ont été publiées ces dernières années à la fin du chapitre 9.

1.2 Système de preuves pour un langage fonctionnel pur

Plutôt que d'augmenter l'expressivité des types pour pouvoir y encoder des propriétés plus riches (Xi & Pfenning, 1999, Altenkirch *et al*, 2005, Paulin-Mohring, 1992), on rajoute à la syntaxe du langage de programmation des outils pour insérer des assertions logiques, dans l'esprit de la méthode de Hoare (1969).

Si on reprend la fonction *hd* de la section précédente, on utilise une *précondition* pour caractériser le domaine des entrées de la fonction :

```
let hd ℓ where length ℓ > 0 =
  match ℓ with
  | Nil → absurd      (* peut être omis. *)
  | Cons (x, xs) → x
```

où la fonction *length* est une fonction logique dénotant la longueur de la liste et qu'on pourrait définir inductivement de la façon suivante :

```
logic fun length =
  | Nil → 0
  | Cons (_, xs) → 1 + length xs
```

Pour montrer la validité de la fonction *hd*, il suffit au système de preuves de montrer qu'il est absurde d'avoir une liste vide et de longueur strictement positive, ce qui est trivial. De plus, à chaque appel de la fonction *hd*, le programme de vérification va exiger une preuve que la longueur de la liste passée en argument est strictement positive. Parfois, cette preuve sera triviale (si par exemple, on passe une liste constante). Dans d'autres situations, cette preuve nécessitera une aide humaine (par exemple, si la liste est issue d'un calcul complexe).

La fonction *concat* s'écrit très simplement à l'aide de cette méthode :

let rec *concat ℓ1 ℓ2 returns ℓ3* ***where*** *length (ℓ3) = length (ℓ1) + length (ℓ2) =*
  match *ℓ1* ***with***
  | *Nil* → *ℓ2*
  | *Cons (x, xs)* → *Cons (x, concat xs ℓ2)*

On affirme ici à l'aide d'une *postcondition* que la longueur de la liste calculée est la somme des longueurs des deux listes prises en entrée. Le lecteur pourra vérifier que les deux preuves à faire pour chacun des cas sont triviales. Elles sont effectivement automatiquement traitées à l'aide d'un prouveur automatique.

La seconde partie de cette thèse est donc une réponse aux difficultés rencontrées lorsque nous avons essayé d'étendre l'expressivité des GADT étudié dans la première partie.

1.3 Plan

Comme nous l'avons indiqué, cette thèse est composée de deux parties.

La première traite du problème de la synthèse des types dans une extension de ML par des GADT. Le chapitre 3 introduit les GADT à travers la programmation d'un interprète pour les termes bien typés d'un langage jouet. À la fin de ce chapitre introductif, on explique la problématique de la synthèse des types en présence de GADT et la solution adoptée, une inférence de types stratifiée. On la développe à travers la définition de deux systèmes de types, MLGX (chapitre 4) et MLGI (chapitre 5) dont on montre les corrections dans le chapitre 6. On introduit ensuite des outils appelés « formes » (chapitre 7) utiles pour la définition de deux algorithmes d'inférence locale (chapitre 8). Pour finir cette première partie, on présente en détail une application des types algébriques généralisés liée à la génération d'analyseurs syntaxiques bien typés et efficaces ainsi que quelques applications issues de la littérature.

La seconde partie débute au chapitre 10 par une introduction sur les choix de *design* du langage de programmation certifiée que nous étudions. On introduit la logique sous-jacente, la logique d'ordre supérieure polymorphiquement typée, dans laquelle sont exprimées les propriétés (chapitre 11) et on formalise ensuite le langage calculatoire de programmation à proprement parler (chapitre 12) . Le système de preuves est introduit dans le chapitre 13. Des extensions du noyau théorique sont présentées dans le chapitre 14. On les exploite au sein d'une application du système à la certification d'un module manipulant des arbres binaires de recherche équilibrés (chapitre 15). Cette partie se termine par une description de l'état de l'art et de notre positionnement dans le domaine de la programmation certifiée (chapitre 16). En guise de conclusion, la dernière partie de cette thèse établit une synthèse des travaux effectués (à la fois en termes théoriques et d'implémentations) et discute de quelques perspectives pour la recherche à venir.

CHAPITRE DEUX

Notations, conventions et rappels

Ce chapitre explicite quelques notations et conventions qu'on trouvera dans le développement de cette thèse. On en profite pour rappeler quelques notions fondamentales sur la sémantique des langages de programmation et leur typage. Le lecteur déjà familier de ces définitions peut se contenter de consulter la table des notations 2.5 qui en résume le contenu et les figures 2.2, 2.3 et 2.7 qui spécifient le langage $\mathcal{T}$ qui servira de support pour nos exemples dans le prochain chapitre.

2.1 Ensembles, listes et vecteurs

Comme la plupart des travaux à saveur mathématique, on suppose connu les bases de la théorie des ensembles. On note $\emptyset$ l'ensemble vide. On écrit $x \in S$ pour signifier qu'un élement x appartient à un ensemble S. Un ensemble formé des éléments $x_1, \ldots, x_n$ pourra être défini de manière extensionnelle par la notation $\{x_1, \ldots, x_n\}$.

On peut construire l'union $S_1 \cup S_2$ de deux ensembles S_1 et S_2, leur intersection $S_1 \cap S_2$, leur différence $S_1 \setminus S_2$. La proposition $S_1 \subseteq S_2$ signifie que S_1 est inclus dans S_2. La proposition $S_1 \# S_2$ signifie que les ensembles S_1 et S_2 sont disjoints.

L'ensemble des entiers naturels sera noté $\mathbb{N}$. Nous noterons $i, \ldots, j$ l'intervalle constitué des entiers de i à j.

On se donne une famille d'opérateurs $(\times_i)_{i \in \mathbb{N}}$. Chaque $\times_i$ est d'arité i et construit un produit cartésien de i ensembles $S_1, \ldots, S_i$ qu'on note $S_1 \times \ldots \times S_i$ (on omet l'indice). Les éléments de ce produit cartésien sont des i-uplets de la forme $(x_1, \ldots, x_i)$ tels que pour tout i, $x_i \in S_i$.

On utilise fréquemment une méta-variable pour désigner les éléments d'un ensemble particulier pour rendre implicites certaines quantifications universelles. Par exemple, i, j, k, n et m seront des méta-variables représentant des entiers naturels.

Une liste est une fonction d'un intervalle fini d'entiers $1, \ldots, n$ vers les éléments d'un ensemble S. On notera $x_1, \ldots, x_n$ pour la liste définie pour $i \in 1, \ldots, n$ par $i \mapsto x_i$. La liste vide sera notée ϵ. La concaténation de deux listes ℓ_1 et ℓ_2 sera notée $\ell_1.\ell_2$. On utilisera la notation $\ell.S$ où S est un ensemble de listes pour dénoter $\{\ell.\ell' \mid \ell' \in S\}$.

Si x est la méta-variable utilisée pour dénoter les éléments de S, on utilise la convention que $\overline{x}$ représente un ensemble fini d'éléments de S.

Une relation sur un ensemble S est un sous-ensemble de $S \times S$. Les relations seront notées de manière infixe. Une relation est une relation d'équivalence si elle est réflexive, transitive et symétrique. On notera $\widehat{x}$, la classe d'équivalence de x induite par une relation d'équivalence donnée. Il s'agit de l'ensemble des éléments y de S qui sont en relation avec x.

Termes

t	::=	n	entier
	|	$\mathsf{inc}(t)$	incrémentation
	|	$\mathsf{isz}(t)$	test à zéro
	|	$\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$	branchement
	|	(t, t)	paire
	|	$\mathsf{fst}(t)$	projection gauche
	|	$\mathsf{snd}(t)$	projection droite

Valeurs

v	::=	n	entier
	|	true	vrai
	|	false	faux
	|	(v, v)	paire

FIG. 2.1: Grammaire du langage $\mathcal{T}$ (termes et valeurs).

2.2 Définition d'une syntaxe

Dans ce manuscrit, on s'intéressera souvent à des objets syntaxiques, le plus souvent des termes du premier ordre ou d'ordre supérieur. Plutôt que d'énumérer la signature Σ de l'algèbre de termes, on utilisera une grammaire en forme BNF (Backus *et al*, 1963). Les non-terminaux (ou méta-variables) seront écrits dans une police italique. On utilisera des indices pour parler de plusieurs objets d'une même classe syntaxique. Les symboles terminaux, les éléments de Σ, seront les symboles restants, des mots-clés généralement écrits en gras ou bien des opérateurs. Par la numérotation de gauche à droite des arguments des application des symboles de Σ, on peut définir à l'aide d'une liste d'entiers la notion standard de chemin permettant de dénoter sans ambiguïté l'occurrence d'un terme dans un autre.

Exemple 2.2.1
La syntaxe des termes et des valeurs du langage $\mathcal{T}$ est donnée par la grammaire en forme BNF de la figure 2.2. La méta-variable t est utilisée pour les termes et la méta-variable v est utilisée pour les valeurs. t_1, t_2, …, t_n est une énumération de termes.

- « $\mathsf{if}\ \mathsf{isz}(\mathsf{fst}(0, 1))\ \mathsf{then}\ 1\ \mathsf{else}\ 0$ » est un terme t_1 du langage $\mathcal{T}$. L'occurrence du terme 1 correspondant au second argument de la fonction fst est accessible par le chemin $1, 1, 2$.
- « $\mathsf{isz}(\mathsf{true})$ » est un terme t_2 du langage $\mathcal{T}$. ◇

Parfois, un exposant entier servira à indiquer l'application itérée d'un symbole de fonction. Par exemple, $f^3(x)$ est un raccourci d'écriture pour $f(f(f(x)))$.

Pour définir la syntaxe de termes d'ordre supérieur, on utilisera aussi une grammaire en forme BNF mais on explicitera les lieurs. Les lieurs introduisent des variables « muettes » qu'on peut renommer sans changer le sens du terme. Une variable est libre si elle n'est pas liée. L'ensemble des variables libres d'un terme t se note $\mathbf{ftv}(t)$. Les variables liées peuvent être renommées arbitrairement tant qu'elles ne capturent pas une variable libre. Ce renommage est appelé α-conversion. Pour définir cette notion correctement, on peut se doter de la notion de variable fraîche pour un terme syntaxique qui signifie que la variable ne fait pas partie des variables libres du terme. Par abus de notations, on écrit $x \# t$ pour signifier que la variable x est fraîche pour le terme t. On consultera les travaux d'Andrew Pitts (2003) pour avoir une formalisation précise de ces notions. La substitution sans capture d'une variable libre x par un terme t_2 dans un terme t_1 sera notée $[x \mapsto t_2]t_1$. Lorsque l'on veut appliquer une composition de substitutions à un terme, remplaçant un vecteur de variables $\overline{x}$ par des termes $\overline{t}$, on écrira $[\overline{x} \mapsto \overline{t}]t$. Un renommage est une substitution particulière qui a la propriété d'être une bijection d'un ensemble fini de variables dans un autre.

Exemple 2.2.2
La figure 2.2 décrit la syntaxe du λ-calcul qui sera le noyau des langages de programmation de cette thèse. L'opérateur λ est un lieur. L'ensemble des variables libres d'un terme du λ-calcul peut être défini inductivement sur la syntaxe des termes ainsi :

Termes

t	$::=$	x	variable
	$\mid$	$\lambda x.t$	abstraction
	$\mid$	$t\ t$	application

Valeurs

v	$::=$	x	variable
	$\mid$	$\lambda x.t$	abstraction

FIG. 2.2: Grammaire du langage λ-calcul (termes et valeurs).

$$\frac{t \to n}{\mathsf{inc}(t) \to n+1} \qquad \frac{t \to n \quad n \neq 0}{\mathsf{isz}(t) \to \mathsf{false}} \qquad \frac{t \to 0}{\mathsf{isz}(t) \to \mathsf{true}} \qquad \frac{c \to \mathsf{true} \quad t_1 \to v_1}{\mathsf{if}\ c\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \to v_1}$$

$$\frac{c \to \mathsf{false} \quad t_2 \to v_2}{\mathsf{if}\ c\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \to v_2} \qquad \frac{t \to (v_1, v_2)}{\mathsf{fst}(t) \to v_1} \qquad \frac{t \to (v_1, v_2)}{\mathsf{snd}(t) \to v_2} \qquad \frac{t_1 \to v_1 \quad t_2 \to v_2}{(t_1, t_2) \to (v_1, v_2)}$$

FIG. 2.3: Sémantique opérationnelle à grands pas du langage $\mathcal{T}$.

$$\begin{array}{lcl} \mathbf{ftv}\,(x) & = & \{x\} \\ \mathbf{ftv}\,(t_1\ t_2) & = & \mathbf{ftv}\,(t_1) \cup \mathbf{ftv}\,(t_2) \\ \mathbf{ftv}\,(\lambda x.t) & = & \mathbf{ftv}\,(t) \setminus \{x\} \end{array} \qquad \diamond$$

Par la suite, nous ne redéfinirons pas l'ensemble des variables libres pour chaque nouveau langage car une définition inductive similaire de $\mathbf{ftv}\,(t)$ se déduira facilement.

2.3 Description d'une sémantique opérationnelle

On utilisera une approche opérationnelle, à base de règles d'inférence, pour donner une sémantique à nos objets syntaxiques. Nous appellerons arbres de dérivation l'ensemble des applications de règles de déduction qui conduisent à la construction d'un jugement. Les arbres de dérivations sont donc des objets inductifs sur lesquels nous ferons régulièrement des démonstrations par induction. On dira qu'un système de règles est dirigé par la syntaxe lorsqu'une seule règle d'inférence s'applique pour une construction syntaxique donnée.

Nous définirons généralement des jugements de sémantique opérationnelle dans un style à petits pas ou à grands pas (Kahn, 1987).

Exemple 2.3.1
Dans le langage $\mathcal{T}$, on choisit de noter « $t \to v$ », le jugement signifiant « le terme t s'évalue en la valeur v ». L'ensemble des règles d'inférence de la figure 2.3 en donne une définition inductive dans un style à grands pas (Kahn, 1987). Elles permettent par exemple de déduire que le terme t_1 s'évalue en la valeur « 1 ». Au contraire, aucune règle ne s'applique pour obtenir une valeur v telle que $t_2 \to v$. On dit que ce terme t_2 est bloqué. Un tel terme correspond à un programme dont l'évaluation a échoué.$\diamond$

Les sémantiques à petits pas seront définies par des règles transformant le terme à évaluer petit à petit afin d'obtenir une évaluation à l'aide d'une fermeture réflexive et transitive. Souvent, on limitera les lieux où les règles s'appliquent en définissant des contextes d'évaluation, notés $\mathbb{E}$, et une règle de passage au contexte. On notera $\mathbb{E}\,[t]$, la substitution textuelle du trou [] par le terme t.

$$\begin{array}{rcl} \mathbb{E} & ::= & [] \\ & | & v\,\mathbb{E} \\ & | & \mathbb{E}\,t \end{array}$$

FIG. 2.4: Contexte d'évaluation de la stratégie d'appel par valeur.

$$\begin{array}{lrcll} (\textsc{Beta}) & (\lambda x.t)v & \rightarrow & [x \mapsto v]t & \\ (\textsc{Context}) & \mathbb{E}\,[t_1] & \rightarrow & \mathbb{E}\,[t_2] & \text{si } t_1 \rightarrow t_2 \end{array}$$

FIG. 2.5: Sémantique opérationnelle à petits pas du λ-calcul avec une stratégie d'appel par valeur.

Exemple 2.3.2
La sémantique à petits pas du λ-calcul en appel par valeur est donnée par les figures 2.5 et 2.4. Le lecteur pourra vérifier que cette stratégie d'évaluation calcule les arguments des fonctions avant d'effectuer leur appel. Par ailleurs, cette stratégie ne réduit pas le corps des fonctions. ◇

2.4 Sémantique statique par typage

Le rôle du typage d'un langage de programmation est de fournir, par une observation statique (sans mettre en jeu l'évaluation), la garantie qu'un programme bien typé s'évalue sans erreur.

Le bon typage est souvent utilisée comme approximation conservative du bon fonctionnement d'un programme. Il s'agit d'approcher la forme des valeurs qui peuvent intervenir lors de l'évaluation d'un terme en fonction de la forme de ce terme. Pour le langage de programmation $\mathcal{T}$, ces formes – les types du langage de programmation – sont définis par la syntaxe de la figure 2.6. On peut alors définir le jugement « $\vdash t : \theta$ » qui se lit « le terme t a le type θ » par induction dans la figure 2.6.

Exemple 2.4.1
On a $\vdash t_1 : \mathsf{int}$ et il n'existe pas de type τ tel que $\vdash t_2 : \tau$. ◇

Montrer qu'un système de type joue bien son rôle, c'est montrer que le typage est un invariant de l'évaluation à l'aide un résultat de correction de la forme suivante par exemple :

Théorème 2.4.2 (Les programmes bien typés de $\mathcal{T}$ s'évaluent sans erreur (et terminent))
Si $\vdash t : \theta$ *alors* $t \rightarrow v$ *et* $\vdash v : \theta$. ◇

On pourrait démontrer cet énoncé par une induction sur la dérivation de typage de t en ayant par avance montré la corrélation entre la forme des valeurs et leurs types. S'étendre plus précisément sur les bases de la théorie du typage des langages de programmation n'est pas du ressort de cette thèse, les ouvrages (Pierce, 2002, Pottier & Rémy, 2005) seront une aide précieuse pour le lecteur ayant besoin de compléter ses connaissances sur ce domaine.

$$\begin{array}{rcll} \theta & ::= & \mathsf{int} & \text{entier} \\ & | & \mathsf{bool} & \text{booléen} \\ & | & \theta \times \theta & \text{produit} \end{array}$$

FIG. 2.6: Grammaire des types du langage $\mathcal{T}$.

$$\frac{\vdash t \ : \ \mathsf{int}}{\vdash \mathsf{inc}(t) \ : \ \mathsf{int}} \qquad \frac{\vdash t \ : \ \mathsf{int}}{\vdash \mathsf{isz}(t) \ : \ \mathsf{bool}} \qquad \frac{\vdash c \ : \ \mathsf{bool} \qquad \vdash t_1 \ : \ \theta \qquad \vdash t_2 \ : \ \theta}{\vdash \mathsf{if}\ c\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \ : \ \theta}$$

$$\frac{\vdash t_1 \ : \ \theta_1 \qquad \vdash t_2 \ : \ \theta_2}{\vdash (t_1, t_2) \ : \ \theta_1 \times \theta_2} \qquad \frac{\vdash t \ : \ \theta_1 \times \theta_2}{\vdash \mathsf{fst}(t) \ : \ \theta_1} \qquad \frac{\vdash t \ : \ \theta_1 \times \theta_2}{\vdash \mathsf{snd}(t) \ : \ \theta_2}$$

FIG. 2.7: Typage du langage $\mathcal{T}$.

2.5 Table des notations

Notation	*Signification*
i, j, k, m, n	Méta-variables dénotant des entiers naturels
$i, \ldots, j$	Sous-ensemble des entiers de i à j
$S_1 \cup S_2$	Union de deux ensembles S_1 et S_2
$S_1 \cap S_2$	Intersection entre deux ensembles S_1 et S_2
$S_1 \setminus S_2$	Différence entre deux ensembles S_1 et S_2
$S_1 \# S_2$	L'ensemble S_1 et l'ensemble S_2 sont disjoints
$S_1 \times \ldots \times S_n$	Produit cartésien des ensembles $S_1, \ldots, S_n$
$x \in S$	x appartient à l'ensemble S
ϵ	Liste vide
$\ell_1.\ell_2$	Concaténation de la liste ℓ_1 et de la liste ℓ_2.
$\ell.S$	Ensemble des listes formées par la concaténation de la liste ℓ à toutes les listes de S.
$\overline{x}$	Méta-variable dénotant un vecteur d'éléments x
$\widehat{x}$	Classe d'équivalence de l'élément x
$f^n\,(t)$	Application itérée n fois du symbole f
$\mathbf{ftv}\,(t)$	Ensemble des variables libres du terme t
$[x \mapsto t_2]t_1$	Substitution sans capture de la variable x par le terme t_2 dans le terme t_1
$[\overline{x} \mapsto \overline{t}]t$	Composition de substitutions.
$x \# t$	La variable x est fraîche pour le terme t

FIG. 2.8: Table des notations.

PREMIÈRE PARTIE

Types algébriques généralisés

CHAPITRE TROIS

Extension de ML par des GADT

Ce chapitre introductif présente d'une manière plus approfondie le langage de programmation vu dans l'introduction et qui est le sujet de la première partie de cette thèse. Pour commencer, la section 3.1 donne un exemple d'utilisation des GADT pour se familiariser informellement à leur typage. Dans la section 3.2, quelques mots sur la notion centrale d'égalité entre types introduisent les difficultés liées à la synthèse des types. Enfin, différents travaux connexes ainsi que notre contribution sont présentés dans la section 3.3.

3.1 Evaluateur de termes bien typés d'un langage minimaliste

Commençons la présentation du langage de programmation à l'aide d'un exemple. Il s'agit d'écrire un évaluateur des termes bien typés pour le langage $\mathcal{T}$ présenté dans le chapitre précédent dont les valeurs sont isomorphes aux données primitives du langage hôte, ML. Nous allons voir qu'on peut tirer parti des GADT pour supprimer l'information dynamique contenue habituellement dans les valeurs des interprètes. L'utilisation des GADT pour implémenter ces *tagless interpreters* (Cheney & Hinze, 2003) n'est pas une contribution de cette thèse mais un exemple courant de la littérature qui nous servira de support de comparaison avec les autres approches.

On commence dans la section 3.1.1 par expliquer comment implémenter un interprète en ML à l'aide de types algébriques. La section 3.1.2 met en lumière l'incapacité du système de type de ML à capturer le bon typage des termes du langage $\mathcal{T}$ pris en entrée et l'inefficacité qui en résulte. L'extension de ML par des GADT permet de passer outre ce problème grâce à un typage plus précis que nous expliquerons dans la section 3.1.3.

3.1.1 Implémentation de l'évaluateur en ML avec des ADTs

On décrit un programme ML (insatisfaisant) servant à évaluer un terme du langage $\mathcal{T}$ de la section précédente. Ce programme est composé de deux déclarations de type pour les termes et les valeurs de $\mathcal{T}$ (figure 3.1) et d'une définition récursive de la fonction d'évaluation (figure 3.3). Pour l'instant, ce programme n'utilise pas l'extension des types algébriques généralisés mais des types algébriques classiques.

Une déclaration de type algébrique introduit un nouveau type de données (ici, les types *term* et *value*) ainsi qu'un ensemble fini de constructeurs de données (ici, *Lit*, *Inc*, *IsZ*, ...). À chaque constructeur de données K est associé un schéma de type ML qui sert à déterminer le type et le nombre des arguments de K. Lorsque l'on applique un constructeur de données, on forme une valeur dont la structure est celle d'un arbre étiqueté par le nom du constructeur de données et dont les sous-arbres sont les valeurs passées en argument.

```
type term =
  Lit    : int → term
  Inc    : term → term
  IsZ    : term → term
  If     : term × term × term → term
  Pair   : term × term → term
  Fst    : term → term
  Snd    : term → term

type value =
  VInt   : int → value
  VBool  : bool → value
  VPair  : value × value → value
```

FIG. 3.1: Types algébriques dénotants les termes et les valeurs du langage $\mathcal{T}$.

Exemple 3.1.1
VPair(*VInt*(0), *VTrue*) est une valeur du type algébrique *value* qui correspond à la valeur $(0, \mathsf{true})$ du langage $\mathcal{T}$. ◇

Lors de l'exécution du programme, le nom du constructeur de données qui étiquette la valeur permet d'observer comment elle a été construite. Comme l'ensemble des constructeurs de données est fini, cette observation se réduit à une analyse par cas (aussi appelée destruction, inversion, filtrage ou *pattern matching*). Chaque cas de l'analyse est formé d'un motif et d'un terme : ce couple sera appelé une branche par la suite. Informellement, un motif suit la syntaxe des valeurs en permettant le remplacement de certains sous-arbres par des variables. Lorsqu'un motif est superposable à la valeur analysée, le code associé est évalué.

Revenons à notre évaluateur. Celui-ci va calculer des valeurs de type *value*. Or, parmi ces valeurs, on trouve des booléens, des entiers et des paires qui sont déjà présents en ML. On aimerait donc réutiliser les opérateurs de ML sur ces types. À cet effet, la figure 3.2 offre une série de fonctions très simples qui convertissent les valeurs de type *value* en valeurs ML isomorphes. Ces fonctions vont être utilisées par l'évaluateur pour tirer parti du bon typage du terme à évaluer et de ses sous-termes, c'est-à-dire de la connaissance de la forme des valeurs résultant de leurs évaluations.

Décrivons en détail la fonction *as_int*. La construction let de ML permet de lier un nom à une valeur dans la suite du programme. Ici, la syntaxe « $\lambda t. \ldots$ » nous indique qu'il s'agit d'une fonction attendant un argument t. Le terme « match t with ... » est une analyse par cas sur la forme de t. Cette analyse effectue une disjonction sur deux cas.

Le premier correspond aux valeurs de la forme *VInt*(t), c'est-à-dire aux entiers du langage $\mathcal{T}$. Dans le corps de cette branche, la variable t est liée et elle est de type *int* (ceci se déduit de la déclaration de *Lit*). On renvoie cet entier ML ce qui implémente effectivement la conversion recherchée.

Le second cas correspond à l'ensemble des autres formes de valeurs possibles (le complémentaire des motifs traités par les branches précédentes s'écrit _ dans le langage de motifs de ML). Or, la précondition de la fonction, indiquée informellement à l'aide d'un commentaire, affirme qu'on ne l'utilisera pas dans ces cas-là. On aimerait indiquer au système que, si le programme est correct, alors cette branche ne sera pas exécutée. Malheureusement, le système de type de ML n'est pas assez riche pour capturer une telle propriété (contrairement aux langages que nous allons présenter dans cette thèse). Le programmeur a alors recours à un opérateur assert e qui stoppe le programme si l'expression booléenne e est fausse. En écrivant assert false, on s'assure que le programme s'arrêtera si

```
(* Précondition : l'argument est un booléen. *)
let as_bool = λt.
  match t with
    | VBool(b) → b
    | _ → (* Cas impossible. *) assert false

(* Précondition : l'argument est un entier. *)
let as_int = λt.
  match t with
    | VInt (i) → i
    | _ → (* Cas impossible. *) assert false

(* Précondition : l'argument est une paire. *)
let as_pair = λt.
  match t with
    | VPair(t1, t2) → (t1, t2)
    | _ → (* Cas impossible. *) assert false
```

FIG. 3.2: Trois fonctions ML de conversion entre valeurs de $\mathcal{T}$ et valeurs ML.

cette branche est exécutée. Bien sûr, ce procédé est insatisfaisant puisqu'il ne donne aucune garantie quant à la correction du programme et fournit seulement un moyen de détecter *a posteriori* l'existence d'un bug.

```
(* Précondition : l'argument est un terme bien typé. *)
let rec eval : term → value = λt.
  match t with
    | Lit (i) → VInt(i)
    | Inc (t) → VInt(as_int(eval t) + 1)
    | IsZ t → VBool(as_int(eval t) = 0)
    | If (b, t, e) → if as_bool(eval b) then eval t else eval e
    | Pair (a, b) → VPair(eval a, eval b)
    | Fst (t) → fst (as_pair(eval t))
    | Snd (t) → snd (as_pair(eval t))
```

FIG. 3.3: Une fonction ML évaluant un terme du langage de programmation $\mathcal{T}$.

Passons enfin à l'écriture de l'évaluateur d'un terme du langage $\mathcal{T}$. Comme ses règles d'évaluation sont dirigées par la syntaxe (voir la figure 2.7), on écrit naturellement la fonction *eval* à l'aide d'une définition récursive discrimant sur chaque cas de sa syntaxe (figure 3.3).

Le type de cette fonction est $term \rightarrow value$: on retrouve la forme du jugement d'évaluation $t \rightarrow v$. Focalisons-nous sur la construction de la valeur correspondant à l'évaluation d'un terme de la forme $IsZ(t)$, en supposant que le terme t a le type int. D'après la règle d'évaluation, on doit d'abord évaluer le terme t en une valeur v. Ensuite, si la valeur v est un entier non nul, on renvoie la valeur $VBool(\mathsf{false})$ et on renvoie $VBool(\mathsf{true})$ si la valeur v est un entier nul. L'utilisation de la fonction de comparaison entre entiers de ML s'impose mais celle-ci a le type $int \rightarrow int \rightarrow bool$. Il faut donc convertir v en une valeur de type int. C'est le rôle de *as_int* dont la précondition est vérifiée grâce au bon typage supposé du terme à évaluer. Enfin, comme la fonction d'évaluation doit renvoyer une valeur de type

$$
\begin{array}{l}
\mathsf{type}\ \mathit{term}\ : \star \to \star = \\
\quad \mathit{Lit}\ \ : \forall\alpha.\mathit{int} \to \mathit{term}\ \mathbf{?} \\
\quad \mathit{Inc}\ \ : \forall\alpha.\mathit{term}\ \mathit{int} \to \mathit{term}\ \mathbf{?} \\
\quad \mathit{IsZ}\ \ : \forall\alpha.\mathit{term}\ \mathit{int} \to \mathit{term}\ \mathbf{?} \\
\quad \mathit{If}\ \ \ : \forall\alpha.\mathit{term}\ \mathit{bool} \times \mathit{term}\ \alpha \times \mathit{term}\ \alpha \to \mathit{term}\ \alpha \\
\quad \mathit{Pair} : \forall\alpha.\mathit{term}\ \mathbf{?} \times \mathit{term}\ \mathbf{?} \to \mathit{term}\ \mathbf{?} \\
\quad \mathit{Fst}\ \ : \forall\alpha.\mathit{term}\ \mathbf{?} \to \mathit{term}\ \mathbf{?} \\
\quad \mathit{Snd}\ \ : \forall\alpha.\mathit{term}\ \mathbf{?} \to \mathit{term}\ \mathbf{?}
\end{array}
$$

FIG. 3.4: Une tentative de codage du bon typage des termes dans ML.

value, il est nécessaire d'encapsuler cette valeur de type *bool* à l'aide du constructeur *VBool*.

3.1.2 Problème de l'évaluateur écrit en ML sans GADT

Qu'arrive-t-il si on utilise la fonction *eval* de la section précédente avec un terme mal typé (comme par exemple $\mathit{IsZ}(\mathit{VPair}(\mathit{VInt}(0), \mathit{VInt}(0)))$? L'appel à *as_int* échoue lors de l'évaluation. Lorsque le terme est bien typé dans le langage $\mathcal{T}$, cette fonction introduit également une inefficacité car elle réalise un isomorphisme trivial (en supprimant l'étiquette de la valeur de type *term*).

On aimerait obtenir du typeur une *garantie statique* nous assurant que la propriété de bon typage du terme fourni en entrée est toujours vraie. Pour cela, deux méthodes sont envisageables, chacune prenant un point de vue différent sur le programme. Ou bien on enrichit la *spécification* de la fonction d'évaluation en indiquant au typeur qu'elle n'accepte qu'un sous-ensemble des valeurs de type *term*, ce sera la méthode retenue dans la seconde partie de cette thèse (§II). Ou bien on restreint les valeurs de type *term* à ne contenir que des termes bien typés. Autrement dit, on aimerait que la propriété « être bien typé » soit vraie par construction pour toute valeur de type *term*. Cette méthode est permise par un enrichissement du système de type de ML à l'aide de types algébriques généralisés.

3.1.3 Évaluateur de termes bien typés en ML avec GADT

Pour encoder l'invariant de bon typage des termes, nous allons utiliser les paramètres de type de ML pour y encoder non pas la forme d'une donnée mais *une propriété dénotée par un type*. On introduit donc le type paramétré *term* α qui est donné aux valeurs *dont l'évaluation conduit à une valeur de type* α. Le type α ne correspond ainsi pas au type d'une des composantes du terme mais au type de la valeur qui résulte de son évaluation.

Si ce codage semble attirant, sa mise en oeuvre est loin d'être évidente en ML sans GADT. En effet, si on essaie de rajouter un tel paramètre à la définition de *term*, on est rapidement bloqué comme l'illustre la figure 3.4. Expliquons-en les raisons. Dans cette figure, la notation $\star \to \star$ correspond à la sorte du constructeur de type *term*. Elle indique seulement que le constructeur est d'arité 1.

Tout d'abord, on constate que le seul type satisfaisant est celui donné à *If*. En effet, il exprime bien le fait que la condition du branchement (le premier argument) doit s'évaluer en un booléen et que si chacune des branches s'évaluent en une valeur de type α alors le branchement s'évalue en une valeur de type α.

En revanche, le type associé à *Lit* ne correspond pas à nos voeux : $\mathit{Lit}(0)$ a le type *term* α, ce qui est bien trop général. On aimerait plutôt lui donner le type *term int*, c'est-à-dire donner le type $\mathit{int} \to \mathit{term}\ \mathit{int}$ au constructeur *Lit*. De même, si t_1 a le type *term* τ_1 et t_2 a le type *term* τ_2 alors on souhaite que $\mathit{Pair}(t_1, t_2)$ ait le type *term* $(\tau_1 \times \tau_2)$. Cela revient à donner au constructeur *Pair*, le schéma de type $\forall\beta_1\beta_2.\mathit{term}\ \beta_1 \times \mathit{term}\ \beta_2 \to \mathit{term}\ (\beta_1 \times \beta_2)$.

$$
\begin{array}{l}
\mathsf{type}\ \mathit{term}\ : \star \to \star = \\
\quad \mathit{Lit}\ \ : \mathit{int} \to \mathit{term}\ \mathit{int} \\
\quad \mathit{Inc}\ \ : \mathit{term}\ \mathit{int} \to \mathit{term}\ \mathit{int} \\
\quad \mathit{IsZ}\ \ : \mathit{term}\ \mathit{int} \to \mathit{term}\ \mathit{bool} \\
\quad \mathit{If}\ \ \ : \forall\alpha.\mathit{term}\ \mathit{bool} \times \mathit{term}\ \alpha \times \mathit{term}\ \alpha \to \mathit{term}\ \alpha \\
\quad \mathit{Pair} : \forall\beta_1\beta_2.\mathit{term}\ \beta_1 \times \mathit{term}\ \beta_2 \to \mathit{term}\ (\beta_1 \times \beta_2) \\
\quad \mathit{Fst}\ \ : \forall\beta_1\beta_2.\mathit{term}\ (\beta_1 \times \beta_2) \to \mathit{term}\ \beta_1 \\
\quad \mathit{Snd}\ \ : \forall\beta_1\beta_2.\mathit{term}\ (\beta_1 \times \beta_2) \to \mathit{term}\ \beta_2
\end{array}
$$

FIG. 3.5: Types algébriques généralisés dénotant les termes bien typés du langage $\mathcal{T}$.

Ces schémas de type ne correspondent pas aux schémas habituels donnés aux constructeurs de données. En ML, tous les schémas de type des constructeurs de données du constructeur de type *term* doivent avoir la forme :

$$\forall\alpha.\tau_1 \times \ldots \times \tau_n \to \mathit{term}\ \alpha$$

La nouveauté apportée par les types algébriques généralisés, c'est d'autoriser pour les constructeurs du type *term* n'importe quel schéma de type de la forme :

$$\forall\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \mathit{term}\ \tau$$

Cette forme de schéma de type diffère de la forme précédente car elle permet de spécifier un type particulier τ pour le paramètre de type de *term* et elle augmente l'ensemble des variables de type $\overline{\beta}$ du schéma. Cette valeur particulière du paramètre et les variables de type $\overline{\beta}$ peuvent être différentes pour chaque constructeur de données. Nous verrons un peu plus loin quelles conséquences a cette liberté sur le bon typage.

Pour le moment, fort est de constater que ce relâchement nous permet d'écrire une déclaration de type algébrique généralisé correspondant à nos *desiderata* (figure 3.5).

Il est possible de montrer que les valeurs construites à l'aide de ces constructeurs de données sont exactement les termes bien typés du langage $\mathcal{T}$ grâce au théorème suivant :

Théorème 3.1.2 (Codage du bon typage dans le type *term*)
Soit T la valeur ML dénotant le terme t du langage $\mathcal{T}$ et τ le type ML dénotant le type θ du langage $\mathcal{T}$. T est de type term τ si et seulement si $\vdash t : \theta$. ◇

Preuve. La preuve est immédiate par induction sur $\vdash t : \theta$ car il y a un isomorphisme clair entre les règles de typage dirigées par la syntaxe du langage $\mathcal{T}$ et les schémas de type des constructeurs de données du type *term*. □

Les exemples suivants permettent d'illustrer le bon comportement de ce codage :

Exemple 3.1.3
- « $\mathit{Inc}(\mathit{Lit}(1))$ » est bien typé et de type $\mathit{term}\ \mathit{int}$.
- « $\mathit{IsZ}(\mathit{VPair}(\mathit{VInt}(0), \mathit{VInt}(0)))$ » est mal typé. ◇

Comment cette extension du système de type influe-t-elle sur l'écriture de la fonction d'évaluation et sur son bon typage ? Avant d'aborder cette question, il est intéressant d'observer un autre formalisme de déclaration des types généralisés tel qu'on le trouve dans la figure 3.6.

Ce formalisme s'appuie sur une syntaxe équivalente pour les schémas de type qui fait intervenir une égalité de type :

$$\forall\overline{\beta}\alpha[\alpha = \tau].(\tau_1 \times \ldots \times \tau_n) \to \mathit{term}\ \alpha$$

$$
\begin{array}{l}
\mathsf{type}\ term\ = \\
\quad Lit\ \ : \forall\alpha[\alpha = int].int \to term\ \alpha \\
\quad Inc\ \ : \forall\alpha[\alpha = int].term\ int \to term\ \alpha \\
\quad IsZ\ \ : \forall\alpha[\alpha = bool].term\ int \to term\ \alpha \\
\quad If\ \ \ : \forall\alpha.term\ bool \times term\ \alpha \times term\ \alpha \to term\ \alpha \\
\quad Pair : \forall\beta_1\beta_2\alpha[\alpha = \beta_1 \times \beta_2].term\ \beta_1 \times term\ \beta_2 \to term\ \alpha \\
\quad Fst\ \ : \forall\beta_1\beta_2\alpha[\alpha = \beta_1].term\ (\beta_1 \times \beta_2) \to term\ \alpha \\
\quad Snd\ \ : \forall\beta_1\beta_2\alpha[\alpha = \beta_2].term\ (\beta_1 \times \beta_2) \to term\ \alpha
\end{array}
$$

FIG. 3.6: Un autre formalisme pour la déclaration des types algébriques généralisés.

$$
\begin{array}{l}
\mathsf{let\ rec}\ eval : \forall\alpha.term\ \alpha \to \alpha = \lambda t. \\
\quad \mathsf{match}\ t\ \mathsf{with} \\
\quad\quad \mid\ Lit\ (i) \to (*\ \alpha = int\ *)\ i \\
\quad\quad \mid\ Inc\ (t) \to (*\ \alpha = int\ *)\ eval\ t + 1 \\
\quad\quad \mid\ IsZ\ (t) \to (*\ \alpha = bool\ *)\ eval\ t = 0 \\
\quad\quad \mid\ If\ (b,\ t,\ e) \to \mathsf{if}\ eval\ b\ \mathsf{then}\ eval\ t\ \mathsf{else}\ eval\ e \\
\quad\quad \mid\ Pair\ (a,\ b) \to (*\ \exists\beta_1\beta_2.\alpha = \beta_1 \times \beta_2\ *)\ (eval\ a, eval\ b) \\
\quad\quad \mid\ Fst\ (t) \to (*\ \exists\beta_1\beta_2.\alpha = \beta_1\ *)\ fst\ (eval\ t) \\
\quad\quad \mid\ Snd\ (t) \to (*\ \exists\beta_1\beta_2.\alpha = \beta_2\ *)\ snd\ (eval\ t)
\end{array}
$$

FIG. 3.7: Une fonction ML évaluant un terme du langage $\mathcal{T}$ en utilisant des GADT.

où les variables de type $\overline{\beta}$ interviennent dans les types $\tau_1, \ldots, \tau_n$ et τ.

L'explicitation des égalités de type modifie légèrement le point de vue sur l'application d'un constructeur de données. En effet, on remarque que chaque constructeur a une égalité de type qui lui est propre. Par conséquent, par rapport à ML, l'application d'un constructeur de données requiert la validation d'une égalité de type supplémentaire. En contrepartie, lors de l'analyse d'une valeur construite à partir de ce constructeur particulier, on peut sans risque supposer que l'égalité de type correspondante est vérifiée, ce qui constitue un *apport d'information statique*.

Voyons maintenant comment cet apport contribue à l'implémentation de la fonction d'évaluation décrite dans la figure 3.7.

Cette fonction a maintenant un type plus précis car il *synchronise* le type de la valeur calculée et le type du terme à évaluer par le biais du partage de la variable de type α. Cette propriété plus forte permet d'éviter les étapes de conversion. Par exemple, comme le type de b dans la branche traitant le *If* est *term bool*, on sait que *eval b* est un booléen qui peut être testé par le branchement de ML.

Ce type précis est accepté grâce aux égalités de type qu'on acquiert lorsque l'on a plus d'information sur la forme du terme à évaluer. En guise d'illustration, considérons le cas où la valeur analysée est de la forme $Lit(i)$. La déclaration de la figure 3.6 nous indique que la valeur analysée a nécessairement été construite dans un contexte où l'égalité $\alpha = int$ était vraie. Le typeur peut alors supposer localement dans le corps de cette branche que cette égalité de type est vraie. Dès lors, comme la fonction d'évaluation doit renvoyer une valeur de type α et que le terme i est de type *int*, le corps de la branche est bien typé.

Le cas *Pair* est un peu plus complexe. D'après le schéma de type de *Pair*, lorsque l'on apprend que t est de la forme $Pair(a, b)$, on apprend qu'au moment de la construction de cette valeur, il existait des types β_1 et β_2 tels que a ait le type *term* β_1 et b ait le type *term* β_2. En outre, l'égalité $\alpha = \beta_1 \times \beta_2$

était aussi vérifiée. Toutes ces précieuses informations de typage peuvent être recouvrées dans le corps de la branche associée à ce motif. Ainsi, de l'évaluation de a résulte une valeur de type β_1, de celle de b une valeur de type β_2. Le couple $(eval(a), eval(b))$ est donc du type $\beta_1 \times \beta_2$, c'est-à-dire un type égal à α ce qui rend la branche bien typée. Dans le cas de *Pair*, notons que des variables universellement quantifiées (β_1 et β_2) sont introduites automatiquement par le typeur.

Exhaustivité plus précise

Pour finir, la précision accrue des schémas de type des constructeurs de données induit un meilleur diagnostic statique de l'exhaustivité des *pattern matchings*. En effet, reprenons l'exemple de la fonction *eval* mais spécialisons son type en $term\ int \to int$. On restreint le domaine d'application de l'évaluateur en n'évaluant que des termes de type *int*. Le typeur peut donc déterminer statiquement que la branche *IsZ* peut être supprimée sans atteinte à l'exhaustivité de l'analyse car les valeurs construites à partir de *IsZ* sont de type $term\ bool$.

3.2 Égalité entre types

Le traitement des égalités de type associées aux constructeurs de données est au centre des problèmes de vérification et d'inférence de type lorsque l'on ajoute les GADT à ML. Dans cette section, nous tâchons de donner informellement une idée de ces difficultés.

3.2.1 Règle de conversion

Si on tente de formaliser les arguments qui nous ont convaincus du bon typage de la fonction *eval* de la section 3.1.3, plusieurs ingrédients interviennent.

En premier lieu, la présence d'égalités de type locales aux corps des branches induit l'existence d'un ensemble d'égalités de type valides en chaque point du programme. Du point de vue de la formalisation, cela nécessite l'introduction d'un nouvel objet au sein des hypothèses du jugement de typage : la liste E des égalités de types récoltées dans le contexte où est plongé le terme dont on veut déterminer le type. En conséquence, la forme du jugement de typage pour ML muni de GADT est

$$\Gamma, E \vdash t : \tau$$

et se lit « dans l'environnement de typage Γ et sous les hypothèses d'égalités de type E, le terme t a le type τ ».

Pour saisir le rôle de E, il faut alors se poser les deux questions suivantes : Quand augmente-t-on le système d'équations E ? Quand utilise-t-on les égalités du système E ?

L'augmentation de E est effectuée par le typage des match. En effet, la confrontation du type du terme analysé (dans $eval\ \ t$ qui a le type $term\ \alpha$) et du type du motif (ici, le motif $Lit(i)$ qui a le type $term\ int$) implique une égalité entre types (par exemple, l'égalité $\alpha = int$). Le système d'équations E est donc localement augmenté dans la branche par cette égalité.

L'utilisation de E est moins localisée dans le sens où elle ne dépend pas de la forme du terme qu'on cherche à typer. En d'autres termes, aucune indication syntaxique ne permet de décider où utiliser les équations de E. En fait, la contribution de E est implicite car elle contraint le typage à s'effectuer *modulo* toutes les égalités de type impliquées par E. Nous avons utilisé cette propriété dans notre exemple pour montrer que le corps de la branche avait bien le type attendu pour le retour de la fonction *eval*. Plus précisément, voici la dérivation que nous avons utilisé dans notre explication informelle :

$$\frac{eval : \forall\alpha.term\ \alpha \to \alpha, t : term\ \alpha, i : int; \alpha = int \vdash i : int \qquad \alpha = int \models \alpha = int}{eval : \forall\alpha.term\ \alpha \to \alpha, t : term\ \alpha, i : int; \alpha = int \vdash i : \alpha}$$

Elle se lit ainsi : sachant que le terme i est un entier et que le système d'équations E (ici $\alpha = int$) implique l'égalité $\alpha = int$ alors le terme i est aussi de type α. Cette forme de règle de déduction est appelée *règle de conversion* car elle permet de convertir le type d'un terme en raisonnant non pas sur le terme mais sur son type. Voici la version générale de cette règle :

$$\frac{\Gamma, E \vdash t : \tau_2 \qquad \Gamma, E \models \tau_1 = \tau_2}{\Gamma, E \vdash t : \tau_1}$$

Cette règle de conversion ainsi qu'une légère modification de la règle habituelle pour typer le *pattern matching* sont les seules modifications à apporter à ML pour introduire les types algébriques généralisés. La décidabilité du bon typage dépend alors de la décidabilité des implications entre égalités de type (comme le montre l'hypothèse de la forme $\Gamma, E \models \tau_1 = \tau_2$). Or, si on se limite aux types standards de ML, ces implications sont décidables efficacement par l'algorithme appelé *congruence closure* (Baader & Nipkow, 1998, Nelson & Oppen, 1980).

3.2.2 Problème de l'inférence de types

Un des traits caractéristiques du langage de programmation ML est l'inférence de types découverte par Roger Hindley et Robin Milner (Damas & Milner, 1982). L'objectif de l'inférence de types est de calculer un type principal (le plus général) pour tout programme ML bien typé. L'adaptation de cet algorithme est la difficulté principale posée par les types algébriques généralisés. Dans cette section, nous allons essayer de fournir une intuition sur la nature de cette difficulté.

Perte de la principalité

La première difficulté réside en la *perte de la principalité*. En effet, une fois les GADT introduits dans le langage, un programme ML ne possède plus un type plus général que tous les autres. Pour le comprendre, appuyons-nous sur l'exemple de la figure 3.8.

$$\begin{array}{l}\mathsf{type\ eq} = \\ \quad \mid Eq : \forall\alpha.\mathsf{eq}\ \alpha\ \alpha \\ \\ \mathsf{let}\ cast = \lambda e.\lambda x.\mathsf{match}\ e\ \mathsf{with}\ Eq \to x\end{array}$$

FIG. 3.8: Une fonction possédant plusieurs types principaux.

Ces deux schémas de type ML sont acceptables pour la fonction *cast* :
- $\forall\alpha\beta\gamma.\ \mathsf{eq}\ \alpha\ \beta \to \gamma \to \gamma$
- $\forall\alpha\beta.\ \mathsf{eq}\ \alpha\ \beta \to \alpha \to \beta$
- $\forall\alpha\beta.\ \mathsf{eq}\ \alpha\ \beta \to \beta \to \alpha$

Si le premier schéma doit sembler naturel au lecteur intime avec l'inférence habituelle de ML, le second schéma nécessite une explication (le troisième schéma est du même acabit). Une valeur de type eq peut être vue comme la preuve d'une égalité entre deux types. Ainsi, lorsque l'on apprend que le terme e, de type $\mathsf{eq}\ \alpha\ \beta$, est Eq, de type $\mathsf{eq}\ \delta\ \delta$ (pour un certain δ), on apprend du même coup l'égalité $\alpha = \beta = \delta$. Or, comme x est de type α, x est aussi de type β ce qui justifie le type de retour affecté à la fonction *cast*.

Le point important, c'est qu'aucun de ces schémas de type n'est plus général que l'autre (aucune instanciation des paramètres de l'un ne permet de retrouver l'autre et réciproquement). Nous verrons

dans la section suivante qu'il faut généraliser la syntaxe des types pour réussir à trouver un type plus général que ces trois là.

Cette perte de la principalité pose diverses problèmes. Tout d'abord, la principalité est une propriété au centre de l'algorithme efficace d'inférence de types de Hindley-Milner car elle permet de traiter l'inférence de manière incrémentale sans retour en arrière. Ensuite, si un programme ne possède pas un unique schéma de type plus général, doit-on le rejeter, doit-on calculer l'ensemble des schémas de type plus généraux valides pour ce programme ou peut-on se permettre de faire un choix arbitraire entre tous les schémas possibles ?

Détermination et utilisation des égalités de type

La seconde difficulté est inhérente à la présence de l'ensemble des égalités de type contenues dans le jugement de typage. Dans le cadre de l'inférence de types, les deux questions de la section précédente, « quelles sont les nouvelles égalités de type ? » et « où utilise-t-on ces égalités ? » ne sont plus indépendantes mais au contraire *interdépendantes*.

Pour approfondir notre intuition sur ce problème d'inférence de types, on peut rappeler les travaux de François Pottier et Vincent Simonet (Simonet & Pottier, 2007) sur HMG(X). Ils généralisent la réduction de l'inférence de types de ML à la résolution de contraintes sous préfixes mixtes (Pottier & Rémy, 2005) en augmentant le langage des contraintes par des implications. Sans rentrer dans les détails, la contrainte à résoudre dans le cas de *cast* est de la forme :

$$\ldots \mathsf{def}\ x : \gamma_1\ \mathsf{in}\ \gamma_2 = \gamma_3 \Rightarrow \gamma_1 = \gamma_4 \ldots$$

Les variables $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ sont les variables de la contrainte de typage. Ici, la variable γ_1 correspond au type de la variable x, les variables γ_2 et γ_3 sont telles que le terme e a le type $\mathsf{eq}\ \gamma_2\ \gamma_3$ et la variable γ_4 représente le type de retour de la fonction *cast*. Pour résoudre cette contrainte, plusieurs opérations sont utiles.

Pour traiter cette implication, on doit naturellement explorer plusieurs voies de résolution. L'une suppose la partie gauche valide dans la partie droite c'est-à-dire $\gamma_2 = \gamma_3$ et effectue la résolution des égalités entre types *modulo* cette égalité. La seconde suppose la partie gauche invalide et relâche alors la contrainte en ignorant la partie droite de l'implication. On touche du doigt une explosion combinatoire dans la résolution de ces contraintes et, effectivement, dans le cas général, ce nouveau langage de contraintes nécessite un algorithme de résolution dont la complexité est non élémentaire.

Un autre point important mis à jour par l'étude de HMG(X) (Simonet & Pottier, 2007) est la nécessité d'étendre la syntaxe des schémas de type pour retrouver la propriété de principalité du typage. Seuls des schémas de type contraints, *i.e.* de la forme $\forall\overline{\alpha}[C].\tau$, permettent de capturer dans toute sa généralité le type d'un programme ML utilisant des GADT. Par exemple dans HMG(=) (Simonet & Pottier, 2007), le schéma de type contraint le plus général et valide pour la fonction *cast* est :

$$\forall\gamma_1\gamma_2\gamma_3\gamma_4.[\gamma_1 = \gamma_2 \Rightarrow \gamma_3 = \gamma_4].\mathsf{eq}\ \gamma_1\ \gamma_2 \to \gamma_3 \to \gamma_4$$

Cependant, cette forme de schémas de type ne nous semblent pas adéquate pour un langage de programmation car elle fait apparaître des contraintes dont les formes résolues sont très lourdes et dont la résolution ne peut pas être effectuée « de tête » par le programmeur.

3.2.3 Solution retenue pour l'inférence de types

MLGX, une dose d'explicite pour retrouver la principalité

La section précédente a montré dans quelle mesure la détermination des égalités de type et leur utilisation étaient sujets à des choix interdépendants. Le caractère implicite de la règle de conversion

et la nature disjonctive de l'implication sont à l'origine de ce non-déterminisme. C'est pourquoi nous choisissons de supprimer ces deux traits du langage en élaborant un langage plus explicite que nous appelons MLGX.

Nous supprimons tout d'abord la règle de conversion implicite au profit d'une explicitation syntaxique de l'utilisation des égalités de type. Pour cela, on introduit un opérateur de *coercion* de la forme :

$$(t : \tau_1 \triangleright \tau_2)$$

Il signifie au système de type qu'un changement explicite du type d'un terme est nécessaire en indiquant qu'un terme dont le type est τ_1 doit être vu comme un terme de type τ_2. Bien évidemment, cette construction est acceptée par le système de type seulement si les égalités de type du contexte établissent effectivement l'égalité $\tau_1 = \tau_2$.

Nous supprimons ensuite la nécessité des implications de type dans le langage de contraintes en *forçant l'expression observée par un pattern matching à être annotée par un type rigide ou connu.* Dès lors, on extrait des égalités de type simplement en confrontant le type de chaque motif et ce type connu. Cette confrontation est locale et peut être effectué indépendamment de la vérification et de la synthèse des types. On peut donc déterminer en tout point du programme, dans une passe préliminaire, les égalités de type qui y sont valides.

Dans MLGX, la fonction *cast* s'écrit :

$$\mathsf{let}\ \mathit{cast} = \forall\alpha\beta.\lambda e\, x.\mathsf{match}\ (e : \mathsf{eq}\ \alpha\ \beta)\ \mathsf{of}\ \mathit{Eq} \rightarrow (x : \alpha \triangleright \beta)$$

Pour pouvoir spécifier le type rigide de la valeur analysée à l'aide d'une annotation, on a introduit deux variables de type universellement quantifiées. Ensuite, on explicite la coercion du type α de la variable x en un type β.

Il n'y a pas d'ambiguïté sur le schéma de type de *cast*, il s'agit de

$$\forall\alpha\beta.\ \mathsf{eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \beta$$

Plus généralement, cette dose d'explicitation permet à MLGX de recouvrer des types principaux. Mieux encore, la suppression des implications de type dans le langage de contraintes permet de retrouver l'inférence de types dans le style de Hindley-Milner. En effet, une fois vérifiée la validité des coercions, on peut interpréter $(x : \alpha \triangleright \beta)$ comme l'application de la fonction identité (sans contenu calculatoire) de type $\alpha \rightarrow \beta$ ce qui est un cas standard d'application de fonction en ML.

MLGX est décrit dans le chapitre 4 de cette thèse.

MLGI, une dose d'inférence locale prévisible pour alléger le travail d'annotation

Le point noir de MLGX est la lourdeur des annotations de type. Pour s'en donner une idée, la figure 8.3 explicite les coercions et les annotations nécessaires pour écrire la fonction *eval* dans MLGX.

En observant ces annotations, on constate que certaines sont difficiles à inférer tandis que d'autres se déduisent facilement du contexte. Plus précisément, on a vu précédemment que déterminer le schéma de type d'une fonction utilisant des GADT est très complexe [1]. L'annotation soulignée dans la figure devrait donc être requise. Par contre, une fois cette annotation connue, les annotations en gris sont déduites facilement. Notons la présence de variables de type dans les motifs, comme dans le motif « *Pair* β_1 β_2 *a b* ». Il s'agit d'une introduction de variables de type universellement quantifiées dont le scope est limité au corps de la branche de ce motif.

La solution proposée pour décharger le programmeur de ce fardeau qu'est l'annotation est un algorithme *d'inférence locale* (aussi appelé *élaboration*) qui va insérer automatiquement des coercions

[1] En vérité dans le cadre de la récursion polymorphe, c'est même un problème indécidable.

```
let rec eval : ∀α.term α → α = ∀α.λt.
  match (t : term α) with
    | Lit i → (i : (int ▷ α))
    | Inc t → (eval t + 1 : (int ▷ α))
    | IsZ t → (eval t = 0 : (bool ▷ α))
    | If b t e → if eval b then eval t else eval e
    | Pair β1 β2 a b → ((eval a, eval b) : (β1 × β2 ▷ α))
    | Fst β1 β2 t → fst (eval t)
    | Snd β1 β2 t → snd (eval t)
```

FIG. 3.9: La fonction d'évaluation du langage $\mathcal{T}$ dans MLGX.

et des annotations pour transformer un programme écrit dans un style semi-implicite (dans un langage nommé MLGI) en un programme de MLGX.

Cet algorithme ne va pas chercher à résoudre une contrainte globale de typage pour déduire les annotations car cela reviendrait à traiter le problème général de l'inférence de types. Au contraire, on ne va pas explorer l'ensemble des choix de typage possibles mais, plus modestement et du mieux possible, se contenter de *propager les annotations* de l'utilisateur à l'aide d'un algorithme *prévisible* pour le programmeur. En d'autres termes, MLGI doit permettre au programmeur de travailler virtuellement dans MLGX sans avoir à écrire les annotations « évidentes ».

Apparaît alors une difficulté inhérente à l'incomplétude de la solution choisie : dans quelle mesure s'assurer que les annotations écrites par le programmeur suffisent bien à déterminer quel programme MLGX il a en tête ? Nous avons mis en place plusieurs outils techniques pour nous prémunir contre les choix arbitraires que pourrait faire notre algorithme d'inférence locale. Au centre de ces outils, on trouve le concept de *forme* qui permet de restreindre la propagation des annotations de l'utilisateur à un contexte local. Nous consacrons le chapitre 7 à cet outil.

En résumé : une approche stratifiée pour l'inférence des types algébriques généralisés

L'originalité de notre approche est d'avoir séparé le traitement du problème difficile de l'inférence des types algébriques généralisés de l'inférence de types dans le style de Hindley-Milner grâce à une inférence *stratifiée*.

La strate MLGX fournit une *vérification* de la bonne utilisation des types algébriques généralisés ainsi qu'une *inférence de types correcte et complète* pour la partie ML.

La strate MLGI fournit un traitement *incomplet*, mais *correct et prévisible* de l'inférence des types généralisés.

3.3 Travaux connexes et contributions

Une des premières apparitions des types algébriques généralisés dans le domaine des langages programmation provient d'un article de Hongwei Xi (Xi *et al*, 2003). Il met à plat les règles de typage et les résultats méta-théoriques ainsi qu'un grand nombre d'applications pertinentes des GADT à la programmation.

Dans le domaine de la preuve assistée par ordinateur, un fragment des inductifs de Coq correspond à l'expressivité des GADT. Néanmoins, la règle CONV de Coq ne tient pas compte des égalités entre types obtenues par inversion. Le principe d'inversion de la construction match de Coq n'utilise les équations que sur les types des branches. L'exemple suivant déclare un type algébrique généralisé en Coq.

```
Inductive ty : Type → Type :=
  | Nat : ty nat
  | Bool : ty bool.
```

Malheureusement, le raffinement de type n'a pas lieu si on utilise le *pattern matching* de Coq. En effet, la fonction suivante n'est pas acceptée alors que son équivalent l'est dans MLGI.

```
Definition print (a : Type) (t : ty a) (y : a) : nat :=
  match t with
    | Nat ⇒ y + 1
    | Bool ⇒ 0
  end.
```

Le type de la variable y n'est pas raffinée. Coq rejette ce programme en arguant que le type a et le type nat ne sont pas compatible dans la première branche.

On peut transformer ce programme pour qu'il soit accepté par Coq de la façon suivante :

```
Definition print (a : Type) (t : ty a) (y : a) : nat :=
  match t in (ty T) return (T → nat) with
    | Nat ⇒ fun (y0 : nat) ⇒ y0 + 1
    | Bool ⇒ fun (y0 : bool) ⇒ 0
  end y.
```

La méthode consiste à effectuer une η-expansion pour faire apparaître le type de la variable y dans le type de la branche pour que l'inversion lui applique le raffinement. Le passage à l'échelle de cette méthode n'est bien sûr pas envisageable.

Si les détails formels de l'extension du système de type de ML par des GADT sont en grande partie fixés par les travaux de Hongwei Xi, le problème de la synthèse des types occupe encore de nombreux auteurs car la solution à base d'inférence bidirectionnelle de Hongwei Xi ne permet pas une extension stricte des programmes ML (certains programmes ML ne sont plus acceptés par le système). Plusieurs auteurs ont intégré les GADT dans leur langage de programmation (C♯ (Kennedy & Russo, 2005), Scala (Odersky & Zenger, 2005), Haskell (Peyton Jones *et al*, 2004)) avec plus ou moins de difficultés.

Comme François Pottier et Vincent Simonet, Peter J. Stuckey et Martin Sulzmann (Sulzmann *et al*, 2006) réduisent le problème de l'inférence de types à la résolution de contraintes mettant en jeu des implications. Cependant, leur approche est différente dans le sens où ils développent des solveurs incomplets qui peuvent faire des choix arbitraires ou explorer un grand nombre de typages possibles. L'incomplétude de leur solveur est très inconfortable pour le programmeur car ils ne caractérisent pas un sous-ensemble de programmes bien typés et acceptés à coup sûr par leur système. L'ensemble des programmes acceptés est caractérisé uniquement par leur algorithme complexe de résolution. Il y a peu de chance qu'un programmeur n'ayant pas d'expertise dans ce domaine puisse prédire la validité de ce qu'il écrit (et donc puisse comprendre si son programme est rejeté à cause d'une erreur de type ou par la faute d'une incomplétude du solveur). À notre avis, l'approche stratifiée que nous proposons donne au programmeur un cadre clair et complet (MLGX) qui lui permet de raisonner simplement et de faire accepter n'importe quel programme bien typé par le simple processus de rajout d'annotations de type.

Simon Peyton Jones, Geoffrey Washburn, et Stephanie Weirich (Peyton Jones *et al*, 2004) font une proposition, implémentée depuis la version 6.4 du compilateur Haskell GHC, qui semble bien fonctionner en pratique. Comme nous, ils suggèrent d'utiliser intensivement les annotations de type fournies par le programmeur. Si on reprend l'exemple de la fonction *eval*, Simon Peyton Jones *et al.*

estiment qu'une fois fourni le schéma de type de *eval*, il est « clair » que la valeur analysée par le *pattern matching* est de type *term* α et ainsi, il devient « clair » que l'égalité $\alpha = int$ devient valable dans la première branche. Plutôt que de raisonner à l'aide d'implications ou de système d'équations explicites, il choisit alors d'appliquer la substitution $[\alpha \mapsto int]$ sur l'environnement de typage et sur le type attendu pour le corps de la première branche et de continuer à traiter le reste du programme suivant l'algorithme d'inférence de types habituel. Ainsi, il reste seulement à vérifier que l'expression i est de type *int*, ce qui est immédiat.

Bien que l'idée première de Simon Peyton Jones *et al* soit très simple, il s'avère que l'imbrication de l'inférence dans le style de Hindley-Milner d'une part et du processus de propagation des annotations et d'application des substitutions induites par les égalités de types d'autre part est plus complexe qu'on pourrait l'imaginer. En effet, il est essentiel que le processus de raffinement de type (l'application de la substitution) soit insensible à l'ordre dans lequel l'algorithme d'inférence traverse l'arbre de syntaxe représentant le programme. En d'autres termes, la connaissance des égalités de type ne doit pas dépendre de l'état de l'algorithme d'unification du premier ordre, maillon central de l'inférence de ML. Dès lors, Simon Peyton Jones *et al.* choisissent de bloquer la contribution des types inférés dans la propagation des annotations par le moyen d'une barrière syntaxique, un constructeur de type dit *wobbly*. Maintenir cette barrière demande des opérations « administratives » qui parasitent à notre avis la compréhension du processus d'inférence. Plutôt que de maintenir une barrière entre deux processus concurrents (la propagation des annotations de type et l'inférence dans le style de Hindley-Milner), nous choisissons de les séparer en deux passes.

Si la solution de Simon Peyton Jones et la nôtre sont très proches quant à la volonté d'utiliser uniquement les annotations de type pour traiter la partie GADT de l'inférence de types, notre formalisation stratifiée permet de donner de l'implémentation de GHC une vision plus simple et nous a permis à l'occasion d'y trouver des défauts. Nous avons alors pu développer un algorithme qui accepte des programmes bien typés que GHC rejette. Nous en donnerons des exemples dans la section 7.3.

3.4 Plan de la partie 1

Le développement formel de cette première partie concernant les types algébriques généralisés commence dans le chapitre 4 par une présentation de la syntaxe, de la sémantique opérationnelle, du système de type et d'inférence de types de MLGX. Le chapitre 5 introduit brièvement MLGI. Le chapitre 6 détaille les résultats méta-théoriques sur ces deux systèmes. L'étude du treillis des formes, de ses opérations et de ses propriétés occupe le chapitre 7. On s'appuie sur ces objets pour décrire deux algorithmes d'inférence locale dans le chapitre 8. Enfin, une application originale des GADT, l'implémentation efficace et sûre d'analyseurs syntaxiques LR, est décrite en détail dans le chapitre 9.

CHAPITRE QUATRE

MLGX

Dans ce chapitre, nous formalisons MLGX, un langage qui étend ML avec des types algébriques généralisés où les coercions de type sont explicites et les *pattern matching* sont annotés. On décrit d'abord la syntaxe et la sémantique dynamique, puis le système de type et enfin l'algorithme de synthèse des types. MLGX est une contribution de cette thèse qui montre comment intégrer un fragment explicitement typé effectuant une *vérification* du bon usage des GADT dans un langage muni de l'inférence de types dans le style de Hindley-Milner.

4.1 Syntaxe et sémantique

La syntaxe de MLGX est décrite par la figure 4.1. Nous décrivons chaque catégorie syntaxique dans les paragraphes suivants.

▶ **Types** Un type τ est soit une variable de type α (ou γ ou β), soit un type de fonction $\tau_1 \to \tau_2$, soit une application d'un constructeur de type algébrique ε aux vecteurs $\overline{\tau}_1$ et $\overline{\tau}_2$ (l'explication de la présence de deux groupes de paramètres apparaît dans le paragraphe suivant).

Un schéma de type $\forall\overline{\alpha}.\tau$ lie un vecteur $\overline{\alpha}$ de variables de type dans un type τ. Chaque type de la forme $[\overline{\alpha} \mapsto \overline{\tau}]\tau$ est une instance du schéma de type $\forall\overline{\alpha}.\tau$. On écrit $\sigma \preceq \tau$ quand τ est une instance de σ.

De la même façon, une annotation de type simple $\exists\overline{\gamma}.\tau$ lie $\overline{\gamma}$ dans le type τ ; une annotation de type polymorphe $\exists\overline{\gamma}.\sigma$ lie $\overline{\gamma}$ dans le schéma de type σ ; et une coercion $\exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)$ lie $\overline{\gamma}$ dans la paire de types $(\tau_1 \triangleright \tau_2)$. Les relations d'instance induites s'écrivent $\theta \preceq \tau$, $\varsigma \preceq \sigma$, et $\kappa \preceq (\tau_1 \triangleright \tau_2)$.

Quand c'est possible, nous suivons la convention informelle que les méta-variables $\overline{\alpha}$ et $\overline{\beta}$ représentent des variables de type « rigides » (abstraites), c'est-à-dire liées universellement dans le contexte alors que la méta-variable $\overline{\gamma}$ représente des variables « flexibles » (à inférer), c'est-à-dire liées existentiellement dans le contexte. Cependant, certains noms de variables jouent des rôles différents dans différents contextes. Ainsi, cette convention ne tient pas toujours et on tâchera de lever l'ambiguïté dès qu'elle se présentera. Par la suite, nous supposerons aussi que les notations $\overline{\alpha}$, $\overline{\beta}$ et $\overline{\gamma}$ représentent un vecteur de variables de type distinctes.

▶ **Constructeurs de types algébriques** Nous supposons donnés un certain nombre de constructeurs de types algébriques, que nous notons ε. Chacun de ces constructeurs de types est paramétré par deux groupes de paramètres de types : leur application s'écrit $\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$, où $\overline{\tau}_1$ et $\overline{\tau}_2$ sont des vecteurs de types. Nous ferons référence aux paramètres du premier groupe en les appelant *ordinaires* et à ceux du second groupe en les appelant *généralisés*. Quand le second groupe est vide, ε est un *type algébrique*

Types	$\tau ::=$	
Variable de type		α
Type des fonctions	$\mid$	$\tau \to \tau$
Type algébrique	$\mid$	$\varepsilon\,\overline{\tau}\,\overline{\tau}$
Schémas de type	$\sigma ::=$	$\forall\overline{\alpha}.\tau$
Annotations de type simple	$\theta ::=$	$\exists\overline{\gamma}.\tau$
Annotations de type polymorphe	$\varsigma ::=$	$\exists\overline{\gamma}.\sigma$
Coercions de type	$\kappa ::=$	$\exists\overline{\gamma}.(\tau \triangleright \tau)$
Termes	$t ::=$	
Variable		x
Fonction	$\mid$	$\lambda(x:\theta).t$
Application de fonction	$\mid$	$t\ (t)$
Définition locale	$\mid$	$\mathsf{let}\, x\ =\ t\ \mathsf{in}\ t$
Point fixe	$\mid$	$\mu(x:\varsigma).t$
Application de constructeur de données	$\mid$	$K\,((t,\ldots,t))$
Analyse par cas	$\mid$	$\mathsf{match}\,(t:\theta)\,\mathsf{with}\,\overline{c}$
Introduction de variable de type	$\mid$	$\forall\overline{\alpha}.t$
Annotation de type	$\mid$	$(t:\theta)$
Coercion de type	$\mid$	$(t:\kappa)$
Branches	$c ::=$	$p.t$
Motifs	$p ::=$	$K\,\overline{\beta}\,\overline{x}$

FIG. 4.1: Types et termes de MLGX.

ordinaire iso-existentiels de ML (Läufer & Odersky, 1994), sinon ε est un *type algébrique généralisé*. Par exemple, le type *term* du précédent chapitre possède un unique paramètre de type généralisé et aucun paramètre de type ordinaire. L'*arité* du constructeur de type ε est la somme des longueurs de ses deux vecteurs de paramètres.

Distinguer ces deux groupes nous permet de traiter les types algébriques ordinaires et les types algébriques généralisés d'une manière uniforme plutôt que d'introduire des notions différentes. Ce choix sert à éliminer une possible redondance dans notre présentation. De plus, cette approche augmente l'expressivité du système : un type algébrique peut avoir à la fois des arguments de type ordinaires, inférés par l'inférence dans le style de Hindley-Milner et des arguments généralisés qui doivent être explicitement précisés par le programmeur ou inférés par l'algorithme d'inférence locale. Enfin, cette distinction nous permet de constater que MLGX est effectivement une stricte extension conservative de ML (tout programme ML sans GADT est traité par une inférence de types classique).

▶ **Constructeurs de données** Nous supposons qu'à chaque constructeur de type algébrique est associé un ensemble de constructeurs (de données), notés K. Chaque constructeur de données se voit affecter un schéma de type clos *via* une déclaration de la forme :

$$K :: \forall\overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\alpha}\,\overline{\tau},$$

où $\overline{\alpha} \# \overline{\beta}$ et $\mathbf{ftv}(\overline{\tau}) \subseteq \overline{\beta}$. (Comme le schéma de type est clos, nous avons aussi $\mathbf{ftv}(\tau_1, \ldots, \tau_n) \subseteq \overline{\alpha}\overline{\beta}$.) Le nombre n est l'arité du constructeur K et la taille des vecteurs $\overline{\alpha}$ et $\overline{\beta}$ est déterminée par ε.

La longueur du vecteur $\overline{\beta}$ correspond au nombre de variables de type introduites par K. Nous noterons $K \preceq \sigma$ lorsque σ est une instance du schéma de type de K.

Quand le second groupe de paramètres est vide, la déclaration du constructeur de données se simplifie en :

$$K :: \forall \overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\, \overline{\alpha}$$

On retrouve alors les types iso-existentiels dans le style de Läufer-Odersky (Läufer & Odersky, 1994). Ainsi, nos types algébriques ordinaires contiennent cette extension. Si, en outre, le vecteur $\overline{\beta}$ est vide alors la déclaration se simplifie encore :

$$K :: \forall \overline{\alpha}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\, \overline{\alpha}$$

Cette dernière restriction est celle des types algébriques standards de ML (Pottier & Rémy, 2005, p.456).

▶ **Termes** Dans chaque λ-abstraction, la variable liée x est décorée par une annotation de type simple θ. La version non annotée de la λ-abstraction $\lambda x.t$ peut être définie comme un sucre syntaxique pour $\lambda(x : \exists\gamma.\gamma).t$. En effet, l'annotation de type $\exists\gamma.\gamma$ peut toujours être instanciée en un type τ quelconque.

Dans chaque point fixe $\mu(x : \varsigma).t$, la variable liée x porte une annotation de type polymorphe ς pour éviter les problèmes d'indécidabilité de la synthèse de type en présence de récursion polymorphe (Henglein, 1993). Quand ς est $\exists\gamma.\gamma$, le type de x est inféré mais doit être monomorphe.

Chaque *pattern matching* fait intervenir un vecteur de branches $\overline{c}$. Une branche est une paire formée d'un motif p et d'un terme t. Lorsque le motif est de la forme $K\,\overline{\beta}\,\overline{x}$, $\overline{\beta}$ est un ensemble de variables de type liées dans le terme t. Par ailleurs, la longueur de $\overline{\beta}$ doit correspondre au nombre de variables de type introduites par K. Les identifiants $\overline{x}$ sont aussi liés dans le terme t.

L'existence des $\overline{\beta}$ dans les motifs forcent le nommage des variables de type introduites par les constructeurs de type algébriques iso-existentiels. Cette construction rappelle le `open` de Läufer-Odersky (Läufer & Odersky, 1994) où les variables de type introduites doivent aussi être nommées. Néanmoins, en pratique, si ces variables ne sont pas utilisées dans le terme t alors elles peuvent être omises par le programmeur et insérées automatiquement par le système en s'appuyant sur la déclaration du constructeur de données.

En pratique, le langage de motifs doit être plus riche en permettant l'expression de motifs de profondeur supérieure à 1 et des opérateurs comme la disjonction, la conjonction ou les motifs attrape-tout. Pour simplifier la présentation, nous ne traitons pas ces motifs. Leur ajout sera discuté dans la conclusion de cette thèse.

Dans la construction $\forall\overline{\alpha}.t$, les variables de type $\overline{\alpha}$ sont liées dans le terme t. Elles sont interprétées comme étant liées universellement ce qui signifie qu'il faut que le terme t soit bien typé pour toute instanciation possible de ces variables de type. En pratique, on devrait aussi permettre la construction duale $\exists\overline{\alpha}.t$, où les variables de type sont liées existentiellement, de manière à ce que le terme t puisse être bien typé *modulo* une certaine instanciation de ces variables. Dans cette formalisation, on cantonne l'introduction de variables liées existentiellement aux annotations de type θ et ς ainsi qu'aux coercions de type κ. On en tire un invariant qui simplifie la présentation : si une variable de type apparaît libre dans une annotation alors elle est rigide puisqu'elle a pu être introduite dans le terme seulement par une quantification universelle.

Valeurs	$v ::=$	
Variable		x
Fonction		$\mid \lambda(x : \theta).t$
Données		$\mid K\,((v_1, \ldots, v_n))$
Contexte d'évaluation	$\mathbb{E} ::=$	
Focus		$[]$
À gauche de l'application		$\mid \mathbb{E}\ t$
À droite de l'application		$\mid v\ \mathbb{E}$
À gauche d'un let		$\mid \mathsf{let}\, x\ =\ \mathbb{E}\ \mathsf{in}\, t$
À droite d'un let		$\mid \mathsf{let}\, x\ =\ v\ \mathsf{in}\, \mathbb{E}$
À valeur à analyser		$\mid \mathsf{match}\, \mathbb{E}\ \mathsf{with}\ \overline{c}$

FIG. 4.2: Contextes d'évaluation et valeurs de MLGX.

(E-CONTEXT)	$\mathbb{E}\,[t]$	$\rightarrow$	$\mathbb{E}\,[t']$ si $t \rightarrow t'$
(E-BETA)	$(\lambda(x : \theta).t)v$	$\rightarrow$	$[x \mapsto v]t$
(E-LET)	$\mathsf{let}\, x\ =\ v\ \mathsf{in}\, t$	$\rightarrow$	$[x \mapsto v]t$
(E-FIX)	$\mu(x : \varsigma).t$	$\rightarrow$	$[x \mapsto \mu(x : \varsigma).t]t$
(E-MATCH)	$\mathsf{match}\, K\,((v_1, \ldots, v_n))\ \mathsf{with}\ K\,\overline{\beta}\ x_1 \ldots x_n.t \mid \overline{c}$	$\rightarrow$	$[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]t$
(E-NOMATCH)	$\mathsf{match}\, K'\,((v_1, \ldots, v_m))\ \mathsf{with}\ K\,\overline{\beta}\ x_1 \ldots x_n.t \mid \overline{c}$	$\rightarrow$	$\mathsf{match}\, K'\,((v_1, \ldots, v_m))\ \mathsf{with}\ \overline{c}$ si $K \neq K'$

FIG. 4.3: Sémantique opérationnelle à petits pas de MLGX.

4.2 Sémantique opérationnelle

La sémantique de MLGX n'est pas différente de celle de ML muni de types algébriques. La figure 4.2 décrit les environnements d'évaluation de l'appel par valeur : on évalue les arguments d'une fonction avant de l'exécuter, le terme gauche d'un let avant son terme droit et évidemment la valeur à analyser avant les branches dans un *pattern matching*. La figure 4.3 décrit les règles d'évaluation. On y trouve la β-réduction, la réduction des valeurs récursives, la réduction du *pattern matching* et la réduction sous un contexte d'évaluation.

4.3 Contraintes et systèmes d'équations

4.3.1 Syntaxe

Une contrainte C encode un problème d'unification où « unification » signifie unification du premier ordre sous un préfixe mixte (Miller, 1992). Les contraintes vont dénoter des problèmes de synthèse de types. La figure 4.4 contient la spécification de leur syntaxe. On trouve les constructions qui

Contraintes	$C ::=$	
Vérité		$\mid \mathsf{true}$
Égalité		$\mid \tau = \tau$
Conjonction		$\mid C \wedge C$
Quantification existentielle		$\mid \exists \overline{\gamma}.C$
Quantification universelle		$\mid \forall \overline{\alpha}.C$
Instanciation		$\mid x \preceq \tau$
Définition		$\mid \mathsf{def}\ x : \forall \overline{\alpha}[C].\tau\ \mathsf{in}\ C$
Système d'équations entre types	$E ::= \mathsf{true} \mid \tau = \tau \mid E \wedge E$	

FIG. 4.4: Système d'équations entre types et contraintes.

modélisent naturellement l'unification du premier ordre sous préfixe mixte : l'égalité, la conjonction et les quantifications universelles et existentielles.

Reste à expliquer les deux dernières constructions. La contrainte $\mathsf{def}\ x : \forall \overline{\alpha}[C].\tau\ \mathsf{in}\ C$ définit un sous-problème dont la solution est un schéma de type et qu'on nomme à l'aide de la variable x. La contrainte $x \preceq \tau$ vérifie que le schéma associé à la variable x est instanciable en τ. D'un point de vue sémantique, ces constructions sont redondantes avec les autres constructions standards mais elles capturent de manière plus précises les propriétés des contraintes de type associées aux programmes ML et favorisent une résolution plus efficace car elles modélisent un partage dans le processus de résolution. Pour une description très approfondie de ces constructions, on pourra consulter le chapitre sur ce sujet écrit par François Pottier et Didier Rémy (Pottier & Rémy, 2005).

Un système d'équations E est une conjonction (éventuellement vide) d'égalités entre types. Les systèmes d'équations sont utilisés dans les jugements de typage pour garder la trace des équations de type introduites par le *pattern matching*. Les variables de type intervenant dans les types des équations du système E sont rigides, c'est-à-dire universellement quantifiées dans le contexte.

4.3.2 Interprétation logique des contraintes

Nous ne détaillons pas la sémantique des contraintes qui est déjà expliquée dans la formalisation de HM(X) (Pottier & Rémy, 2005). La satisfiabilité, l'implication et l'équivalence des contraintes sont définies *via* l'interprétation standard dans l'univers de Herbrand, c'est-à-dire au sein du modèle des arbres finis. On écrit $C_1 \models C_2$ quand la contrainte C_1 implique la contrainte C_2. On écrit $C_1 \equiv C_2$ lorsque deux constraintes sont équivalentes, c'est-à-dire lorsque $C_1 \models C_2$ et $C_2 \models C_1$.

On définit aussi $C \models \exists \overline{\gamma}.(\tau_1 \triangleright \tau_2)$ comme un sucre syntaxique pour $C \models \forall \overline{\gamma}.(\tau_1 = \tau_2)$. Intuitivement, $C \models \kappa$ signifie que, d'après la contrainte C, toute instance de la coercion κ est valide.

4.4 Système de type

Les règles de typage de MLGX sont données par la figure 4.5. Le jugement de typage est de la forme

$$E, \Gamma \vdash t : \tau$$

et se lit « dans l'environnement Γ, sous le système d'équations E, le terme t a le type τ ».

La plupart de ces règles sont issues de ML. La règle X-FORALL n'est pas très courante (Pottier & Rémy, 2005) mais est orthogonale à l'ajout des GADT. Enfin, les règles X-CASE, X-CLAUSE, X-PAT sont adaptées pour les GADT. Comme certains changements par rapport à la présentation initiale

$$\text{X-Var}\ \frac{(x:\sigma)\in\Gamma}{E,\Gamma\vdash x:\sigma} \qquad \text{X-Lam}\ \frac{E,(\Gamma;x:\tau_1)\vdash t:\tau_2 \qquad \theta\preceq\tau_1}{E,\Gamma\vdash\lambda(x:\theta).t:\tau_1\to\tau_2} \qquad \text{X-App}\ \frac{E,\Gamma\vdash t_1:\tau_1\to\tau_2 \qquad E,\Gamma\vdash t_2:\tau_1}{E,\Gamma\vdash t_1\ (t_2):\tau_2}$$

$$\text{X-Let}\ \frac{E,\Gamma\vdash t_1:\sigma \qquad E,(\Gamma;x:\sigma)\vdash t_2:\tau}{E,\Gamma\vdash \mathsf{let}\, x\ =\ t_1\ \mathsf{in}\ t_2:\tau} \qquad \text{X-Fix}\ \frac{E,(\Gamma;x:\sigma)\vdash t:\sigma \qquad \varsigma\preceq\sigma}{E,\Gamma\vdash\mu(x:\varsigma).t:\sigma}$$

$$\text{X-Cstr}\ \frac{K\preceq\tau_1\times\ldots\times\tau_n\to\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \qquad \forall i\quad E,\Gamma\vdash t_i:\tau_i}{E,\Gamma\vdash K\,(t_1,\ldots,t_n):\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2} \qquad \text{X-Case}\ \frac{E,\Gamma\vdash(t:\theta):\tau_1 \qquad \forall i\quad E,\Gamma\vdash(p_i:\theta).t_i:\tau_1\to\tau_2}{E,\Gamma\vdash\mathsf{match}\,(t:\theta)\,\mathsf{with}\,p_1.t_1\ldots p_n.t_n:\tau_2}$$

$$\text{X-Forall}\ \frac{E,\Gamma\vdash t:\tau \qquad \overline{\alpha}\#\mathbf{ftv}\,(E,\Gamma)}{E,\Gamma\vdash\forall\overline{\alpha}.t:\forall\overline{\alpha}.\tau} \qquad \text{X-Annot}\ \frac{E,\Gamma\vdash t:\tau \qquad \theta\preceq\tau}{E,\Gamma\vdash(t:\theta):\tau}$$

$$\text{X-Coerce}\ \frac{E,\Gamma\vdash t:\tau_1 \qquad \kappa\preceq(\tau_1\triangleright\tau_2) \qquad E\models\kappa}{E,\Gamma\vdash(t:\kappa):\tau_2} \qquad \text{X-Gen}\ \frac{E,\Gamma\vdash t:\tau \qquad \overline{\alpha}\#\mathbf{ftv}\,(E,\Gamma,t)}{E,\Gamma\vdash t:\forall\overline{\alpha}.\tau}$$

$$\text{X-Inst}\ \frac{E,\Gamma\vdash t:\sigma \qquad \sigma\preceq\tau}{E,\Gamma\vdash t:\tau} \qquad \text{X-Clause}\ \frac{p:\varepsilon\,\overline{\tau}_1\,\overline{\tau}'_2\vdash(\overline{\beta},E',\Gamma') \qquad E\wedge E',\Gamma\Gamma'\vdash t:\tau_2 \qquad \overline{\beta}\#\mathbf{ftv}\,(E,\Gamma,\tau_2) \qquad \overline{\gamma}\#\mathbf{ftv}\,(E,\Gamma,t,\tau_2)}{E,\Gamma\vdash(p:\exists\overline{\gamma}.\varepsilon\star\overline{\tau}'_2).t:\varepsilon\,\overline{\tau}_1\star\to\tau_2}$$

$$\text{X-Pat}\ \frac{K\preceq\forall\overline{\beta}.\tau_1\times\ldots\times\tau_n\to\varepsilon\,\overline{\tau}_1\,\overline{\tau} \qquad \overline{\beta}\#\mathbf{ftv}\,(\overline{\tau}_1,\overline{\tau}_2)}{K\,\overline{\beta}\,x_1\ldots x_n:\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2\vdash(\overline{\beta},\overline{\tau}_2=\overline{\tau},(x_1:\tau_1;\ldots;x_n:\tau_n))}$$

FIG. 4.5: **ML** avec GADT dans un style explicite.

des GADT (Xi *et al*, 2003) ont dû être effectués pour coller au style explicite de MLGX, on décrit maintenant chacune des règles.

► **X-Var** La règle X-VAR se contente d'extraire le schéma de type de la variable stocké dans l'environnement. Cette règle est tout à fait standard.

► **X-Lambda** La règle LAMBDA diffère de la règle habituelle de ML car on doit tenir compte de l'annotation que porte l'argument. Dans le cadre de la vérification de types, on s'assure que cette annotation correspond au type de l'entrée. Pour vérifier cette correspondance, il faut se souvenir qu'une annotation θ est de la forme $\exists\overline{\gamma}.\tau$ et qu'une hypothèse de la forme $\theta\preceq\tau'$ implique la recherche d'une instanciation pour les $\overline{\gamma}$ de manière à égaliser τ et τ'. Cette annotation sert à contraindre le type de la fonction.

► **X-App** La règle X-APP vérifie que l'argument de l'application correspond bien au type d'entrée de la fonction et propage le type de retour. Cette règle est la même que pour le λ-calcul simplement typé.

▶ **X-Let** Dans la règle X-Let, on attend un schéma de type pour la partie gauche du let. Il s'agit de la généralisation sur le let caractéristique de ML (Damas & Milner, 1982). Ce schéma de type est ensuite associé à la variable liée par la définition locale dans la suite du programme.

▶ **X-Fix** La règle X-Fix s'occupe de vérifier la validité de l'annotation de type qui doit être au moins aussi générale que le schéma de type de la valeur récursive. Ce même schéma de type est associé dans l'environnement de typage à la variable x en cours de définition.

▶ **X-Cstr** La règle X-Cstr utilise le jugement $K \preceq \tau_1 \times \ldots \times \tau_n \to \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2$ pour vérifier que les arguments passés au constructeur de données correspondent bien aux types déclarés dans son schéma de type. Notons qu'une application d'un constructeur de données est obligatoirement totale dans un programme bien typé.

▶ **X-Annot** On vérifie que les annotations écrites par le programmeur dans MLGX sont des instances du type attendu pour le terme comme pour les règles X-Lambda et X-Fix.

▶ **X-Coerce** La règle X-Coerce vérifie qu'une coercion κ est applicable à un terme t de type τ_1 pour le convertir en un terme de type τ_2. Pour cela, on vérifie que la coercion κ s'instancie bien en la coercion voulue $(\tau_1 \triangleright \tau_2)$ et on montre $E \models \kappa$ qui va s'assurer que pour toute instanciation des variables flexibles de la coercion κ, l'égalité entre les deux types à convertir est valide. Notons que lorsque $E \equiv \mathsf{false}$, on peut donner n'importe quel type à un terme puisque toutes les égalités entre types sont valides dans un tel contexte.

▶ **X-Gen** La règle X-Gen construit un schéma de type à l'aide de toutes les variables de type libre dans le type d'un terme et qui ne sont pas contraintes par l'environnement de typage. Ces variables doivent aussi être fraîches pour le terme et le système d'équations E.

▶ **X-Inst** La règle X-Inst est duale de la règle X-Gen. Son rôle est d'instancier un schéma de type de manière à obtenir un type monomorphe fixé.

▶ **X-Forall** Introduire des variables de type universellement quantifiées ne pose pas de problème en ML puisque les jugements de typage acceptent des schémas de type. On doit s'assurer de la fraîcheur des variables de type choisies par rapport au terme t, au système d'équations E et à l'environnement de typage Γ. Ces variables de type « rigides » peuvent être utilisées dans les annotations de type et donc apparaître dans le type τ.

▶ **X-Case** La règle X-Case traite chacune des branches en utilisant l'annotation de type dont est décoré le terme analysé après avoir vérifié qu'elle est effectivement correcte. L'annotation servira à établir localement le nouveau système d'équations valable dans le corps des branches. Formellement, cette propagation s'exprime par l'appel au jugement $(p_i : \theta).t_i$.

▶ **X-Clause** Le jugement $(p : \theta).t$ est défini par la règle X-Clause. Cette règle est au cœur du traitement des GADT, décrivons-la en détail.

La conclusion de la règle indique d'abord que l'annotation doit avoir la forme :

$$\exists \overline{\gamma}.\varepsilon \star \overline{\tau}'_2$$

On utilise $\star$ pour indiquer que l'ensemble des paramètres ordinaires ne sont pas pris en compte par la règle. Restent donc les paramètres généralisés qu'on nomme $\overline{\tau}'_2$. L'ensemble des paramètres

ordinaires $\overline{\tau}_1$ retenus pour appliquer cette règle est issu du type $\varepsilon\,\overline{\tau}_1\star$. Le type τ_2 correspond au type du corps de la branche.

La première hypothèse $p : \varepsilon\,\overline{\tau}_1\,\overline{\tau}'_2 \vdash (\overline{\beta}, E', \Gamma')$ utilise le type $\varepsilon\,\overline{\tau}_1\,\overline{\tau}'_2$ ainsi construit pour déduire $(\overline{\beta}, E', \Gamma')$ qui est engendré par le filtrage du motif p. Les variables de type $\overline{\beta}$ sont les variables de type supplémentaires introduites par le motif, le système d'équations E' est l'ensemble des égalités de type acquises et l'environnement de typage Γ' associe un type à chaque variable du motif. La règle X-Pat explique la manière exacte dont ils sont déduits du motif p et du type $\varepsilon\,\overline{\tau}_1\,\overline{\tau}'_2$.

La seconde hypothèse $E \wedge E', \Gamma\Gamma' \vdash t : \tau_2$ tient compte de ce fragment pour typer le corps de la branche avec le type attendu τ_2 en augmentant l'ensemble des équations et l'ensemble des variables liées.

▶ **X-Pat** L'application de la règle X-Pat se décompose en deux étapes. D'abord, on instancie le schéma de type de K de manière à ce que les paramètres ordinaires valent $\overline{\tau}_1$. Une fois cette instanciation effectuée, on déduit les types $\tau_1, \ldots, \tau_n$ de $x_1, \ldots, x_n$ ainsi que l'égalité entre le type des paramètres généralisés de la déclaration de K et les types $\overline{\tau}_2$ issus de l'annotation du *pattern matching*.

C'est donc l'annotation de type, et seulement elle, qui sert à déterminer *localement* les équations de type dans MLGX. En effet, si on observe à la fois la règle X-Clause et X-Pat, on voit que seules la valeur des paramètres généralisés $\overline{\tau}'_2$ issue de l'annotation de type et la valeur des paramètres généralisés $\overline{\tau}$ rentrent en compte pour former l'équation $\overline{\tau}'_2 = \overline{\tau}$.

Dès lors, on peut faire la remarque suivante très importante pour la suite : si les variables de type flexibles $\overline{\gamma}_2$ n'apparaissent pas dans les types $\overline{\tau}'_2$ alors $E, \Gamma \vdash t : \varepsilon\,\overline{\tau}_1\overline{\tau}'_2$ si le terme t est la valeur analysée. Dans ce cas, on obtient une *connaissance totale* du système d'équations E puisqu'on connaît le type exact de la valeur analysée.

▶ **Exemple de dérivation de typage dans MLGX** En guise d'exemple d'utilisation des règles de MLGX, nous décrivons en détail la dérivation de typage d'une branche de la fonction *eval*. Le terme suivant doit avoir le type α dans l'environnement $\Gamma \equiv \mathit{eval} : \forall\alpha.\mathit{term}\ \alpha \to \alpha; t : \mathit{term}\ \alpha$.

$$\begin{array}{l}\mathsf{match}\,(t : \mathit{term}\ \alpha)\,\mathsf{with}\\ \quad \mid \mathit{Lit}\ i \to (i : (\mathit{int} \triangleright \alpha))\\ \quad \mid \ldots\end{array}$$

Pour cela, on doit appliquer la règle X-Case :

$$\text{X-Case}\ \frac{E, \Gamma \vdash (t : \mathit{term}\ \alpha) : \mathit{term}\ \alpha \qquad E, \Gamma \vdash (\mathit{Lit}\ i : \mathit{term}\ \alpha).(i : (\mathit{int} \triangleright \alpha)) : \mathit{term}\ \alpha \to \alpha}{E, \Gamma \vdash \mathsf{match}\,(t : \mathit{term}\ \alpha)\,\mathsf{with}\ \mathit{Lit}\ i \to (i : (\mathit{int} \triangleright \alpha)) : \alpha}$$

Le jugement portant sur t est une simple application de la règle X-Var. Pour la seconde prémisse, on doit appliquer la règle X-Clause et la règle X-Pat :

$$\text{X-Clause}\ \frac{\alpha = \mathit{int}, \Gamma; (i : \mathit{int}) \vdash (i : (\mathit{int} \triangleright \alpha)) : \alpha \qquad \text{X-Pat}\ \dfrac{\mathit{Lit} \preceq \mathit{int} \to \mathit{term}\ \mathit{int}}{\mathit{Lit}\ i : \mathit{term}\ \alpha \vdash (\emptyset, \alpha = \mathit{int}, (i : \mathit{int}))}}{E, \Gamma \vdash (\mathit{Lit}\ i : \mathit{term}\ \alpha).(i : (\mathit{int} \triangleright \alpha)) : \mathit{term}\ \alpha \to \alpha}$$

Enfin, on vérifie que la coercion est valide à l'aide de la règle X-Coerce :

$$\text{X-Coerce}\ \frac{E, \Gamma \vdash i : \mathit{int} \qquad (\mathit{int} \triangleright \alpha) \preceq (\mathit{int} \triangleright \alpha) \qquad \alpha = \mathit{int} \models (\mathit{int} \triangleright \alpha)}{E, \Gamma \vdash (i : (\mathit{int} \triangleright \alpha)) : \alpha}$$

4.5 Synthèse des types

Nous reprenons l'approche de HM(X) (Pottier & Rémy, 2005) pour traiter la synthèse des types de MLGX. Il s'agit de réduire ce problème à celui de la résolution de contraintes de type. La figure 4.6 décrit la génération des contraintes de typage pour un terme de MLGX. On définit $(\!| E \vdash t : \tau |\!)$ qui est la contrainte nécessaire et suffisante pour que le terme t ait le type τ sous le système d'équations E.

La plupart de ces règles se trouvent déjà dans l'inférence de types standard de ML (Pottier & Rémy, 2005). Nous allons décrire les règles spécifiques à MLGX, c'est-à-dire celles associées aux coercions, aux *pattern matchings* et aux branches.

La règle G-Coerce vérifie d'abord que E implique la validité de la coercion $\exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)$. Cette vérification est simple à mettre en œuvre : étant donné que $\overline{\gamma} \# \mathbf{ftv}(E)$, il est équivalent de calculer l'unificateur le plus général de E et de vérifier que c'est aussi un unificateur pour $\tau_1 = \tau_2$. Si cette vérification échoue, il n'y a aucune chance que le programme soit bien typé et on peut arrêter la synthèse des types dès maintenant. Si cette vérification réussit alors la contrainte de typage associée à la coercion exprime le fait que, dorénavant, le terme t doit avoir le type τ_1 mais le type visible de l'extérieur est τ_2. La contrainte obtenue est alors identique à celle engendrée pour l'application d'une fonction de type $\forall\overline{\gamma}.\tau_1 \to \tau_2$ au terme t.

La règle G-Case est simple. L'annotation de type θ est véhiculée dans chaque branche et dans le type attendu τ pour la valeur analysée. C'est la variable flexible γ qui dénote le type de cette dernière.

La règle G-Clause paraphrase X-Clause. Elle s'appuie sur le schéma de type de K. La contrainte commence par introduire existentiellement le vecteur $\overline{\alpha}$ qui correspond aux paramètres ordinaires du constructeur de type algébrique. Ensuite, on trouve une conjonction.

Le premier terme $\exists\overline{\gamma}'_2.(\tau' = \varepsilon\,\overline{\alpha}\,\overline{\gamma}'_2)$ détermine des valeurs appropriées pour les variables de type $\overline{\alpha}$ en les égalisant avec les paramètres ordinaires de τ', le type de la valeur analysée. Les variables de type $\overline{\gamma}'_2$ n'apparaissent pas : elles servent uniquement à oublier les paramètres généralisés. Dans le second terme, le système d'équations E est augmentée avec l'équation $\overline{\tau}'_2 = \overline{\tau}$, obtenue en confrontant l'annotation de type avec le schéma de type de K. On retrouve ici le même processus de formation des équations que lors de la vérification des types. Les variables de type qui apparaissent au sein de ces équations sont un sous-ensemble de $\overline{\beta}\overline{\gamma}$. Ces variables de type sont universellement quantifiées en tête de la contrainte ce qui permet de maintenir l'invariant que les types intervenant dans le système d'équations sont rigides. Cet invariant est nécessaire pour s'assurer la décidabilité efficace de l'implication entre les égalités de type.L'environnement de type Γ est étendu avec les liaisons des variables libres du motif et, enfin, on engendre la contrainte correspondant au bon typage du corps de la branche t en lui demandant d'avoir le type τ.

Il est intéressant d'observer comment les règles de génération de contraintes se simplifient si on les spécialise aux types algébriques ordinaires ou aux types algébriques purement généralisés.

Si on suppose que ε est un type algébrique ordinaire, on a

$$\begin{aligned}&(\!| E \vdash (K\,\overline{\beta}\,x_1 \ldots x_n : \bot).t : \tau' \to \tau |\!) = \\ &\qquad \exists\overline{\alpha}.((\tau' = \varepsilon\,\overline{\alpha}) \wedge \forall\overline{\beta}.\mathsf{def}\, x_1 : \tau_1; \ldots; x_n : \tau_n \,\mathsf{in}\, (\!| E \vdash t : \tau |\!)) \\ &\text{si } K :: \forall\overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\alpha}\end{aligned}$$

ce qui correspond à la règle de génération de contraintes de types pour les types algébriques iso-existentiels.

Si on suppose que ε est un type algébrique purement généralisé, on a

$$\begin{aligned}&(\!| E \vdash (K\,\overline{\beta}\,x_1 \ldots x_n : \exists\overline{\gamma}.\varepsilon\,\overline{\tau}'_2).t : \tau' \to \tau |\!) = \\ &\qquad \forall\overline{\beta}\overline{\gamma}.\mathsf{def}\, x_1 : \tau_1; \ldots; x_n : \tau_n \,\mathsf{in}\, (\!| E \wedge \overline{\tau}'_2 = \overline{\tau} \vdash t : \tau |\!) \\ &\text{si } K :: \forall\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\tau}\end{aligned}$$

ce qui correspond à une formulation proche de celle de l'article de François Pottier et Vincent Simonet (Pottier & Simonet, 2003) car elle fait apparaître une implication de la forme « $E \vdash \ldots$ » qui a la particularité d'avoir un membre gauche en forme résolue.

La preuve de correction et complétude de la génération des contraintes de type pour ML (Pottier & Rémy, 2005) a déjà été faite. L'inférence de types pour MLGX ne pose pas de difficultés supplémentaires. En effet, la vérification des coercions et la détermination du système d'équations E n'influent ni sur la génération ni sur la résolution des contraintes de types. La conception de MLGX a été guidée par cette séparation entre la vérification des GADT d'une part et l'inférence de types pour la partie ML d'un autre part. D'un point de vue formel, il manque dans la preuve existante (Pottier & Rémy, 2005), la prise en compte des annotations de type mais cette fonctionnalité est admise par tous comme ne posant pas de problème.

G-Var
$(\!| E \vdash x : \tau |\!) =$
$\quad x \preceq \tau$

G-Lam
$(\!| E \vdash \lambda(x : \theta).t : \tau |\!) =$
$\quad \exists\gamma_1\gamma_2.(\theta \preceq \gamma_1 \wedge \gamma_1 \to \gamma_2 = \tau \wedge \mathsf{def}\ x : \gamma_1\ \mathsf{in}\ (\!| E \vdash t : \gamma_2 |\!))$

G-App
$(\!| E \vdash t_1\ t_2 : \tau |\!) =$
$\quad \exists\gamma.((\!| E \vdash t_1 : \gamma \to \tau |\!) \wedge (\!| E \vdash t_2 : \gamma |\!))$

G-Fix
$(\!| E \vdash \mu(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau').t : \tau |\!) =$
$\quad \exists\overline{\gamma}.(\mathsf{def}\ (x : \forall\overline{\alpha}.\tau')\ \mathsf{in}\ (\!| E \vdash t : \tau |\!) \wedge \exists\overline{\alpha}.(\tau' = \tau))$

G-Let
$(\!| E \vdash \mathsf{let}\, x\ =\ t_1\ \mathsf{in}\ t_2 : \tau |\!) =$
$\quad \mathsf{def}\ x : \forall\alpha[(\!| E \vdash t_1 : \alpha |\!)].\alpha\ \mathsf{in}\ (\!| E \vdash t_2 : \tau |\!)$

G-Forall
$(\!| E \vdash \forall\overline{\alpha}.t : \tau |\!) =$
$\quad \forall\overline{\alpha}.\exists\gamma.(\!| E \vdash t : \gamma |\!) \wedge \exists\overline{\alpha}.(\!| E \vdash t : \gamma |\!)$

G-Annot
$(\!| E \vdash (t : \exists\overline{\gamma}.\tau') : \tau |\!) =$
$\quad \exists\overline{\gamma}.\tau' = \tau \wedge (\!| E \vdash t : \tau |\!)$

G-Coerce
$(\!| E \vdash (t : \exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)) : \tau |\!) =$
$\quad \exists\overline{\gamma}.((\!| E \vdash t : \tau_1 |\!) \wedge \tau_2 = \tau)$
si $E \models \forall\overline{\gamma}.\tau_1 = \tau_2$

G-Case
$(\!| E \vdash \mathsf{match}\ (t : \theta)\ \mathsf{with}\ p_1.t_1 \ldots p_n.t_n : \tau |\!) =$
$\quad \exists\gamma.((\!| E \vdash (t : \theta) : \gamma |\!) \wedge \bigwedge_i (\!| E \vdash (p_i : \theta).t_i : \gamma \to \tau |\!))$

G-Clause
$(\!| E \vdash (K\,\overline{\beta}\ x_1 \ldots x_n : \exists\overline{\gamma}.\varepsilon \star \overline{\tau}'_2).t : \tau' \to \tau |\!) =$
$\quad \exists\overline{\alpha}.(\exists\overline{\gamma}'_2.(\tau' = \varepsilon\,\overline{\alpha}\,\overline{\gamma}'_2) \wedge \forall\overline{\beta}\overline{\gamma}.\mathsf{def}\ x_1 : \tau_1; \ldots; x_n : \tau_n\ \mathsf{in}\ (\!| E \wedge \overline{\tau}'_2 = \overline{\tau} \vdash t : \tau |\!))$
si $K :: \forall\overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\alpha}\,\overline{\tau}$

FIG. 4.6: Génération des contraintes de typage spécifiques à MLGX.

CHAPITRE CINQ

MLGI

Dans ce chapitre, on s'intéresse à MLGI, un langage contenant ML et des types algébriques généralisés. Contrairement à MLGX, MLGI permet une utilisation des GADT dans un style implicite ce qui en fait un bon candidat comme langage de surface. La section 5.1 présente rapidement la syntaxe qui diffère peu et la sémantique qui ne diffère pas du tout de celles de MLGX. Le système de type occupe la section 5.2. On le compare au système défini par Hongwei Xi (Xi *et al*, 2003) dans la section 5.4. Les problèmes liés à la synthèse des types sont abordés de manière informelle dans la section 5.5.

5.1 Syntaxe et sémantique

La figure 5.1 spécifie la syntaxe de MLGI. Elle rappelle très fortement la figure 4.1. En effet, les syntaxes de MLGI et de MLGX diffèrent uniquement sur la construction match qui doit impérativement être annotée dans MLGX.

On peut voir tout programme de MLGX comme un programme de MLGI et réciproquement. Il suffit d'une part de remarquer que tout *pattern matching* annoté d'un programme MLGX est la composition d'un *pattern matching* et d'une annotation de type dans MLGI. Il suffit d'autre part de considérer que, dans un programme MLGI, un *pattern matching* non annoté peut être vu comme un *pattern matching* de MLGX annoté par $\exists\gamma.\gamma$.

Dans MLGI, $\lambda x.t$ est un sucre syntaxique pour $\lambda(x : \exists\gamma.\gamma).t$. Toutes les constructions classiques de ML se retrouvent donc dans MLGI.

Comme tout programme MLGX et ML peut être écrit dans MLGI *modulo* quelques sucres syntaxiques, MLGI est un trait d'union entre ces deux langages.

La sémantique de MLGI est la même que celle de MLGX, nous ne la rappelons donc pas.

5.2 Système de type

Le système de type de MLGI est défini par les règles de la figure 5.2. La plupart des règles de MLGX sont réutilisées sans modification : les règles qui ont trait au noyau de ML sont inchangées, les règles Clause et Case sont modifiées et une règle Conv est rajoutée.

▶ **Case** La règle Case de MLGI est plus standard que celle de MLGX grâce à l'absence d'annotation de type.

▶ **Clause** La règle Clause de MLGI s'appuie sur le type exact de la branche et non pas sur une annotation de type pour déterminer le fragment $(\overline{\beta}, E', \Gamma')$. La simplicité apparente de la règle de MLGI

Types	$\tau ::=$	
Variable de type		α
Type des fonctions		$\mid \tau \to \tau$
Type algébrique		$\mid \varepsilon\, \overline{\tau}\, \overline{\tau}$
Schemas de type	$\sigma ::= \forall \overline{\alpha}.\tau$	
Annotations de type simple	$\theta ::= \exists \overline{\gamma}.\tau$	
Annotations de type polymorphe	$\varsigma ::= \exists \overline{\gamma}.\sigma$	
Coercions de type	$\kappa ::= \exists \overline{\gamma}.(\tau \triangleright \tau)$	
Termes	$t ::=$	
Variable		x
Fonction		$\mid \lambda(x : \theta).t$
Application de fonction		$\mid t\ (t)$
Definition locale		$\mid \mathsf{let}\, x\ =\ t\ \mathsf{in}\ t$
Point fixe		$\mid \mu(x : \varsigma).t$
Application de constructeur de données		$\mid K\ ((t, \ldots, t))$
Analyse par cas		$\mid \mathsf{match}\, t\, \mathsf{with}\, \overline{c}$
Introduction de variable de type		$\mid \forall \overline{\alpha}.t$
Annotation de type		$\mid (t : \theta)$
Coercion de type		$\mid (t : \kappa)$
Branches	$c ::= p.t$	
Motifs	$p ::= K\, \overline{\beta}\, \overline{x}$	
Système de d'équation entre types	$E ::= \mathsf{true} \mid \tau = \tau \mid E \wedge E$	

FIG. 5.1: Types et termes de MLGI.

par rapport à celle de MLGX n'est qu'une illusion : désormais l'ensemble des équations acquises par l'analyse d'une valeur dépend de *toute la dérivation de typage du programme*. Dans MLGI, la détermination de E requiert donc une analyse globale contrairement à l'observation locale de l'annotation qui suffit dans MLGX.

▶ **Conv** La règle CONV introduit l'utilisation implicite, non dirigée par la syntaxe, des égalités induites par le système d'équations E. Pour cela, on permet au programmeur de travailler *modulo* E ce qui se réalise par la possibilité de modifier à tout moment le type τ_1 d'un terme en un type τ_2 équivalent.

5.3 Propriétés méta-théoriques

Nous aurons besoin d'une propriété méta-théorique sur les dérivations de typage de MLGI pour effectuer les preuves du chapitre suivant.

$$\textsc{Var}\ \frac{(x:\sigma)\in\Gamma}{E,\Gamma\vdash x:\sigma} \qquad \textsc{Lam}\ \frac{E,(\Gamma;x:\tau_1)\vdash t:\tau_2 \qquad \theta\preceq\tau_1}{E,\Gamma\vdash\lambda(x:\theta).t:\tau_1\to\tau_2} \qquad \textsc{App}\ \frac{E,\Gamma\vdash t_1:\tau_1\to\tau_2 \quad E,\Gamma\vdash t_2:\tau_1}{E,\Gamma\vdash t_1\ (t_2):\tau_2}$$

$$\textsc{Let}\ \frac{E,\Gamma\vdash t_1:\sigma \qquad E,(\Gamma;x:\sigma)\vdash t_2:\tau}{E,\Gamma\vdash \mathsf{let}\, x\ =\ t_1\ \mathsf{in}\ t_2:\tau} \qquad \textsc{Fix}\ \frac{E,(\Gamma;x:\sigma)\vdash t:\sigma \qquad \varsigma\preceq\sigma}{E,\Gamma\vdash\mu(x:\varsigma).t:\sigma}$$

$$\textsc{Cstr}\ \frac{K\preceq\tau_1\times\ldots\times\tau_n\to\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \quad \forall i\quad E,\Gamma\vdash t_i:\tau_i}{E,\Gamma\vdash K\,(t_1\ldots t_n):\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2} \qquad \textsc{Case}\ \frac{E,\Gamma\vdash t:\tau_1 \quad \forall i\quad E,\Gamma\vdash c_i:\tau_1\to\tau_2}{E,\Gamma\vdash \mathsf{match}\,t\,\mathsf{with}\ c_1\ldots c_n:\tau_2} \qquad \textsc{Forall}\ \frac{E,\Gamma\vdash t:\tau \qquad \overline{\alpha}\,\#\,\mathbf{ftv}\,(E,\Gamma)}{E,\Gamma\vdash\forall\overline{\alpha}.t:\forall\overline{\alpha}.\tau}$$

$$\textsc{Annot}\ \frac{E,\Gamma\vdash t:\tau \qquad \theta\preceq\tau}{E,\Gamma\vdash(t:\theta):\tau} \qquad \textsc{Coerce}\ \frac{E,\Gamma\vdash t:\tau_1 \qquad \kappa\preceq(\tau_1\triangleright\tau_2) \qquad E\models\kappa}{E,\Gamma\vdash(t:\kappa):\tau_2} \qquad \textsc{Conv}\ \frac{E,\Gamma\vdash t:\tau_1 \qquad E\models\tau_1=\tau_2}{E,\Gamma\vdash t:\tau_2}$$

$$\textsc{Gen}\ \frac{E,\Gamma\vdash t:\tau \qquad \overline{\alpha}\,\#\,\mathbf{ftv}\,(E,\Gamma,t)}{E,\Gamma\vdash t:\forall\overline{\alpha}.\tau} \qquad \textsc{Inst}\ \frac{E,\Gamma\vdash t:\sigma \qquad \sigma\preceq\tau}{E,\Gamma\vdash t:\tau}$$

$$\textsc{Clause}\ \frac{p:\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2\vdash(\overline{\beta},E',\Gamma') \qquad E\wedge E',\Gamma\Gamma'\vdash t:\tau_2 \quad \overline{\beta}\,\#\,\mathbf{ftv}\,(E,\Gamma,\tau_2)}{E,\Gamma\vdash p.t:\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2\to\tau_2}$$

$$\textsc{Pat}\ \frac{K\preceq\forall\overline{\beta}.\tau_1\times\ldots\times\tau_n\to\varepsilon\,\overline{\tau}_1\,\overline{\tau} \qquad \overline{\beta}\,\#\,\mathbf{ftv}\,(\overline{\tau}_1,\overline{\tau}_2)}{K\,\overline{\beta}\,x_1\ldots x_n:\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2\vdash(\overline{\beta},\overline{\tau}_2=\overline{\tau},(x_1:\tau_1;\ldots;x_n:\tau_n))}$$

FIG. 5.2: **ML** avec GADT dans un style implicite.

Lemme 5.3.1 (Affaiblissement des égalités dans MLGI)
Si $E,\Gamma\vdash t:\sigma$ *et* $E'\models E$ *alors* $E',\Gamma\vdash t:\sigma$. ◇

Preuve. La preuve se fait par induction sur la dérivation de typage du terme t. Il suffit de remarquer que les seules utilisations du système d'équations E sont les prémisses de la règle CONV de la forme $E\models\tau_1=\tau_2$ et les prémisses de la règle COERCE de la forme $E\models\kappa$. L'hypothèse $E'\models E$ implique qu'on peut remplacer le système d'équations E par le système d'équations E' sans mettre en jeu la validité de ces jugements. Dès lors, la substitution $[E\mapsto E']$ appliquée à la dérivation de typage du jugement $E,\Gamma\vdash t:\sigma$ donne une dérivation de typage valide pour le jugement $E',\Gamma\vdash t:\sigma$. □

5.4 Comparaison avec le système de Hongwei Xi

Dans son article introduisant les GADT, Hongwei Xi (Xi *et al*, 2003) fournit deux systèmes de type. Le premier est un λ-calcul polymorphiquement typé d'ordre 2 et explicitement typé. Il définit la vérification des types dans ce système. Les règles de MLGI sont quasi-identiques aux siennes si on met de côté les règles GEN et INST ainsi que la stratification des types en schémas de type et types monomorphes propre à ML.

Le second système de type est un langage externe où la plupart des instanciations de types et généralisations sont élaborés pour rendre la programmation plus simple. Bien que ce système soit proche de ML, il diffère de MLGI et MLGX qui sont des extensions conservatives de ML alors que certains programmes bien formés de ML ne sont pas acceptés par le système de Hongwei Xi (Xi *et al*, 2003).

5.5 Quelques mots sur la synthèse des types dans MLGI

Comme nous l'avons suggéré dans l'introduction, la synthèse de type dans un système implicitement typé comme MLGI est complexe en présence de GADT. En effet, la façon la plus canonique de formuler les contraintes de typage dans MLGI est d'introduire des implications dans la syntaxe des contraintes (Pottier & Simonet, 2003, Stuckey & Sulzmann, 2005) :

$$
\begin{array}{rl}
\textbf{Contraintes} & C ::= \\
\textit{Vérité} & \mid \mathsf{true} \\
\textit{Egalité} & \mid \tau = \tau \\
\textit{Conjonction} & \mid C \wedge C \\
\textit{Quantification existentielle} & \mid \exists \overline{\gamma}.C \\
\textit{Quantification universelle} & \mid \forall \overline{\alpha}.C \\
\textit{Instanciation} & \mid x \preceq \tau \\
\textit{Définition} & \mid \mathsf{def}\ x : \forall \overline{\alpha}[C].\tau\ \mathsf{in}\ C \\
\textit{Implication} & \mid C \Rightarrow C
\end{array}
$$

Les contraintes de types générées pour la construction match et les branches deviennent :

$$
\begin{array}{lcl}
(\!|\, \mathsf{match}\ t\ \mathsf{with}\ c_1 \ldots c_n : \tau \,|\!) & = & \exists \gamma.(\!|\, t : \gamma \,|\!) \wedge \bigwedge_{i=1}^{n} (\!|\, c_i : \gamma \to \tau \,|\!) \\
(\!|\, K\ \overline{\beta}\ x_1 \ldots x_n.t : \gamma \to \tau \,|\!) & = & \forall \overline{\alpha}.\forall \overline{\beta}.\gamma = \varepsilon\ \overline{\alpha}\ \overline{\tau}' \Rightarrow \mathsf{def}\ x_1 : \tau_1; \ldots; x_n : \tau_n\ \mathsf{in}\ (\!|\, t : \tau \,|\!) \\
 & & \text{avec } K :: \forall \overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\ \overline{\alpha}\ \overline{\tau}'
\end{array}
$$

Plusieurs difficultés justifient notre volonté de ne pas générer des contraintes de cette forme.

Tout d'abord, il est bien connu que la résolution d'une implication induit une combinatoire car elle peut se ré-exprimer sous la forme d'une disjonction. Dans notre cas, du point de vue du programme, une implication a un membre gauche faux lorsque la branche correspondante dans le programme représente un cas inadapté au type de la valeur analysée.

Une première amélioration consiste à savoir décider si ce membre gauche est faux ou non à l'aide une observation locale. La méthode la plus simple pour obtenir cette localité correspond à la première restriction de MLGX : si la variable γ est remplacée par un type rigide, ce qui correspond à forcer les valeurs analysées à être annotées par leur type, alors on sait décider si le membre gauche de l'implication est vrai ou faux.

Ensuite, il faudrait pouvoir supprimer les implications en traduisant les égalités obtenues à l'aide de contraintes. Or, cela n'est pas si simple puisque les égalités ne sont valides que localement, dans les membres droits des implications. On ne peut évidemment pas traduire les implications en conjonctions. Il n'est pas non plus correct d'utiliser la méthode de Simon Peyton-Jones qui revient à calculer le *mgu* de l'équation et à l'appliquer à la contrainte du membre droit. En effet, la contrainte suivante :

$$\forall \alpha.\exists \gamma.\mathsf{def}\ x : \gamma\ \mathsf{in}\ (\alpha = \mathit{int} \Rightarrow x \preceq \alpha) \wedge x \preceq \alpha$$

n'est pas équivalente à la contrainte :

$$\forall \alpha.\exists \gamma.\mathsf{def}\ x : \gamma\ \mathsf{in}\ x \preceq \mathit{int} \wedge x \preceq \alpha$$

puisque la première est satisfiable et la seconde ne l'est pas. La dernière méthode consisterait à effectuer une élévation de la sorte (Miller, 1992) de la variable de type γ de manière à faire commuter le $\forall$ et le $\exists$. On réécrirait alors la contrainte en :

$$\exists\gamma.\forall\alpha.\mathsf{def}\ x : \gamma\ \alpha\ \mathsf{in}\ \gamma\ int = int \wedge \gamma\ \alpha = \alpha$$

On peut imaginer utiliser le fragment décidable de l'unification d'ordre supérieur sur les *patterns* (Dowek *et al*, 1996) pour résoudre de telles contraintes car les fonctions des types dans les types à inférer ici seraient de la forme : $\Lambda\overline{\alpha}.\tau$ (la forme des schémas de type de ML).

L'exemple de la fonction *cast* présentée plus tôt illustre que la perte de la principalité est toujours présent même en augmentant ainsi le domaine des solutions des contraintes. En effet, pour la fonction *cast*, la contrainte ressemblerait à :

$$\exists\gamma.\forall\alpha\beta.\mathsf{def}\ x : \gamma\ \alpha\ \beta\ \mathsf{in}\ \gamma\ \alpha\ \beta = \alpha = \beta$$

Dans ce cas, deux affectations sont possibles pour la variable γ : $\Lambda\alpha\beta.\alpha$ et $\Lambda\alpha\beta.\beta$. Aucune de ces solutions n'est préférable à l'autre. Il faudrait donc rejeter les programmes engendrant des contraintes dont les solutions sont de cette forme.

Le système hypothétique obtenu si on suivait cette voie nous semble assez complexe et nous avons préféré expliciter l'utilisation des équations à l'aide de coercion pour utiliser l'inférence de types dans le style de Hindley-Milner dont on connaît l'efficacité en pratique.

CHAPITRE SIX

Correction de MLGX et MLGI

La preuve de *type soundness* de ML muni de GADT n'est pas une contribution de cette thèse (Xi *et al*, 2003). Dans ce chapitre, on s'appuie sur ce résultat pour montrer la correction de MLGX ainsi que des propriétés liant MLGX et MLGI.

6.1 Correction de MLGI vis-à-vis de la sémantique

La preuve de correction du système de type de MLGI vis-à-vis de la sémantique a déjà été faite par Hongwei Xi (Xi *et al*, 2003). En effet, les différences entre MLGI et le système de Xi sont suffisamment superficielles pour que la preuve soit inchangée dans son ensemble. Il ne sert donc à rien de la répéter ici. Nous rappelons tout de même les deux énoncés classiques qui s'appuient sur la méthode syntaxique introduite par Andrew Wright et Matthias Felleison (1994).

Théorème 6.1.1 (Progression de l'évaluation dans MLGI)
Si $E, \Gamma \vdash t : \sigma$ *dans MLGI alors soit le terme* t *est une valeur, soit il existe un terme* t' *tel que* $t \rightarrow t'$.◇

Théorème 6.1.2 (Préservation du typage dans MLGI)
Si $E, \Gamma \vdash t : \sigma$ *dans MLGI et* $t \rightarrow t'$ *alors* $E, \Gamma \vdash t' : \sigma$ *dans MLGI.* ◇

Théorème 6.1.3 (Correction de MLGI)
Si $E, \Gamma \vdash t : \sigma$ *dans MLGI alors l'évaluation du terme* t *ne peut pas atteindre un état bloqué.* ◇

François Pottier et Vincent Simonet (2003) ont aussi prouvé la correction des GADT mais en prouvant aussi que le test d'exhaustivité plus précis permis par les GADT est correct. Nous passons ces détails sous silence car ils n'entrent pas en jeu pour traiter les problèmes d'inférence de types.

6.2 Correction de MLGX par rapport à MLGI

MLGX est une contribution de cette thèse et sa preuve de correction doit donc être faite. Pour cela, on se contente de montrer que le bon typage dans MLGX implique le bon typage dans MLGI.

Théorème 6.2.1 (Correction de MLGX par rapport à MLGI)
Si $E, \Gamma \vdash t : \sigma$ *est valide dans MLGX, alors il est valide dans MLGI aussi.* ◇

Preuve. La preuve se fait par induction structurelle sur les dérivations de type de MLGX avec un cas par règle de typage. Pour toutes les règles qui sont communes à MLGI et MLGX, le résultat est immédiat. Il reste donc deux cas.

▷ *Cas* « X-Case ». Il est clair que cette règle traduit une combinaison de la règle Annot et de la règle Case. De plus, la première prémisse assure que $\theta \preceq \tau_1$, une hypothèse que nous exploitons pour l'examen de règle X-Clause qui suit.

▷ *Cas* « X-Clause ». La conclusion de la règle est

$$E, \Gamma \vdash (p : \exists \overline{\gamma}.\varepsilon\, \overline{\tau}'_1\, \overline{\tau}'_2).t : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \to \tau_2 \tag{1}$$

Conformément à notre dernière remarque, nous pouvons aussi supposer que

$$(\exists \overline{\gamma}.\varepsilon\, \overline{\tau}'_1\, \overline{\tau}'_2) \preceq \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \tag{2}$$

Nous pouvons aussi supposer sans perte de généralité que $\overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}_2)$ (**3**) et $\overline{\gamma} \# \mathbf{ftv}\,(\overline{\beta}, \overline{\tau}_2)$ (**4**). La prémisse de la règle X-Clause, une hypothèse de la règle X-Pat,est

$$p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}'_2 \vdash (\overline{\beta}, E', \Gamma')$$

où E' est de la forme $\overline{\tau}'_2 = \overline{\tau}$. En utilisant $\overline{\tau}_2$ à la place de $\overline{\tau}'_2$ dans la règle Pat et en exploitant (3), nous obtenons :

$$p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, E'', \Gamma') \tag{5}$$

où E'' est $\overline{\tau}_2 = \overline{\tau}$. Maintenant, (2) implique $(\exists \overline{\gamma}.\overline{\tau}'_2) \preceq \overline{\tau}_2$, qui, grâce à (4), peut être écrite $\mathsf{true} \models \exists \overline{\gamma}.(\overline{\tau}'_2 = \overline{\tau}_2)$. De plus, (4) implique $\overline{\gamma} \# \mathbf{ftv}\,(\overline{\tau})$, donc nous avons

$$\overline{\tau}_2 = \overline{\tau} \models \overline{\tau}_2 = \overline{\tau} \wedge \exists \overline{\gamma}.(\overline{\tau}'_2 = \overline{\tau}_2) \models \exists \overline{\gamma}.(\overline{\tau}'_2 = \overline{\tau})$$

c'est-à-dire $E'' \models \exists \overline{\gamma}.E'$.

Comme le montre la dernière prémisse de X-Clause, les variables de type $\overline{\gamma}$ sont fraîches pour le système d'équations E, ce qui implique :

$$E \wedge E'' \models \exists \overline{\gamma}.(E \wedge E') \tag{6}$$

En appliquant l'hypothèse de récurrence à la seconde prémisse de règle X-Clause, on montre que

$$E \wedge E', \Gamma\Gamma' \vdash t : \tau_2 \tag{7}$$

est valide dans MLGI.

Par la dernière prémisse de la règle X-Clause et par (4), nous avons $\overline{\gamma} \# \mathbf{ftv}\,(\Gamma, \Gamma', t, \tau_2)$ (**8**).

En appliquant le lemme 5.3.1 à (7), (6), et (8), on obtient :

$$E \wedge E'', \Gamma\Gamma' \vdash t : \tau_2 \tag{9}$$

Nous pouvons conclure en construisant dans MLGI l'instance de la règle Clause où les deux premières prémisses sont (5) et (9), et dont la conclusion est le but :

$$E, \Gamma \vdash p.t : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \to \tau_2 \qquad \square$$

Comme MLGI et MLGX possèdent la même sémantique opérationnelle, on déduit du théorème précédent et du théorème de correction de MLGI vis-à-vis de la sémantique, le théorème de correction de MLGX vis-à-vis de la sémantique :

Théorème 6.2.2 (Correction de MLGX vis-à-vis de la sémantique)
Si $E, \Gamma \vdash t : \sigma$ est valide dans MLGX alors l'évaluation du terme t ne peut pas atteindre un état bloqué. ◇

6.3 Complétude avec assistance de MLGX par rapport à MLGI

MLGX accepte autant de programmes que MLGI *modulo* l'explicitation de l'utilisation des GADT ce qui en fait une cible pertinente pour l'inférence stratifiée que nous allons présenter plus loin.

Pour montrer ce résultat, on définit formellement ce qu'on entend par « *modulo* l'explicitation de l'utilisation des GADT ». Il s'agit de redéfinir l'égalité entre programmes en ignorant les annotations de coercions et les annotations de types. Ainsi, deux programmes équivalents ont la même sémantique opérationnelle mais leurs dérivations de typage diffèrent éventuellement.

Définition 6.3.1
*Définissons l'*équivalence modulo annotations*, écrite* $\equiv$*, comme la fermeture réflexive et par congruence de l'ensemble des axiomes suivants :*

$$\frac{t \equiv t' \qquad \overline{\alpha} \# \mathbf{ftv}\,(t)}{t \equiv \forall \overline{\alpha}.t'} \qquad\qquad \frac{t \equiv t'}{t \equiv (t' : \theta)} \qquad\qquad \frac{t \equiv t'}{t \equiv (t' : \kappa)}$$

Le résultat suivant montre qu'on pourra toujours rajouter suffisamment d'annotations de types et de coercions pour qu'un programme bien typé de MLGI soit accepté par MLGX. On obtient ainsi en passant une certaine garantie de complétude pour l'inférence stratifiée : un programmeur qui rencontre des difficultés à faire accepter un programme bien typé dans MLGI à cause de l'incomplétude de l'inférence locale pourra toujours rajouter des annotations pour le faire accepter dans MLGX.

Théorème 6.3.2 (Complétude avec assistance de MLGX)
Si $E, \Gamma \vdash t : \sigma$ *est valide dans MLGI, alors il existe un terme* t' *tel que* $t \equiv t'$ *et* $E, \Gamma \vdash t' : \sigma$ *est valide dans MLGX.* ◇

Preuve. La preuve est une induction structurelle sur les dérivations de typage de MLGI avec un cas par règle de typage. Pour toutes les règles qui sont communes à MLGI et MLGX, exceptée la règle Gen, le résultat est immédiat. Nous nous attaquons maintenant aux cas restants.

▷ *Cas* « Gen ». La conclusion de la règle est $E, \Gamma \vdash t : \forall \overline{\alpha}.\tau$. Ses prémisses sont

$$E, \Gamma \vdash t : \tau \tag{1}$$

et

$$\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma, t) \tag{2}$$

En appliquant l'hypothèse d'induction à (1), on obtient un terme t' tel que $t \equiv t'$ (**3**) est valide et

$$E, \Gamma \vdash t' : \tau \tag{4}$$

est valide dans MLGX.

De plus, (2) implique

$$\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma) \tag{5}$$

Appliquer la règle X-Forall à (4) et (5) fournit

$$E, \Gamma \vdash \forall \overline{\alpha}.t' : \forall \overline{\alpha}.\tau$$

dans MLGX. En outre, par (3) et (2), $t \equiv \forall \overline{\alpha}.t'$ est donc valide.

▷ *Cas* « Conv ». La conclusion de la règle est

$$E, \Gamma \vdash t : \tau_2$$

Ses prémisses sont

$$E, \Gamma \vdash t : \tau_1 \tag{1}$$

et

$$E \models \tau_1 = \tau_2 \tag{2}$$

Appliquer l'hypothèse d'induction à (1) fournit un terme t' tel que $t \equiv t'$ (**3**) est valide et

$$E, \Gamma \vdash t' : \tau_1 \tag{4}$$

est valide dans MLGX. Appliquer la règle X-Coerce à (4) et (2) donne

$$E, \Gamma \vdash (t' : (\tau_1 \triangleright \tau_2)) : \tau_2$$

valide dans MLGX. De plus, par (3), $t \equiv (t' : (\tau_1 \triangleright \tau_2))$ est valide.

▷ *Cas* « Case ». La conclusion de la règle est :

$$E, \Gamma \vdash \mathsf{match}\, t\, \mathsf{with}\, p_1.t_1 \ldots p_n.t_n : \tau_2$$

Ses prémisses sont

$$E, \Gamma \vdash t : \tau_1 \tag{1}$$

et, pour tout $i \in \{1, \ldots, n\}$,

$$E, \Gamma \vdash p_i.t_i : \tau_1 \to \tau_2 \tag{2}$$

Appliquer l'hypothèse d'induction à (1) fournit un terme t' tel que $t \equiv t'$ (**3**) est valide et

$$E, \Gamma \vdash t' : \tau_1 \tag{4}$$

est valide dans MLGX.

Supposons que l'entier n est non nul. Alors, τ_1 doit être de la forme $\varepsilon\, \overline{\tau}_1\, \overline{\tau}_2$. Soit θ valant $\exists \overline{\gamma}_1.\varepsilon\, \overline{\gamma}_1\, \overline{\tau}_2$, où les variables de type $\overline{\gamma}_1$ sont choisies fraîches pour $\overline{\tau}_2$. Il est clair que $\theta \preceq \tau_1$ est valide. En conséquence, (4) implique :

$$E, \Gamma \vdash (t' : \theta) : \tau_1 \tag{5}$$

est valide dans MLGX.

En examinant le jugement (2), qui est une conséquence de la règle Clause, en appliquant l'hypothèse d'induction, et en appliquant la règle X-Clause, on obtient les termes $t'_1 \ldots t'_n$ tels que, pour tout $i \in \{1, \ldots, n\}$, $t_i \equiv t'_i$ (**6**) est valide et

$$E, \Gamma \vdash (p_i : \theta).t'_i : \tau_1 \to \tau_2 \tag{7}$$

est valide dans MLGX.

Appliquer la règle X-Case à (5) et (7) montre que

$$E, \Gamma \vdash \mathsf{match}\, (t : \theta)\, \mathsf{with}\, p_1.t'_1 \ldots p_n.t'_n : \tau_2$$

est valide dans MLGX. En outre, par (3) et (6),

$$\mathsf{match}\, t\, \mathsf{with}\, p_1.t_1 \ldots p_i.t_i \equiv \mathsf{match}\, (t : \theta)\, \mathsf{with}\, p_1.t'_1 \ldots p_n.t'_n$$

est valide.

Il est bon de noter que l'annotation de type θ qui a été insérée ne contient aucune information à propos des paramètres de types ordinaires de ε : il contient seulement l'information concernant les paramètres de type généralisés $\overline{\tau}_2$. En particulier, quand ε est un type algébrique ordinaire, l'annotation de type θ a la forme $\exists \overline{\gamma}_1.\varepsilon\, \overline{\gamma}_1$ qui est équivalent à une absence d'annotations d'après nos conventions.

Si l'entier n est nul, notons θ pour $\exists \gamma.\gamma$. Alors, $\theta \preceq \tau_1$ est valide ce qui implique (5), comme plus haut. De plus, il n'y a pas de branches donc (7) est tout aussi valide. Le résultat suit de là. □

CHAPITRE SEPT

Semi-treillis des formes

Le prochain chapitre sera dédié à la spécification de deux algorithmes d'élaboration d'un programme de MLGI en un programme de MLGX. Cette élaboration prend la forme d'une propagation *prévisible* des annotations écrites par l'utilisateur. L'adjectif « prévisible » est une notion assez informelle et en donner une définition scientifique serait ardu. Nous choisissons donc de nous appuyer sur une conviction : la propagation est prévisible si elle s'effectue localement, c'est-à-dire des termes vers les sous-termes et réciproquement. Autrement dit, l'interaction entre les sous-termes s'effectue toujours de proche en proche et non à longue distance (par le biais d'une variable globale d'unification par exemple). De manière imagée, on peut dire que le programmeur doit pouvoir « suivre du doigt » la propagation des types pour que celle-ci soit prévisible.

Éviter des raisonnements à longue distance est particulièrement important en présence de GADT parce que le système d'équations E n'est pas uniforme en tout point du terme à typer. Cette variabilité du système d'équations E complique énormément les problèmes d'unification comme nous l'avons montré dans la section concernant la synthèse de type dans MLGI.

Or, l'algorithme $\mathcal{W}$, la résolution des contraintes de type ou l'inférence à la Hindley-Milner plus généralement utilisent des variables d'unification pour modéliser les types inconnus du programme. Ces variables peuvent s'échapper à travers l'environnement et donc provoquer des unifications à longue distance. Par ce seul fait, les variables globales d'unification ne sont pas adaptées.

En guise de substitut, nous introduisons dans ce chapitre un nouvel objet, que nous appelons « forme », qui permet d'utiliser cet outil puissant qu'est l'unification du premier ordre tout en maintenant l'aspect local de la contribution des annotations de type.

7.1 Présentation

Les formes sont définies par

$$s ::= \overline{\gamma}.\tau$$

où les variables de type $\overline{\gamma}$ sont liées dans le type τ. Les variables de type $\overline{\gamma}$ seront dites flexibles. Une variable de type flexible représente un type qui est soit *inconnu* (la forme $\gamma.\gamma \to \gamma$ peut alors représenter le type $int \to int$) ou une variable de type *polymorphe* (la forme $\gamma.\gamma \to \gamma$ décrit aussi le schéma de type $\forall\gamma.\gamma \to \gamma$). Les formes ne sont pas forcément closes. Les variables libres d'une forme seront dites rigides, c'est-à-dire liées universellement dans le programme.

Exemple 7.1.1
La forme $\gamma.\alpha \times \gamma$ décrit une paire dont la première composante a le type α, où la variable rigide α a été introduite par le programmeur, et dont la seconde composante a un type inconnu. ◇

$$
\begin{array}{lcl}
\bot & = & \gamma.\gamma \\
(\overline{\gamma}_1.\tau_1) \to (\overline{\gamma}_2.\tau_2) & = & \overline{\gamma}_1\overline{\gamma}_2.\tau_1 \to \tau_2 \\
 & & \overline{\gamma}_1 \# \mathbf{ftv}\,(\tau_2), \overline{\gamma}_2 \# \mathbf{ftv}\,(\tau_1) \\
\mathcal{D}(\bot) & = & \bot \\
\mathcal{D}(\overline{\gamma}.\tau_1 \to \star) & = & \overline{\gamma}.\tau_1 \\
\mathcal{C}(\bot) & = & \bot \\
\mathcal{C}(\overline{\gamma}.\star \to \tau_2) & = & \overline{\gamma}.\tau_2
\end{array}
$$

FIG. 7.1: Opérations élémentaires sur les formes.

Les formes sont syntaxiquement très proches des schémas de type. En effet, il suffit de rajouter un quantificateur $\forall$ devant une forme pour qu'elle soit semblable à un schéma de type. Néanmoins, nous établirons une distinction sémantique entre ces deux objets par le biais des règles d'élaboration du chapitre suivant.

Les formes partagent aussi de nombreux points communs avec les annotations de type. Par la suite, on convertira d'ailleurs souvent implicitement une annotation de type simple $\exists\overline{\gamma}.\tau$ en la forme $\overline{\gamma}.\tau$. De même, une annotation de type polymorphe $\exists\overline{\gamma}.\forall\overline{\alpha}.\tau$ peut être convertie en une forme mais nous le ferons explicitement.

La figure 7.1 introduit quelques opérations primitives sur les formes. La forme *bottom* $\gamma.\gamma$ sera notée $\bot$. Cette forme ne contient aucune information. À partir de deux formes quelconques s_1 et s_2, on peut construire une forme de fonction $s_1 \to s_2$. Réciproquement, à partir d'une forme s, on définit deux fonctions partielles, extrayant son domaine et son co-domaine, que nous noterons $\mathcal{D}(s)$ et $\mathcal{C}(s)$. Ces opérations sont définies sur les formes qui sont des flèches et sur $\bot$.

L'espace des formes est équipé d'une relation standard d'instanciation définie par l'axiome :

$$\frac{\overline{\gamma}_2 \# \mathbf{ftv}\,(\overline{\gamma}_1.\tau_1)}{\overline{\gamma}_1.\tau_1 \preceq \overline{\gamma}_2.[\overline{\gamma}_1 \mapsto \overline{\tau}_1]\tau_1}$$

Exemple 7.1.2
On a $(\gamma_1.\alpha \times \gamma_1) \preceq (\gamma_2.\alpha \times (\alpha \to \gamma_2))$ en appliquant la substitution $[\gamma_1 \mapsto (\alpha \to \gamma_2)]$. ◇

Les variables flexibles du membre gauche peuvent être instanciées dans le membre droit par des types arbitraires faisant éventuellement intervenir des variables rigides et flexibles. Les variables rigides du membre gauche sont préservées.

Ceci donne à l'espace des formes une structure algébrique très riche (Huet, 1976, chapitre 5) :

Théorème 7.1.3 (Huet)
L'espace des formes est un semi-treillis inférieur et bien fondé dont le plus petit élément est $\bot$. ◇

En d'autres termes, tout ensemble de formes non vide, fini ou infini, admet une plus grande borne inférieure. On note $s_1 \sqcap s_2$ cette borne et elle peut être calculée par anti-unification du premier ordre.

Ce résultat implique aussi que tout ensemble de formes qui admet une borne supérieure, admet une plus petit borne supérieure. La plus petite borne supérieure de deux formes s_1 et s_2, quand elle existe, sera notée $s_1 \sqcup s_2$ et peut être calculée par l'unification du premier ordre.

Exemple 7.1.4
Rappelons que $int \to \bot$ signifie $\gamma.int \to \gamma$. On a alors $(\gamma.\gamma \to \gamma) \sqcup (int \to \bot)$ qui est la forme $int \to int$. ◇

La différence entre les types avec variables flexibles utilisés par l'inférence de ML et les formes est la propriété que deux formes distinctes ne partagent jamais de variables flexibles. Ainsi, aucune

unification de « longue distance » ne peut avoir lieu : pour qu'une annotation soit prise en compte en un certain point du terme, il faut qu'elle soit propagée jusqu'à ce point. C'est sur cette propriété clé que va s'appuyer la localité des algorithmes d'élaboration.

L'ordre sur les formes peut aussi être défini en termes de contraintes :

Lemme 7.1.5
Soient $\overline{\gamma}_1 \# \mathbf{ftv}(\tau_2)$ *et* $\overline{\gamma}_2 \# \mathbf{ftv}(\overline{\gamma}_1.\tau_1)$. *Alors,* $\overline{\gamma}_1.\tau_1 \preceq \overline{\gamma}_2.\tau_2$ *est équivalent à* $\mathsf{true} \models \forall\overline{\gamma}_2.\exists\overline{\gamma}_1.\tau_1 = \tau_2$.◇

Preuve. $\overline{\gamma}_1.\tau_1 \preceq \overline{\gamma}_2.\tau_2$ est équivalent à l'existence de $\overline{\tau}_1$ tel que

$$\overline{\gamma}_2.\tau_2 \equiv \overline{\gamma}_2.[\overline{\gamma}_1 \mapsto \overline{\tau}_1]\tau_1$$

Ceci équivaut à dire que pour toute affectation arbitraire ϕ des $\overline{\gamma}_2$, on a $\phi \models \tau_2 = [\overline{\gamma}_1 \mapsto \overline{\tau}_1]\tau_1$ (**1**). Or, comme $\overline{\gamma}_1 \# \mathbf{ftv}(\tau_2)$, (1) et par les équivalences standard sur les constraintes (Pottier & Rémy, 2005) est équivalent à

$$\phi[\overline{\gamma}_1 \mapsto \overline{\tau}_1] \models \tau_2 = \tau_1 \qquad (\mathbf{2})$$

Les sémantiques de la quantification universelle et existentielle (Pottier & Rémy, 2005) permettent de conclure que (2) équivaut à

$$\mathsf{true} \models \forall\overline{\gamma}_2.\exists\overline{\gamma}_1.\tau_1 = \tau_2$$

□

Ce point de vue suggère un moyen de généraliser la définition de l'ordre pour le rendre relatif à un système d'équations E.

Définition 7.1.6
On écrit $E \models \overline{\gamma}_1.\tau_1 \preceq \overline{\gamma}_2.\tau_2$ *si, et seulement si* $E \models \forall\overline{\gamma}_2.\exists\overline{\gamma}_1.\tau_1 = \tau_2$ *est valide. On écrit* $E \models s_1 = s_2$ *quand* $E \models s_1 \preceq s_2$ *et* $E \models s_2 \preceq s_1$ *sont valides.* ◇

L'exemple suivant illustre comment cette définition prend effet :

Exemple 7.1.7
Notons s_1 pour la forme $\gamma_1.\alpha \times \gamma_1$ et s_2 pour $\gamma_2.int \times (\alpha \to \gamma_2)$. Alors, $s_1 \preceq s_2$ est invalide, parce que la variable de type rigide α ne peut pas être instanciée en *int*, mais

$$\alpha = int \models s_1 \preceq s_2$$

est valide. ◇

Le lemme suivant permet de lier l'instanciation *modulo* le système d'équations E à l'égalité *modulo* E.

Lemme 7.1.8
Si $E \models s \preceq \tau$ *alors il existe un type* τ' *tel que* $s \preceq \tau'$ *et* $E \models \tau = \tau'$ ◇

Preuve. Par définition, $E \models \overline{\gamma}.\tau' \preceq \tau$ est équivalent à $E \models \exists\overline{\gamma}.\tau' = \tau$. L'affectation ϕ des variables de type $\overline{\gamma}$ satisfiant cette dernière contrainte est telle que $\phi \models \tau' = \tau$ (**1**). Sans perte de généralité, on peut choisir $\overline{\gamma} \# \mathbf{ftv}(\tau)$. (1) est alors équivalent à $\models \phi(\tau') = \tau$. $\phi(\tau')$ est l'instance de la forme que nous cherchons. □

La relation d'instance *modulo* un système d'équations E est un pré-ordre.

Théorème 7.1.9
- *Pour toute forme* s, $E \models s \preceq s$.
- *Pour toutes formes* s_1, s_2 *et* s_3, *si* $E \models s_1 \preceq s_2$ *et* $E \models s_2 \preceq s_3$, *alors* $E \models s_1 \preceq s_3$ ◇

Preuve. Ces preuves ne posent aucune difficulté. □

On peut adapter cette définition d'instance *modulo* un système d'équations E pour y inclure l'instanciation d'une forme en un schéma de type.

Définition 7.1.10
On écrit $E \models \overline{\gamma}_1.\tau_1 \preceq \forall\overline{\alpha}.\tau_2$ *si, et seulement si,* $E \models \forall\overline{\alpha}.\exists\overline{\gamma}_1.\tau_1 = \tau_2$. ◇

7.2 Normalisation

Deux formes qui ne sont pas syntaxiquement compatibles au premier ordre (qui ne possèdent pas de plus grand majorant commun) doivent parfois être considérées compatibles en prenant le système d'équation E en compte.

Supposons par exemple que le système d'équations E soit formé de l'unique équation $\alpha = \beta_1 \to \beta_2$. Si une expression admet la forme α et la forme $\gamma.\beta_1 \to \gamma$ alors un algorithme raisonnable d'inférence locale ne doit ni échouer ni en conclure que l'expression a la forme $\bot$.

Au contraire, on peut très bien combiner ces deux formes grâce à l'équation $\alpha = \beta_1 \to \beta_2$. Mais, quel représentant choisir ? Nous choisissons la forme *la plus informative*, c'est-à-dire la forme qui expose le plus la structure de l'expression. Dans notre cas, il s'agit de $\beta_1 \to \beta_2$ qui illustre le fait que l'expression considérée est une fonction. Choisir cette information plus précise maximise la propagation de l'information de typage. En effet, par exemple, l'application de l'opérateur $\mathcal{D}(\cdot)$ réussira sur la forme $\beta_1 \to \beta_2$ et propagera la forme rigide β_1 alors que sur la forme α, elle sera indéfinie. La notion d'informativité qui nous intéresse est donc corrélée à la réussite de l'application des opérateurs partiels, et donc par extension, à la réussite de l'application des règles de l'inférence locale.

Cette notion d'informativité des formes est concrétisée par une *normalisation des formes* par rapport à une théorie équationnelle E. Si une équation de E est de la forme $\alpha = \tau$, avec τ qui n'est pas une variable de type, alors la normalisation réécrit α en τ.

La définition de la normalisation est simple mais introduit une certaine dose d'arbitraire dans le système lorsque l'on doit choisir un représentant entre deux variables de type égalisées par le système d'équations E. Si l'équation $\alpha = \beta$ est dans E, doit-on réécrire α en β ou l'inverse ? L'important est de toujours faire le même choix, par exemple en utilisant un ordre arbitraire sur les variables de type rigides. Ce choix influence la manière dont le programme est élaboré car les annotations ajoutées dans le programme par l'algorithme d'inférence locale vont utiliser les formes normalisées. Si le programme est mal typé dans MLGX, ces annotations vont avoir une influence sur les messages d'erreur du typeur.

Par ailleurs, nous supposerons que la contrainte E est satisfiable. C'est une hypothèse raisonnable car nous n'aurons pas à normaliser des formes dans un contexte où le système d'équation E est équivalent à false : dans un tel contexte, tout terme est bien typé et l'inférence de formes n'a donc pas d'intérêt.

De plus, comme nous interprétons les contraintes dans un modèle d'arbre fini, la satisfiabilité de la contrainte E implique que le système d'équations E soit acyclique. Cette hypothèse garantit la terminaison de la normalisation. On pourrait relaxer cette contrainte en introduisant des types equi-récursifs dans le système mais nous ne rentrons pas dans cette discussion dans cette formalisation.

Définition 7.2.1
Soit $<$, *un ordre total, fixé et arbitraire sur l'ensemble des variables de type. Alors, la relation de réécriture* $\rightsquigarrow_E$ *sur les types est engendrée par les axiomes suivants :*

$$\begin{array}{lllll} (1) & \alpha & \rightsquigarrow_E & \alpha' & \text{si } E \models \alpha = \alpha' \text{ et } \alpha' < \alpha \\ (2) & \alpha & \rightsquigarrow_E & \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 & \text{si } E \models \alpha = \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \\ (3) & \alpha & \rightsquigarrow_E & \tau_1 \to \tau_2 & \text{si } E \models \alpha = \tau_1 \to \tau_2 \end{array}$$

La relation $\rightsquigarrow_E$ peut effectivement être vue comme une fonction de normalisation comme le montrent les deux énoncés suivants :

Théorème 7.2.2
La relation $\rightsquigarrow_E$ est noethérienne. ◇

Preuve. L'ordre multi-ensemble induit par $<$ permet d'ordonner l'ensemble des variables libres des types et décroît strictement pour chacune des règles de réécriture. La non-cyclicité du système E est utilisée dans les règles (2) et (3) car la variable α ne peut pas apparaître dans un type τ auquel elle est égale *modulo* le système d'équations E. □

Théorème 7.2.3
La relation $\rightsquigarrow_E$ est confluente. ◇

Preuve. D'après le lemme de Newman et le lemme 7.2.2, il suffit de montrer que cette relation est localement confluente. On montre directement pour chaque paire de réécriture possible qu'on peut trouver un type joignant.

▷ *Cas* « (1) et (2) » On a $E \models \alpha = \alpha'$ et $E \models \alpha = \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$ donc par transitivité de l'égalité, $E \models \alpha' = \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$ et on peut donc joindre α' à $\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$ par application de la règle (2).

▷ *Cas* « (1) et (3) » Même raisonnement.

▷ *Cas* « (2) et (3) » C'est un cas impossible car il suppose l'égalité de $\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2$ et de $\tau_1 \to \tau_2$ sous le système d'équations E et le système d'équations E est aussi supposé satisfiable.

Il existe donc une et une seule forme normale sous E (satisfiable et non cyclique) calculée par la fermeture transitive et réflexive de $\rightsquigarrow_E$.

Notation 7.2.1
On écrit $\tau\lfloor_E$ pour la forme normale du type τ. On écrit $s\lfloor_E$ pour $\overline{\gamma}.(\tau\lfloor_E)$ quand s est $\overline{\gamma}.\tau$ et $\overline{\gamma} \# \mathbf{ftv}(E)$ est valide. Les notations $\theta\lfloor_E$ et $\varsigma\lfloor_E$ sont définies de la même façon. ◇

Cette définition a un sens car l'ordre sur les variables $\overline{\gamma}$ est sans importance (car $\overline{\gamma} \# E$). La forme normale de s sous E est un représentant de la classe d'équivalence de s :

Lemme 7.2.4
$E \models s = s\lfloor_E$ est valide. ◇

Preuve. Par induction sur la taille de la dérivation de normalisation. Le cas de base où la forme s est déjà normalisée est clair, $s = s\lfloor_E$. Si on suppose que $s \rightsquigarrow_E s' \rightsquigarrow_E^\star s\lfloor_E$ alors on sait par induction que $E \models s' = s\lfloor_E$. Or, les règles réécrivent un type en un type égal *modulo* E donc $E \models s = s'$ donc $E \models s = s\lfloor_E$ par transitivité de l'égalité. □

La normalisation est stable à travers l'équivalence des contraintes.

Lemme 7.2.5
Si $E \equiv E'$ alors $s\lfloor_E = s\lfloor_{E'}$. ◇

Preuve. Par définition de l'équivalence entre contraintes, les égalités entre types impliquées par E sont exactement les mêmes que celles impliquées par E'. □

La normalisation commute avec la substitution :

Lemme 7.2.6
$([\gamma \mapsto \tau_1]\tau_2)\lfloor_E = [\gamma \mapsto \tau_1\lfloor_E](\tau_2\lfloor_E)$ si $\gamma \# \mathbf{ftv}(\tau_1, E)$ ◇

Preuve. Immédiat par une induction sur τ_2. □

La normalisation préserve l'ordre d'instanciation :

Lemme 7.2.7
$s_1 \preceq s_2$ implique $s_1 \downarrow_E \preceq s_2 \downarrow_E$. ◇

Preuve. La définition de $s_1 \preceq s_2$ équivaut à

$$\overline{\gamma}_2.[\overline{\gamma}_1 \mapsto \overline{\tau}_1]\tau_1 = \overline{\gamma}_2.\tau_2$$

qui implique par $\|\overline{\gamma}_1\|$ applications du lemme précédent :

$$\overline{\gamma}_2.[\overline{\gamma}_1 \mapsto (\overline{\tau}_1 \downarrow_E)](\tau_1 \downarrow_E) = \overline{\gamma}_2.\tau_2 \downarrow_E$$

ce qui équivaut à $s_1 \downarrow_E \preceq s_2 \downarrow_E$. □

La normalisation est préservée par les opérations $\sqcup$, $\mathcal{D}(\bullet)$ et $\mathcal{C}(\bullet)$ et l'application d'un constructeur de type ou de la flèche :

Lemme 7.2.8
On a :
(i) $(s_1 \sqcup s_2)\downarrow_E = (s_1 \downarrow_E) \sqcup (s_2 \downarrow_E)$;
(ii) $\mathcal{D}(s \downarrow_E) = \mathcal{D}(s) \downarrow_E$;
(iii) $\mathcal{C}(s \downarrow_E) = \mathcal{C}(s) \downarrow_E$.
(iv) $(\varepsilon\ \ (\overline{s}))\downarrow_E = \varepsilon\ \ (\overline{s \downarrow_E})$
(v) $(s_1 \to s_2)\downarrow_E = s_1 \downarrow_E \to s_2 \downarrow_E$ ◇

Preuve. Par une induction sur le calcul de la forme normalisée et grâce à la confluence de $\rightsquigarrow_E$. □

La normalisation par rapport à un système E implique la normalisation dans un système plus faible :

Lemme 7.2.9
On a $\rightsquigarrow_E \subseteq \rightsquigarrow_{E \wedge E'}$ ◇

Preuve. Chaque règle de $\rightsquigarrow_E$ est déclenchée lorsqu'une implication de la forme $E \models C$ est valide et donc *a fortiori* lorsqu'une contrainte de la forme $E \wedge E' \models C$ l'est. □

Lemme 7.2.10
Soient E et E', deux systèmes d'équations. Pour toute forme s, $s\downarrow_{E\wedge E'}\downarrow_E = s\downarrow_{E\wedge E'}$. ◇

Preuve. C'est une reformulation du lemme 7.2.9. □

7.3 Élagage

7.3.1 Présentation

Un problème n'a pas été abordé jusqu'à maintenant : comment garantir que la propagation des types est correcte, c'est-à-dire que les annotations et les coercions insérées respectent l'intention du programmeur ?

Supposons que le programme à élaborer soit bien typé dans MLGI (mais qu'il manque des annotations de type pour qu'il le soit aussi dans MLGX). On ne peut certainement pas espérer qu'à coup sûr l'élaboration soit capable de trouver toutes les annotations de type nécessaires car cela reviendrait à espérer la complétude de l'inférence de types. Cependant, on veut garantir que l'élaboration maintient la propriété d'être bien typé dans MLGI. Dans le cas contraire, l'élaboration serait contre-productive puisqu'elle briserait la validité du programme !

Avec ce point essentiel en tête, on préfèrera *ne pas insérer d'annotations plutôt que d'insérer une annotation incorrecte.*

Cette propriété est, à notre avis, un des points essentiels manquant à l'implémentation des GADT dans GHC. L'exemple Haskell suivant est assez édifiant.

```
data EqT a b where EqP :: EqT a a

eq :: ∀ a. a → a → Bool
eq x y = True

f1 :: ∀ a b. EqT a b → b → a → Bool
f1 e x y =
  let z = case e of EqP →x in
    eq z y

f2 :: ∀ a b. EqT a b → a → b → Bool
f2 e x y =
  let z = case e of EqP → x in
    eq z y
```

Dans cet exemple, la déclaration de la fonction *f1* est rejetée par GHC (version 6.7) mais pas celle de la fonction *f2* alors que ces deux fonctions ne diffèrent que par le type donné à chaque variable dans les schémas de type. On observe ici clairement un choix arbitraire fait par l'algorithme d'inférence de type.

Dans MLGX, ces deux fonctions sont ambiguës et rejetées : le type de la variable z est le même que celui de x car aucune coercion n'a été demandée. Les types de z et de y sont donc incompatibles.

Ce défaut de GHC est dû à la propagation d'un type incorrect par l'algorithme d'inférence. En effet, dans la branche la variable x a le type a et le type b. N'ayant pas d'indication sur le type de la variable z, GHC choisit d'appliquer la coercion arbitraire (a ▷ b) ce qui est valide pour la seconde fonction mais par pour la première. Cette annotation est donc incorrecte puisqu'elle transforme la première fonction qui était bien typé dans MLGI en une fonction mal typé dans MLGI. Nous devons donc éviter d'insérer de telles annotations incorrectes.

Pour réaliser cela, quelques précautions sont à prendre.

Imaginons que l'équation $\alpha = \beta$ soit disponible au sein d'une branche d'un *pattern matching*. Supposons en sus que cette branche a le type α. Dans MLGI, il est aussi valide que la branche a le type β. Il est donc correct, et cela justifie l'introduction des opérateurs $\preceq_E$ et $=_E$ de la section précédente, de raisonner *modulo* un système d'équations E. Une forme s dénote implicitement tous les types τ tels que $E \models s \preceq \tau$ est valide.

Cependant, en dehors de la branche, l'équation $\alpha = \beta$ n'est plus valable donc, il y a une différence entre inférer la forme α ou la forme β puisque ces formes ne représentent plus le même ensemble de types. En somme, il est essentiel de reconsidérer le choix des représentants lorsque la théorie équationnelle change à la sortie des branches car un choix arbitraire entre α et β est fatal du point de vue du typage dans MLGI. On préfère donc inférer $\bot$ plutôt que α ou β lorsque ces deux variables ne sont plus égalisées par le système d'équations E. Nous appelons *élagage* ce processus d'approximation des formes.

7.3.2 Définitions et propriétés

L'ensemble des types rigides qui sont des instances d'une forme s *modulo* un système d'équations E sera au cœur des raisonnements sur l'élagage. On formalise donc cet objet.

Définition 7.3.1
On appelle dénotation *de la forme s sous le système d'équations E, notée $(\!|\ s\ |\!)_E$, l'ensemble de tous les types τ tels que $E \models s \preceq \tau$ est valide.* ◇

La propriété suivante montre le lien entre la relation d'instance sur les formes et l'inclusion de leur dénotation.

Lemme 7.3.2
Si $s \preceq s'$ alors $(\!|\ s'\ |\!)_E \subseteq (\!|\ s\ |\!)_E$. ◇

Preuve. On a $E \models s \preceq s'$. Soit $\tau \in (\!|\ s'\ |\!)_E$ alors $E \models s' \preceq \tau$. Par transitivité, $E \models s \preceq \tau$ et donc $\tau \in (\!|\ s\ |\!)_E$. □

On définit l'élagage à l'aide cette notion sémantique :

Définition 7.3.3
La forme obtenue par élagage de la forme s par rapport à E et E', notée $s\restriction_{E,E'}$, est la plus petite borne supérieure des formes s' telles que $s' \preceq s$ est valide et que la dénotation de s' sous E contienne celle de s sous $E \wedge E'$.

On note $\mathcal{V}_{E,E'}(s)$, l'ensemble des formes plus générales que la forme s et qui sont des élagages valides, c'est-à-dire :

$$\mathcal{V}_{E,E'}(s) = \{s' \mid s' \preceq s \wedge (\!|\ s\ |\!)_{E\wedge E'} \subseteq (\!|\ s'\ |\!)_E\}$$

On définit donc l'élagage de la forme s par rapport aux systèmes d'équations E et E' ainsi :

$$s\restriction_{E,E'} = \bigsqcup \mathcal{V}_{E,E'}(s)$$ ◇

Une forme s' est une forme élaguée valide pour la forme s par rapport aux systèmes d'équations E et E' si elle correspond à la forme s dans laquelle on a oublié de l'information ($s' \preceq s$) et qui vérifie la condition de sûreté ($(\!|\ s\ |\!)_{E\wedge E'} \subseteq (\!|\ s'\ |\!)_E$) assurant que l'on n'a pas fait de choix de représentant dans E. Faire l'union de toutes les formes qui sont des élagages valides permet de minimiser la perte d'information.

La première chose à remarquer est la bonne formation de cette définition. L'ensemble des formes admet un majorant qui est s. C'est un ensemble fini car tout ensemble de formes majoré est fini. Il admet donc une plus petite borne supérieure.

Ensuite, on peut reformuler cette définition de manière plus algorithmique. L'élagage est effectué à la frontière entre un système d'équations E et un système plus riche $E \wedge E'$. On se fixe une forme s. La dénotation de s sous E est toujours un sous-ensemble de sa dénotation sous $E \wedge E'$. Si nous n'avons pas de chance, c'est un sous-ensemble strict, ce qui signifie que la dénotation de s change si on passe de $E \wedge E'$ à E. *L'élagage de s consiste à supprimer des sous-termes de manière à empêcher ce phénomène.* Il s'agit donc de déterminer la forme la plus précise s' telle que $s' \preceq s$ est valide et que la dénotation de s' sous E contienne celle de s sous $E \wedge E'$. Autrement dit, on ne fait pas de choix de représentants dans $E \wedge E'$ qui puissent être différents sous E.

Exemple 7.3.4
On se donne E valant true et E' valant $\alpha = \beta_1 \times \beta_2$. Ainsi, élaguer la forme s' valant $\gamma.\alpha \to \gamma$ par rapport à E et E' donne une forme s valant $\gamma_1\gamma_2.\gamma_1 \to \gamma_2$. En effet, la dénotation de s' sous $E \wedge E'$ contient tous les types de la forme $(\beta_1 \times \beta_2) \to \tau$. Or, sa dénotation sous E ne contient pas ces types donc le sous-terme α doit être élagué. La dénotation de s sous E contient elle tous ces types.

Plus subtil : élaguer la forme s' valant $\gamma_1\gamma_2.\gamma_1 \times \gamma_2$ par rapport à E et E' donne la forme s valant $\bot$. En effet la dénotation de s' sous $E \wedge E'$ contient le type α et sa dénotation sous E ne le contient pas. Le produit à la racine de s' doit donc être élagué. On peut vérifier que la dénotation de s ne contient pas α. ◇

Avant de rentrer dans les détails algorithmiques de cette opération, on montre quelques propriétés sur sa spécification.

L'ensemble des formes suffisamment élaguées pour une forme s est un sous-ensemble des formes suffisamment élaguées de ses instances.

Lemme 7.3.5
Si $s \preceq \tau$ alors $\mathcal{V}_{E,E'}(s) \subseteq \mathcal{V}_{E,E'}(\tau)$ ◇

Preuve. Pour toute forme s' telle que $s' \in \mathcal{V}_{E,E'}(s)$, on a $s' \preceq s$ donc $s' \preceq \tau$. On a aussi $(\!|\ s\ |\!)_{E \wedge E'} \subseteq (\!|\ s'\ |\!)_E$. Le lemme 7.3.2 implique $(\!|\ \tau\ |\!)_{E \wedge E'} \subseteq (\!|\ s\ |\!)_{E \wedge E'}$. Il vient $(\!|\ \tau\ |\!)_{E \wedge E'} \subseteq (\!|\ s'\ |\!)_E$. On obtient donc $s' \in \mathcal{V}_{E,E'}(\tau)$. □

Lemme 7.3.6
On a $s\restriction_{E,E'} \preceq s$. De plus, la dénotation de $s\restriction_{E,E'}$ sous E contient celle de s sous $E \wedge E'$. ◇

Preuve. Premièrement, il est clair que $s_1 \preceq s$ et $s_2 \preceq s$ implique $s_1 \sqcup s_2 \preceq s$. Ensuite, la dénotation de $s_1 \sqcup s_2$ sous E est l'intersection des dénotations de s_1 et s_2 sous E. Ainsi, la propriété qui définit l'appartenance à S est préservée par les plus petites majorantes. Dès lors, $s\restriction_{E,E'}$ a aussi cette propriété. □

Lemme 7.3.7
Si $E \wedge E' \models s \preceq \tau$ alors $E \models s\restriction_{E,E'} \preceq \tau$. ◇

Preuve. Simple reformulation du lemme 7.3.6. □

Lemme 7.3.8
$s\restriction_{E,E'}$ est normalisée par rapport à E si s est normalisée par rapport à $E \wedge E'$. ◇

Preuve. Si la forme s est normalisée vis-à-vis de $E \wedge E'$ alors s est normalisée vis-à-vis de E, et majore $s\restriction_{E,E'}$, qui est donc normalisée aussi. □

Le lemme suivant sert à caractériser un type rigide qui doit être élagué en tête.

Lemme 7.3.9
$\tau\restriction_{E,E'} = \bot$ si et seulement si il existe un type τ' tel que τ et τ' n'ont pas le même symbole de tête, $\tau' \in (\!|\ \tau\ |\!)_{E \wedge E'}$ et $\tau' \notin (\!|\ \tau\ |\!)_E$. ◇

Preuve. Supposons $\tau \equiv \tau_1 \to \tau_2$. Les autres cas sont similaires.

$\tau\restriction_{E,E'} = \bot$ équivaut à $\forall s'.s' \preceq s \wedge (\!|\ \tau\ |\!)_{E \wedge E'} \subseteq (\!|\ s'\ |\!)_E \Rightarrow s' = \bot$. Ceci implique que la forme $\bot \to \bot$ est telle que $(\!|\ \tau\ |\!)_{E \wedge E'} \not\subseteq (\!|\ s'\ |\!)_E$. Il existe donc un type τ' tel que $\tau' \in (\!|\ \tau\ |\!)_{E \wedge E'}$ mais $\tau' \notin (\!|\ \bot \to \bot\ |\!)_E$. Or, $\bot \to \bot \preceq \tau$ donc d'après le lemme 7.3.2, $\tau' \notin (\!|\ \tau\ |\!)_E$. L'autre sens est clair. □

7.3.3 Description algorithmique

Si la normalisation par rapport à un système d'équations rigides est une opération très simple et facile à implémenter car il s'agit d'un système de réécriture, l'implémentation efficace de l'opération d'élagage n'est pas tout à fait immédiate.

En effet, pour calculer efficacement la partie à élaguer, il faut avoir une représentation intelligente des systèmes d'équations E et $E \wedge E'$ qui facilite, pour un sous-terme donné, le calcul implicite de la dénotation de ce sous-terme. Décider si la dénotation de ce sous-terme sous E n'englobe pas sa dénotation sous $E \wedge E'$ permet de décider si il doit être élagué.

Il est bien connu que l'algorithme *congruence closure* implémenté à l'aide de graphe dirigé avec partage est une représentation efficace pour décider si deux termes du premier ordre sont égaux *modulo* une théorie équationnelle close (Baader & Nipkow, 1998).

Dans cette section, on présente une variante de cet algorithme permettant d'implémenter l'élagage d'une forme s par rapport à E et $E \wedge E'$.

▶ **Rappels concernant congruence closure**

Définition 7.3.10 (Notations sur les graphes)
Soit un graphe dirigé sans cycle G. On note pour un nœud u :
- $\mathcal{L}(u)$ *pour son étiquette ;*
- $\delta(u)$ *pour la liste ordonnée de ses successeurs* $u[1], \ldots, u[n]$. ◇

On représente une forme $\overline{\gamma}.\tau$ en se donnant un nœud pour chaque symbole et chaque variable (rigide ou flexible) et en tirant une arête étiquetée par i d'un symbole de fonction vers le nœud représentant le sous-terme τ_i. La représentation graphique de la forme $\overline{\gamma}.\tau$ est valide pour un certain $\overline{\gamma}$ fixé. Deux formes α-équivalentes n'ont donc pas la même représentation graphique. Plus formellement, le terme représenté par $\mathcal{T}(u)$ est tel que :

$$\begin{aligned}
\mathcal{T}(u) &= \mathcal{T}(u_1) \to \mathcal{T}(u_2) \\
&\quad \text{si } \mathcal{L}(u) =' \to' \text{ et } \delta(u) = u_1, u_2 \\
\mathcal{T}(u) &= \varepsilon\,(\mathcal{T}(u_1) \ldots \mathcal{T}(u_n)) \\
&\quad \text{si } \mathcal{L}(u) = \varepsilon \text{ et } \delta(u) = u_1, \ldots, u_n \\
\mathcal{T}(u) &= \alpha \\
&\quad \text{si } \mathcal{L}(u) = \alpha \\
\mathcal{T}(u) &= \gamma \\
&\quad \text{si } \mathcal{L}(u) = \gamma \text{ avec } \gamma \in \overline{\gamma}
\end{aligned}$$

La forme représentée par le nœud u est donc $\overline{\gamma}.\mathcal{T}(u)$.

Définition 7.3.11 (Congruence sur les graphes)
Une relation $\sim$ est une congruence si c'est une relation d'équivalence et que pour tous nœuds u et v tels que $\mathcal{L}(u) = \mathcal{L}(v)$, on a (avec n, l'arité de $\mathcal{L}(u)$) :

$$\bigwedge_{i=1}^{n} u[i] \sim v[i] \Rightarrow u \sim v$$

◇

Le calcul efficace de la plus petite congruence contenant une relation d'équivalence $=$ est un algorithme connu (Baader & Nipkow, 1998). La figure 7.2 est une implémentation purement fonctionnelle de cet algorithme. On note $\sim_\emptyset$, la relation identité.

On utilise une structure Union/Find purement fonctionnelle (Conchon & Filliâtre, 2007) pour implémenter les classes d'équivalence de $\sim$. Par abus de notation, on note cet ensemble de classe d'équivalence $\sim$. Pour alléger les notations, lorsque l'on écrit $\widehat{u}$, on fait référence au résultat de *find* $\sim u$. La fonction *union* $\sim \widehat{u}\ \widehat{v}$ construit un nouvel ensemble de classes d'équivalence qui est le même que celui de $\sim$ dans lequel les classes d'équivalence du nœud u et du nœud v ont été fusionnées.

La fonction *merge* prend en argument la relation $\sim$ et deux nœuds u et v tels que $\mathcal{T}(u) = \mathcal{T}(v)$. On construit une nouvelle relation $\sim$ qui prend en compte cette nouvelle équation. Pour cela, on fusionne les classes d'équivalences du nœud u et du nœud v et on effectue une fermeture par congruence en itérant sur les prédécesseurs des nœuds dans le graphe. La fonction *congruence_closure* construit incrémentalement une relation $\sim$ en itérant sur les équations du système E (la fonction *node_of* se contente de calculer la représentation graphique d'un type τ en allouant de nouveaux nœuds et la fonction *fold* est une simple itération d'ordre supérieur commune dans les langages fonctionnels).

Les détails de la preuve de cet algorithme se trouvent déjà dans la littérature (Baader & Nipkow, 1998). Le théorème de correction peut s'énoncer ainsi :

Théorème 7.3.12 (Correction de *congruence closure*)
$E \wedge E' \models \mathcal{T}(u) = \mathcal{T}(v)$ *si et seulement si* $u \sim' v$
avec $\sim' =$ *congruence_closure* $\sim E'$ *et* $\sim =$ *congruence_closure* $\sim_\emptyset E$. ◇

```
let congruent ∼ u v =
  L(u) = L(v) ∧ ∀(p, q) ∈ δ(p) × δ(q), p ∼ q.

let rec merge ∼ u v =
  if (u ≁ v) then begin
    let Q = δ⁻¹(û) and P = δ⁻¹(v̂) in
    let ∼ = union ∼ û v̂ in
      fold (P × Q) ∼ (λ(p, q) ∼. if p ≁ q ∧ congruent ∼ p q then merge ∼ p q else ∼)
  end

let rec congruence_closure ∼ E =
  fold E ∼ (λ(τ1, τ2) ∼. let u1 = node_of τ1 and u2 = node_of τ2 in merge ∼ u1 u2)
```

FIG. 7.2: Version fonctionnelle pure de *congruence closure*.

À l'aide de l'algorithme *congruence closure*, il est donc possible de tester efficacement si deux types rigides sont égaux *modulo* un système d'équations E. Il suffit de tester l'égalité des classes d'équivalence de leurs racines dans la représentation induite par la relation $\sim$.

Pour décider l'équivalence de s_1 et s_2 *modulo* un système d'équations E, il faut tester si toutes leurs instances sont égalisables *modulo* E. Pour cela, on construit les représentations des deux formes. On essaie alors d'unifier les deux représentations obtenues (en utilisant évidemment les variables de type flexibles $\overline{\gamma}$ comme des variables d'unification).

▶ **Élagage : cas d'une forme rigide τ** Pour comprendre comment tirer parti de *congruence closure* pour implémenter efficacement l'élagage, nous allons nous focaliser dans un premier temps sur le cas où la forme s à élaguer est rigide, c'est-à-dire un type connu τ.

Dans ce cas, la représentation du type τ par *congruence closure* est aussi une représentation efficace de $(\!|\ \tau\ |\!)_E$ puisque $(\!|\ \tau\ |\!)_E = \{\tau' \mid E \models \tau \preceq \tau'\} = \{\tau' \mid E \models \tau = \tau'\}$. De même, en rajoutant des équations E' à la représentation de E par *congruence closure*, on obtient une représentation efficace de $(\!|\ \tau'\ |\!)_{E \wedge E'}$.

D'après lemme 7.3.9, on doit élaguer un type rigide τ en tête si il existe un type τ' dans $(\!|\ \tau\ |\!)_{E \wedge E'}$ dont le symbole de tête est différent de celui de τ et qui n'est pas égal à τ *modulo* E. Cette propriété sur le symbole de tête s'exprime très facilement en terme de graphe : il suffit que la classe d'équivalence du nœud u tel que $\mathcal{T}(u) = \tau$ ne soit pas la même sous E et sous $E \wedge E'$ pour que $\tau\restriction_{E,E'} = \bot$. En rajoutant une marque sur chaque classe d'équivalence correspondant au système d'équations qui l'a formé, on peut déterminer en temps constant si un nœud doit être élagué. On se donne donc une fonction $\Delta(u)$ renvoyant une marque associée à la classe d'équivalence du nœud u. On se donne aussi une fonction *mark* telle que *mark* $\sim \widehat{u}\ m$ calcule une nouvelle relation $\sim$ dans laquelle les classes d'équivalence sont préservées mais la classe d'équivalence $\widehat{u}$ est marquée par m.

La seule opération qui contribue à la formation des classes d'équivalence est l'opération *union*. Il suffit donc de rajouter un nouvel argument à *union*, la marque correspondant au système d'équations courant, pour savoir quel est le dernier système qui a contribué à la formation d'une certaine classe d'équivalence. On modifie donc la fonction *merge* pour qu'elle passe ce nouvel argument à la fonction *union*. Le code obtenu se trouve dans la figure 7.3.

▶ **Élagage** On peut maintenant présenter informellement l'algorithme d'élagage d'une forme $s \equiv \overline{\gamma}.\tau$ dont l'implémentation est décrite par la figure 7.4.

```
let union' ~ m û v̂ =
  let ~ = union ~ û v̂ in
    mark ~ û m

let rec merge ~ m u v =
  if (u ⊀ v) then begin
    let Q = δ⁻¹(û) and P = δ⁻¹(v̂) in
    let ~ = union' ~ m û v̂ in
      fold (P × Q) ~ (λ(p, q) ~. if p ⊀ q ∧ congruent ~ p q then merge ~ m p q else ~)
  end
```

FIG. 7.3: Fonction *merge* modifiée par une marque.

L'appel *trim* $E\ \overline{\gamma}.\tau$ calcule l'élagage de la forme $\overline{\gamma}.\tau$. Cette fonction utilise une fonction auxiliaire *trim_type* qui prend en entrée un type τ, un ensemble de variable de types flexibles $\overline{\gamma}$ et qui renvoie un type τ' et un ensemble de variables flexibles $\overline{\gamma}'$ tels que $\overline{\gamma}.\tau\restriction_{E,E'} = \overline{\gamma\gamma}'.\tau'$. Nous allons maintenant expliquer informellement la fonction *trim_type*.

Comme dit plus haut, il s'agit d'un parcours en profondeur de l'arbre de syntaxe du type τ. Pour décider si l'arbre doit être élagué en tête ou si on doit plutôt élaguer certains de ses sous-arbres en laissant son symbole de tête, on s'intéresse à toutes ses instances rigides.

Si il existe une instance rigide τ de la forme s dont le changement de théorie équationnelle nécessite un élagage en tête alors la forme s doit aussi être élaguée en tête.

Lemme 7.3.13
Soit une forme s. Si il existe un type τ tel que $s \preceq \tau$ et $\tau\restriction_{E,E'} = \bot$ alors $s\restriction_{E,E'} = \bot$. ◇

Preuve. Si $\tau\restriction_{E,E'} = \bot$ alors $\bigsqcup \mathcal{V}_{E,E'}(\tau) = \bot$. Or, comme $s \preceq \tau$, par le lemme 7.3.5, $\mathcal{V}_{E,E'}(s) \subseteq \mathcal{V}_{E,E'}(\tau)$. Finalement, $s\restriction_{E,E'} = \bigsqcup \mathcal{V}_{E,E'}(s) = \bigsqcup \mathcal{V}_{E,E'}(\tau) = \bot$. □

Sinon, si pour toute instance rigide τ de la forme s, le symbole de tête ne pose pas de problème d'ambiguïté et peut être maintenu alors l'élagage doit se faire sur les sous-termes de la structure de la forme. Une subtilité empêche cependant d'effectuer une récursion immédiate.

En effet, si on cherche à calculer la forme $s = \gamma.\gamma \to \gamma\restriction_{E,E'}$, pour $E \equiv E' \equiv \mathsf{true}$, on sait que la forme s ne doit pas être élaguée en tête. Il ne faut pas encore conclure pour autant que $\gamma.\gamma \to \gamma\restriction_{E,E'} = (\gamma.\gamma\restriction_{E,E'}) \to (\gamma.\gamma\restriction_{E,E'}) = \gamma\gamma'.\gamma \to \gamma'$ car on perdrait le partage entre le type d'entrée et le type de sortie de cette fonction.

Empêcher le départage n'est pas non plus concevable puisque si on prend $E \equiv \mathsf{true}$ et $E' \equiv \alpha \times \alpha = \delta \wedge \beta \times \beta = \rho$, alors le calcul de $\gamma.(\gamma \times \alpha) \to (\beta \times \gamma)\restriction_{E,E'}$vaudrait $\gamma.(\gamma \times \alpha\restriction_{E,E'} \to \beta \times \gamma\restriction_{E,E'})$, c'est-à-dire $\gamma.(\gamma \times \alpha\restriction_{E,E'} \to \beta \times \gamma\restriction_{E,E'})$, ce qui est faux puisque cette forme doit être élaguée en $\bot \to \bot$.

Dans cet exemple, pour être correct, il faut calculer $\gamma.(\gamma \times \alpha)\restriction_{E,E'} = \bot$ puis $\gamma.(\gamma \times \alpha)\restriction_{E,E'} = \bot$ et en déduire que la forme recherchée est $\bot \to \bot$. Dans le cas du premier exemple, on calcule $\gamma.\gamma\restriction_{E,E'} = \gamma.\gamma$ et lorsque l'on remonte sur la flèche, il faut remarquer que les variables flexibles sont inchangées et les repartager entre le type d'entrée et le type de sortie.

On note m le nombre de nœuds alloués et n le nombre de nœuds de la représentation de s. Durant la descente à travers la forme s, on doit se poser la question de l'unification du sous-terme qu'on examine avec l'un des nœuds alloués. La complexité de l'unification du premier ordre est linéaire (Martelli & Montanari, 1982) en le nombre de nœuds de ce sous-arbre et on doit donc explorer m unifications possibles en pire cas. En pire cas, on doit aussi supposer que toutes ces unifications échouent et que l'algorithme poursuit son parcours à travers tous les sous-termes, reproduisant ces unifications avec

```
let rec trim_type E γ̄ τ =
  match τ with
    | γ → (∅, γ)
    | α → if Δ(α̂) ≠ E then ⊥ else (∅, α)
    | (τ1 → τ2) →
        if ∃u.γ̄.τ ⪯ T(u), Δ(u) ≠ E then
          ⊥
        else
          let (γ̄1, τ1) = trim_type E γ̄ τ1 in
          let (γ̄2, τ2) = trim_type E γ̄ τ2 in
            (γ̄1γ̄2, τ1 → τ2)
    | ε (τ1 ... τn) →
        if ∃u ∈ G.γ̄.τ ⪯ T(u), Δ(u) ≠ E then
          ⊥
        else
          let (γ̄1, τ1) = trim_type E γ̄ τ1 in
            ...
          let (γ̄n, τn) = trim_type E γ̄ τn in
            (γ̄1 ... γ̄n, ε τ 1 ... τn)

let trim E γ̄.τ =
  let (γ̄′, τ) = trim_type E γ̄ τ in
    γ̄γ̄′.τ
```

FIG. 7.4: Fonction d'élagage de la forme $\overline{\gamma}.\tau$ par rapport aux systèmes d'équations E et $E \wedge E'$.

les nœuds sur des sous-termes plus petits. Plus formellement, si le nombre d'opérations effectuées par la fonction *trim* est notée $T(n)$ alors on a :

$$T(|u|) = m\,\Theta(|u|) + \sum_{v \in \delta(u)} T(|v|)$$

Or, $|u| = 1 + \sum_{v \in \delta(u)} |v|$ donc on peut reformuler cette équation ainsi :

$$T(n) = m\,\Theta(n) + \sum_{i=1}^{k} n_i$$

pour $n = 1 + \sum_{i=1}^{k} n_i$

Lemme 7.3.14 (Complexité de la fonction trim dans le pire cas)
La complexité en pire cas du nombre d'opérations $T(n)$ de la fonction trim est $O(mn^2)$ où m est la taille du graphe G, n celle de la forme s. ◇

Preuve. Montrons par récurrence sur n qu'il existe une constante positive c telle que $T(n) \preceq m\,c\,n^2$.

▷ *Cas* « $n = 0$ ». Immédiat.

▷ *Cas* « $n > 0$ ». L'équation de récurrence dans le cas le pire s'écrit :

$$T(n) = m\,\Theta(n) + \max_{n=1+\sum_{i=1}^{k} n_i} \sum_{i=1}^{k} T(n_i)$$

Par hypothèse d'induction, on obtient :

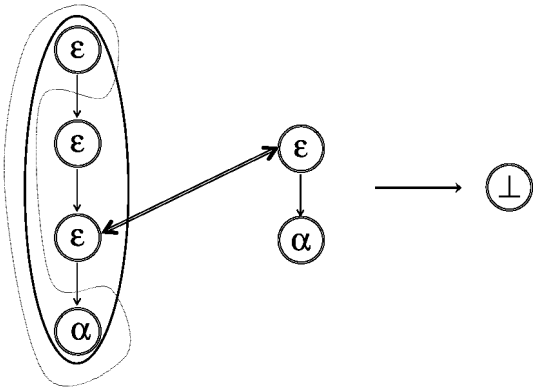

FIG. 7.5: Exécution de la fonction *trim* sur la forme $\varepsilon\,\alpha$ avec $E \equiv \varepsilon^3\,\alpha = \alpha$ et $E' \equiv \varepsilon^2\,\alpha = \alpha$.

$$T(n) \preceq m\,\Theta(n) + \max_{n=1+\sum_{i=1}^{k} n_i} \sum_{i=1}^{k} m\,c\,n_i^2$$

La fonction $(n_1, \ldots, n_k) \mapsto \sum_{i=1}^{k} n_i^2$ atteint son maximum sur les bornes définies par l'équation $n = 1 + \sum_{i=1}^{k} n_i^2$ au point $(n-1, 0, \ldots, 0)$ par exemple. On en déduit :

$$T(n) \preceq m\,(\Theta(n) + c\,(n-1)^2)$$

donc

$$T(n) \preceq m\,c\,n^2$$ □

▶ **Exemples d'élagage** On donne maintenant quelques exemples d'exécution de la fonction *trim*. Ces exemples sont donnés de manière graphique. On illustre d'une part les représentations graphiques des systèmes d'équations : les classes d'équivalence du système E sont entourées par un trait fin, les classes d'équivalence du système E' sont entourées par un trait en gras. On représente d'autre part l'entrée et la sortie de l'exécution de l'élagage. Cette dernière est représentée par la flèche horizontale. Enfin, lorsqu'un nœud de la forme qu'on élague s'unifie avec un nœud du graphe, on le signifie à l'aide d'une flèche double.

La figure 7.5 montre le calcul de $\varepsilon\,\alpha \restriction_{\varepsilon^3\,\alpha=\alpha, \varepsilon^2\,\alpha=\alpha}$. La conjonction des deux systèmes implique par congruence que $\varepsilon\,\alpha = \alpha$. Dès lors, les types $\varepsilon^3\,\alpha, \varepsilon^2\,\alpha, \varepsilon^2\,\alpha$ et α sont égaux sous $E \wedge E'$. Dans le système E par contre, $\varepsilon^2\,\alpha \neq \alpha$ donc sa classe d'équivalence a changé ce qui explique l'élagage à la racine de la forme $\varepsilon\,\alpha$. Ce premier exemple montre que notre algorithme fonctionne aussi sur les systèmes d'équations cycliques. La non-cyclicité est une hypothèse liée à la normalisation uniquement.

La figure 7.6 montre le calcul de $\alpha \times \beta \restriction_{\alpha\times\beta=\delta, \alpha=\omega}$. Dans cet exemple, il n'est pas nécessaire d'élaguer la forme à la racine puisque $(\!|\ \alpha \times \beta\ |\!)_{\alpha\times\beta=\delta\wedge\alpha=\omega} \subseteq (\!|\ \gamma.\gamma \times \beta\ |\!)_{\alpha\times\beta}$. Le sous-terme α doit être élagué puisque lorsque l'équation $\alpha = \omega$ n'est plus valable, le choix de représenter la classe d'équivalence par α plutôt que ω est problématique. L'algorithme prend sa décision en remarquant que le nœud racine du terme $\alpha \times \beta$ dans le graphe n'a pas été marqué par le nouveau système d'équations, contrairement au nœud α.

La figure 7.7 illustre le calcul de $\gamma.\gamma \times \gamma \restriction_{\mathsf{true}, \delta=\alpha\times\beta}$. Cette forme ne s'unifie pas avec l'un des types rigides représentés dans le graphe donc aucun élagage n'est nécessaire.

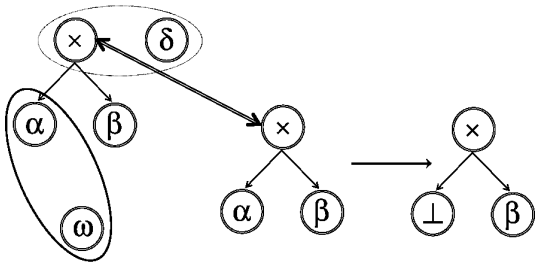

FIG. 7.6: Exécution de la fonction *trim* sur la forme $\alpha \times \beta$ avec $E \equiv \alpha \times \beta = \delta$ et $E' \equiv \alpha = \omega$.

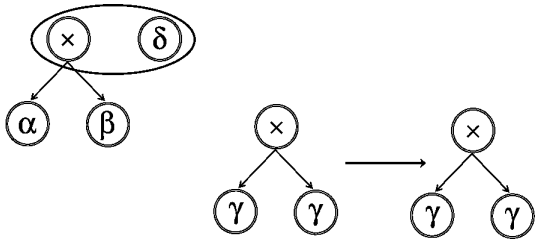

FIG. 7.7: Exécution de la fonction *trim* sur la forme $\gamma.\gamma \times \gamma$ avec $E \equiv \mathsf{true}$ et $E' \equiv \alpha \times \beta = \delta$.

Dans l'exemple de la figure 7.8, on calcule $\gamma\gamma'.\gamma \times \gamma' \restriction_{\mathsf{true},\delta=\alpha\times\beta}$. Ici, la forme $\gamma\gamma'.\gamma \times \gamma'$ s'unifie avec $\alpha \times \beta$ et doit donc être élaguée.

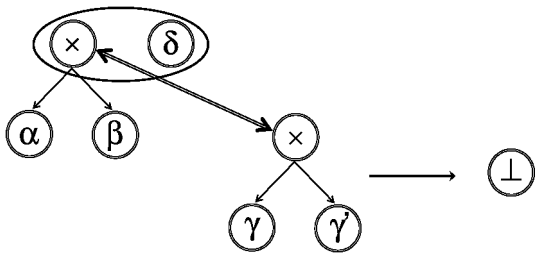

FIG. 7.8: Exécution de la fonction *trim* sur la forme $\gamma\gamma'.\gamma \times \gamma'$ avec $E \equiv \mathsf{true}$ et $E' \equiv \alpha \times \beta = \delta$.

CHAPITRE HUIT

Deux algorithmes d'inférence locale

L'ensemble des outils développés dans le chapitre 7 fournit des blocs élémentaires sur lesquels peut s'appuyer un algorithme d'inférence locale (ou d'inférence de formes) agissant en amont de MLGX dans un système d'inférence de types stratifiée. En réalité, on peut classifier ces algorithmes en fonction de la manière dont l'information de type est propagée à travers l'arbre de syntaxe. Dans la section 8.1, nous décrivons un tel algorithme d'inférence locale, que nous appelons *Wob*. Il est conçu pour calquer le fonctionnement du système « wobbly types » de Simon Peyton Jones *et al.* (2004). Ensuite, nous décrivons un algorithme plus précis dans la section 8.2. Chacun de ces algorithmes a une complexité linéaire sous l'hypothèse que toutes les formes s et le système d'équation E sont de taille bornée.

8.1 Élaboration bi-directionnelle du système Wob

8.1.1 Présentation informelle

Suivant le mécanisme introduit par Benjamin Pierce (Pierce & Turner, 2000) et repris par Simon Peyton Jones *et al.* pour l'ajout des GADT et du polymorphisme d'ordre supérieur (Vytiniotis *et al*, 2005) à l'inférence de types de GHC, *Wob* est bidirectionnel : il agit soit en mode « inférence », soit en mode « vérification ». Un jugement en mode inférence prend la forme

$$E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$$

Ses entrées sont le système d'équation E, l'environnement Γ, qui associe des formes aux variables, et le terme t. Ses sorties sont la forme inférée s et le terme transformé t'.

Un jugement en mode « vérification » prend la forme

$$E, \Gamma \vdash t \Downarrow s \rightsquigarrow t'$$

Ses entrées et sorties sont analogues au jugement précédent excepté la forme s qui est maintenant une entrée, la forme attendue pour le terme t. La définition du système se trouve dans les figures 8.1 et 8.2.

Dans les deux modes, un invariant est maintenu : la forme s est normalisée par rapport à E. Comme on l'a expliqué dans la section 7.2, normaliser les formes est nécessaire pour éviter les erreurs d'unification « bêtes » dues à une prise en compte partielle du système d'équations E. Quand nous écrivons $s_1 \sqcup s_2$, nous nous assurons que les formes s_1 et s_2 sont toutes les deux normalisées. Quand nous écrivons $\mathcal{D}(s)$ ou $\mathcal{C}(s)$, nous nous assurons aussi que s est normalisée. Les formes qui apparaissent dans Γ ne sont pas nécessairement normalisées.

En général, le terme transformé t' est identique au terme t mais où

(i) toutes les annotations explicites sont normalisées,
(ii) de nouvelles annotations de type sont ajoutées sur les analyses par cas et
(iii) des coercions sont insérées lorsque l'on utilise des variables et sur certaines branches de *pattern matching*. La normalisation peut être vue comme une heuristique qui augmente la probabilité du terme transformé à être bien typé dans MLGX.

Nous supprimons la construction $\mu(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t$ et la remplaçons par une nouvelle construction $\mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t$, qui est identique, excepté que les variables de type $\overline{\alpha}$ sont liées non seulement dans le type τ mais aussi dans le terme t. Cette nouvelle construction peut aussi être vue comme un sucre syntaxique pour $\mu(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).\forall\overline{\alpha}.t$. Nous exploitons cette formulation dans les règles Fix-⇑ et Fix-⇓, expliquées dans la section suivante.

Nous interdisons les coercions explicites dans les termes du langage de surface parce qu'elles n'apportent pas plus de pouvoir d'expression que les annotations de type du point de vue de l'inférence stratifiée. En effet, pour que $\exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)$ soit une coercion valide, il faut que $E \models \forall\overline{\gamma}.\tau_1 = \tau_2$ soit valide, ce que implique une égalité entre les normalisations des types τ_1 et τ_2. Comme *Wob* normalise toutes les annotations de type écrites par le programmeur, une coercion de type dégénèrerait en une annotation de type simple dans le terme transformé. Dans le langage de surface, il n'y a donc aucune différence entre une coercion de la forme $(t : \exists\overline{\gamma}.(\tau_1 \triangleright \tau_2))$ et une annotation de type $(t : \overline{\gamma}.(\tau_1 \downarrow_E \sqcup \tau_2 \downarrow_E))$.

8.1.2 Formalisation

L'inférence de formes pour le système *Wob* est définie par l'ensemble des règles des figures 8.1 et 8.2 (les deux parties correspondent juste à un partitionnement pour la mise en page). Ses règles s'appuient sur les notations suivantes :

Notation 8.1.1
$(t \downarrow_E \overline{\gamma}.\tau)$ *est un sucre syntaxique pour* $(t : \exists\overline{\gamma}.(\tau \triangleright \tau \downarrow_E))$, *en supposant* $\overline{\gamma} \# \mathbf{ftv}(E)$. ◇

Notation 8.1.2
$(t \uparrow_E \overline{\gamma}.\tau)$ *est un sucre syntaxique pour* $(t : \exists\overline{\gamma}.(\tau \downarrow_E \triangleright \tau))$, *en supposant* $\overline{\gamma} \# \mathbf{ftv}(E)$. ◇

Elles correspondent à l'insertion d'une coercion pour normaliser la forme d'un terme ou pour transformer la forme normalisée d'un terme en un représentant particulier de sa classe d'équivalence *modulo* le système d'équations E.

On peut maintenant expliquer chaque règle.

► **Var** La règle Var-⇑ cherche la forme s associée à x dans l'environnement. Elle produit une forme inférée $s\downarrow_E$ qui vérifie donc l'invariant forçant toutes les formes inférées à être normalisées par rapport à E. Cette étape de normalisation correspond à une conversion de type : le type de x, une instance de s, est transformé en une instance de $s\downarrow_E$. Pour informer MLGX de ce changement de type, ceci doit être reflété dans le terme transformé par l'insertion d'une coercion explicite. La règle produit donc le terme $(x \downarrow_E s)$. En somme, ce terme signifie que, pour certaines valeurs des variables de type flexibles $\overline{\gamma}$ qui seront inférées par MLGX, le type τ est converti en le type $\tau\downarrow_E$. La règle Var-⇓ est analogue. La forme attendue s' est ignorée. Nous verrons dans la conclusion de cette thèse que cette forme pourrait contribuer à l'inférence de l'environnement si celui-ci était une sortie de notre algorithme.

► **Lam** La règle Lam-⇑ extrait l'annotation de type explicite θ qui décore x et la remplace par $\theta\downarrow_E$ dans le terme transformé. De même, l'environnement de forme Γ est étendu avec l'association $x : \theta\downarrow_E$, et la forme inférée pour la fonction est $\theta\downarrow_E \to s$ si la forme inférée pour le terme t est s. La règle Lam-⇓ est analogue mais combine la forme attendue s avec l'information contenue dans l'annotation de type. Par exemple, si s est $\gamma.\gamma \to \gamma$ et θ est *int* alors la combinaison engendrée s' est $int \to int$, donc x : *int* est ajouté à l'environnement et le terme t est traité avec la forme attendue *int*.

$$\textsc{Var-}\Uparrow\ \frac{(x : s) \in \Gamma}{E, \Gamma \vdash\ x \Uparrow s \downarrow_E \rightsquigarrow (x \downarrow_E s)} \qquad \textsc{Var-}\Downarrow\ \frac{(x : s) \in \Gamma}{E, \Gamma \vdash\ x \Downarrow s' \rightsquigarrow (x \downarrow_E s)}$$

$$\textsc{Lam-}\Uparrow\ \frac{E, \Gamma; x : \theta \downarrow_E \vdash\ t \Uparrow s \rightsquigarrow t'}{E, \Gamma \vdash\ \lambda(x : \theta).t \Uparrow (\theta \downarrow_E \to s) \rightsquigarrow \lambda(x : \theta \downarrow_E).t'} \qquad \textsc{Lam-}\Downarrow\ \frac{s' = s \sqcup (\theta \downarrow_E \to \bot) \quad E, \Gamma; x : \mathcal{D}(s') \vdash\ t \Downarrow \mathcal{C}(s') \rightsquigarrow t'}{E, \Gamma \vdash\ \lambda(x : \theta).t \Downarrow s \rightsquigarrow \lambda(x : \theta \downarrow_E).t'}$$

$$\textsc{App-}\Uparrow\ \frac{E, \Gamma \vdash\ t_1 \Uparrow s \rightsquigarrow t_1' \quad E, \Gamma \vdash\ t_2 \Downarrow \mathcal{D}(s) \rightsquigarrow t_2'}{E, \Gamma \vdash\ t_1\ (t_2) \Uparrow \mathcal{C}(s) \rightsquigarrow t_1'\ (t_2')} \qquad \textsc{App-}\Downarrow\ \frac{E, \Gamma \vdash\ t_1 \Uparrow s_1 \rightsquigarrow t_1' \quad E, \Gamma \vdash\ t_2 \Downarrow \mathcal{D}(s_1 \sqcup (\bot \to s)) \rightsquigarrow t_2'}{E, \Gamma \vdash\ t_1\ (t_2) \Downarrow s \rightsquigarrow t_1'\ (t_2')}$$

$$\textsc{Let-}\Updownarrow\ \frac{E, \Gamma \vdash\ t_1 \Uparrow s_1 \rightsquigarrow t_1' \quad E, \Gamma; x : s_1 \vdash\ t_2 \Updownarrow s_2 \rightsquigarrow t_2'}{E, \Gamma \vdash\ \mathsf{let}\, x\ =\ t_1\ \mathsf{in}\ t_2 \Updownarrow s_2 \rightsquigarrow \mathsf{let}\, x\ =\ t_1'\ \mathsf{in}\ t_2'}$$

$$\textsc{Fix-}\Uparrow\ \frac{\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma) \quad E, \Gamma; x : \overline{\gamma\alpha}.\tau \downarrow_E \vdash\ t \Downarrow \overline{\gamma}.\tau \downarrow_E \rightsquigarrow t'}{E, \Gamma \vdash\ \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t \Uparrow \overline{\gamma\alpha}.\tau \downarrow_E \rightsquigarrow \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau \downarrow_E).t'}$$

$$\textsc{Fix-}\Downarrow\ \frac{\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma, s) \quad E, \Gamma; x : (\overline{\gamma\alpha}.\tau \downarrow_E\ \sqcup\ s) \vdash\ t \Downarrow (\overline{\gamma}.\tau \downarrow_E\ \sqcup\ s) \rightsquigarrow t'}{E, \Gamma \vdash\ \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t \Downarrow s \rightsquigarrow \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau \downarrow_E).t'}$$

FIG. 8.1: L'inférence de formes pour le système *Wob*, partie I.

► **App** Tout comme Simon Peyton Jones *et al.*, les règles APP-⇑ et APP-⇓ infèrent toutes deux la forme de la fonction et utilisent cette information pour vérifier la forme de l'argument. Dans la règle APP-⇓, la forme de l'argument s_1 est combinée à la forme $\bot \to s$, reflétant le fait que la forme de l'application est connue. Comme le fait remarquer Simon Peyton Jones *et al.* (2004, Section 4.6), ces règles ne sont pas très « subtiles » : la forme inférée pour *id* (x), où *id* a la forme $\gamma.\gamma \to \gamma$ et x a la forme *int* est $\bot$. En effet, comme la variable x est examinée en mode vérification, la forme *int* est ignorée. L'algorithme qui nous présenterons dans la section 8.2 est conçu pour remédier à ce problème.

► **Let** La règle LET-⇳ est immédiate. La méta-variable ⇳ signifie ⇑ ou bien ⇓. Aucune généralisation dans le style de Hindley-Milner n'est nécessaire ici car il n'y a rien à généraliser : les seules variables de type libres dans une forme sont des variables rigides.

► **Fix** La règle FIX-⇑ exploite l'annotation de type attachée à la construction $\mu^\star$ pour examiner le terme t en mode vérification. La subtilité, c'est que l'annotation de type polymorphe $\exists\overline{\gamma}.\forall\overline{\alpha}.\tau$ est transformée en deux formes différentes. La forme inférée pour la construction dans son ensemble est (la forme normalisée de) $\overline{\gamma\alpha}.\tau$, une forme où les variables de type $\overline{\alpha}$ sont liées. Cette forme est aussi associée à la variable x dans l'environnement de façon à ce que x puisse être utilisée à différents types à l'intérieur de sa propre définition (autrement dit, il s'agit d'une récursion polymorphe). Cependant,

$$\textsc{Cstr-}\Uparrow\quad \frac{K :: s \qquad \forall i \quad E, \Gamma \vdash\ t_i \Uparrow s_i \rightsquigarrow t'_i}{E, \Gamma \vdash\ K\,(t_1 \ldots t_n) \Uparrow \mathcal{C}(s \sqcup (s_1 \times \ldots \times s_n \to \bot)) \rightsquigarrow K\,(t'_1 \ldots t'_n)}$$

$$\textsc{Cstr-}\Downarrow\quad \frac{K :: s \qquad \forall i \quad E, \Gamma \vdash\ t_i \Downarrow \mathcal{D}_i(s \sqcup (\bot \times \ldots \times \bot \to s')) \rightsquigarrow t'_i}{E, \Gamma \vdash\ K\,(t_1 \ldots t_n) \Downarrow s' \rightsquigarrow K\,(t'_1 \ldots t'_n)}$$

$$\textsc{Case-}\Uparrow\quad \frac{E, \Gamma \vdash\ t \Uparrow s' \rightsquigarrow t' \qquad \forall i \quad E, \Gamma \vdash\ (p_i : s').t_i \Uparrow s_i \rightsquigarrow p_i.t'_i}{E, \Gamma \vdash\ \mathsf{match}\, t\, \mathsf{with}\ p_1.t_1 \ldots p_n.t_n \Uparrow \sqcup_i s_i \rightsquigarrow \mathsf{match}\,(t' : s')\, \mathsf{with}\ p_1.t'_1 \ldots p_n.t'_n}$$

$$\textsc{Case-}\Downarrow\quad \frac{E, \Gamma \vdash\ t \Uparrow s' \rightsquigarrow t' \qquad \forall i \quad E, \Gamma \vdash\ (p_i : s').t_i \Downarrow s \rightsquigarrow p_i.t'_i}{E, \Gamma \vdash\ \mathsf{match}\, t\, \mathsf{with}\ p_1.t_1 \ldots p_n.t_n \Downarrow s \rightsquigarrow \mathsf{match}\,(t' : s')\, \mathsf{with}\ p_1.t'_1 \ldots p_n.t'_n}$$

$$\textsc{Forall-}\Uparrow\quad \frac{\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma) \qquad E, \Gamma \vdash\ t \Uparrow s \rightsquigarrow t'}{E, \Gamma \vdash\ \forall \overline{\alpha}.t \Uparrow \overline{\alpha}.s \rightsquigarrow \forall \overline{\alpha}.t'}$$

$$\textsc{Forall-}\Downarrow\quad \frac{\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma, s) \qquad E, \Gamma \vdash\ t \Downarrow s \rightsquigarrow t'}{E, \Gamma \vdash\ \forall \overline{\alpha}.t \Downarrow s \rightsquigarrow \forall \overline{\alpha}.t'}$$

$$\textsc{Annot-}\Uparrow\quad \frac{E, \Gamma \vdash\ t \Downarrow \theta \lfloor_E \rightsquigarrow t'}{E, \Gamma \vdash\ (t : \theta) \Uparrow \theta \lfloor_E \rightsquigarrow (t' : \theta \lfloor_E)}$$

$$\textsc{Annot-}\Downarrow\quad \frac{E, \Gamma \vdash\ t \Downarrow (\theta \lfloor_E \sqcup s) \rightsquigarrow t'}{E, \Gamma \vdash\ (t : \theta) \Downarrow s \rightsquigarrow (t' : \theta \lfloor_E)}$$

$$\textsc{Clause-}\Uparrow\quad \frac{p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\overline{\gamma}.\Gamma') \vdash\ t \Uparrow s \rightsquigarrow t' \qquad \overline{\beta} \# \mathbf{ftv}\,(E, \Gamma, s \lceil_{E,E'}) \qquad \overline{\gamma} \# \mathbf{ftv}\,(E, \Gamma, \overline{\tau}_2, t)}{E, \Gamma \vdash\ (p : \overline{\gamma}.\varepsilon\, \overline{\tau}_1\, \overline{\tau}_2).t \Uparrow s \lceil_{E,E'} \rightsquigarrow p.t'}$$

$$\textsc{Clause-}\Downarrow\quad \frac{p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\overline{\gamma}.\Gamma') \vdash\ t \Downarrow s \lfloor_{E \wedge E'} \rightsquigarrow t' \qquad \overline{\beta} \# \mathbf{ftv}\,(E, \Gamma, s) \qquad \overline{\gamma} \# \mathbf{ftv}\,(E, \Gamma, \overline{\tau}_2, t, s)}{E, \Gamma \vdash\ (p : \overline{\gamma}.\varepsilon\, \overline{\tau}_1\, \overline{\tau}_2).t \Downarrow s \rightsquigarrow p.(t' \uparrow_{E \wedge E'} s)}$$

FIG. 8.2: L'inférence de formes pour le système *Wob*, partie II.

la forme attendue pour le terme t est plus précise : c'est (la forme normalisée de) $\overline{\gamma}.\tau$, une forme où les variables de type $\overline{\alpha}$ sont exposées. La convention introduite par $\mu^\star$ qui lie désormais les variables de type rigides $\overline{\alpha}$ à l'intérieur du terme t permet de donner du sens cette règle. La règle FIX-⇓ est similaire mais combine l'annotation de type de la forme attendue s.

▶ **Cstr** Les applications des constructeurs de données pourraient être traitées comme des applications quelconques. Cependant, la règle CSTR-⇑ adopte une approche différente qui rappelle l'« application intelligente » de Simon Peyton Jones *et al.* (la règle APPN (2004, section 4.6)). Les arguments sont examinés en mode inférence – et non en mode vérification – pour engendrer des formes $s_1, \ldots, s_n$. Le schéma de type du constructeur de données, vu comme une forme s est alors unifié avec la forme $s_1 \times \ldots \times s_n \to \bot$. On obtient alors une instance adaptée de s dont le codomaine est la forme inférée désirée. Par exemple, si on suppose que *Some* a le schéma de type $\forall\alpha.\alpha \to \mathit{option}\ \alpha$ et la variable x a la forme *int*, cette règle permet d'inférer que *Some* x a la forme *option int*. La règle CSTR-⇓ est analogue à la règle APP-⇓. L'opérateur de forme $\mathcal{D}_i(\cdot)$ extrait la i-ème composante du domaine de son argument : sa définition est analogue à celle de $\mathcal{D}(\cdot)$.

▶ **Forall** Dans la règle Forall-⇑, les variables de type rigides $\overline{\alpha}$ peuvent apparaître libres dans la forme s pour qu'elles puissent être abstraites dans la forme inferée $\overline{\alpha}.s$. Dans la règle Forall-⇓, la forme attendue s peut faire intervenir des variables de type quantifiées $\overline{\gamma}$ mais nous ne pouvons pas deviner comment elles coïncident avec les variables de type universellement quantifiées $\overline{\alpha}$. Par conséquent, la seule chose que nous pouvons faire est de passer s inchangée. Par exemple, quand nous confrontons le terme $\forall\alpha\beta.\lambda(x:\alpha).\lambda(y:\beta).x$ à la forme $s = \gamma_1\gamma_2.\gamma_1 \to \gamma_2 \to \gamma_1$, nous ne pouvons pas deviner que les variables de type γ_1 et γ_2 dans la forme attendue coïncident avec respectivement α et β dans le source sans avoir observé $\lambda(x:\alpha).\lambda(y:\beta).x$. Au final, nous vérifions aussi le sous-terme $\lambda(x:\alpha).\lambda(y:\beta).x$ à l'aide de la forme s. C'est assez précis puisque, dans ce cas, grâce à la règle Lam-⇓, nous sommes capables de retrouver l'information manquante et vérifier en dernier lieu la correspondance entre la dernière occurrence de la variable x et la forme α. Ce problème de correspondance et les faiblesses qui en résultent sont les raisons pour lesquelles nous combinons μ et $\forall$ dans $\mu^\star$: cette astuce permet de pallier à ce problème.

▶ **Annot** Les règles Annot-⇑ et Annot-⇓ n'utilisent pas d'idées supplémentaires. L'annotation de type θ est normalisée dans le terme transformé. Le sous-terme t est toujours examiné en mode vérification.

▶ **Case** Les règles Case-⇑ et Case-⇓ sont immédiates. Le terme t est toujours examiné en mode inférence ce qui construit une forme s'. Cette forme est passée à la règle Clause-⇑ et Clause-⇓ où elle est exploitée pour déterminer quelles sont les nouvelles équations dont on peut tenir compte.

▶ **Clause** Les règles Clause-⇑ et Clause-⇓ sont très similaires à la règle Clause de MLGX. Nous introduisons néanmoins une importante restriction. La nouvelle condition $\overline{\gamma} \# \mathbf{ftv}(\overline{\tau}_2)$ oblige *la forme (inférée) du terme analysé à être totalement explicite sur la partie concernant les paramètres généralisés* : ils ne peuvent pas contenir de variables de type flexibles. Comme nous l'avons expliqué dans le chapitre 4, on acquiert ainsi une *connaissance totale* de E. Si la condition n'est pas remplie, le programme est rejeté.

La première prémisse des règles Clause-⇑ et Clause-⇓ confronte le motif p à la forme du terme analysé pour obtenir de nouvelles variables de type rigides $\overline{\beta}$, de nouvelles équations E' et un nouvel environnement de typage Γ'. Les variables flexibles $\overline{\gamma}$ peuvent être libres dans Γ' donc nous devons les abstraire, point à point, pour produit un nouvel environnement de forme $\overline{\gamma}.\Gamma'$.

La seconde prémisse des règles Clause-⇑ et Clause-⇓ examine le sous-terme t. Les deux règles ont des formulations qui diffèrent de manière subtile.

Dans la règle Clause-⇓, la forme inférée s est normalisée par rapport à $E \wedge E'$ de manière à maintenir l'invariant que la forme attendue est normalisée par rapport à la version courante du système d'équations. Comme pour les règles Var-⇑ et Var-⇓, cette étape de normalisation correspond à une conversion de type : de l'extérieur, le type de la branche est une instance de s mais de l'intérieur, c'est une instance de $s\downarrow_{E\wedge E'}$. Une fois de plus, ceci est reflété par l'insertion d'une coercion explicite de type.

Dans la règle Clause-⇑ aucune coercion n'est insérée car aucune forme attendue n'est fournie, donc aucune normalisation n'est nécessaire.

Au contraire, la forme s inférée pour le terme t est élaguée pour produire une forme inférée $s\restriction_{E,E'}$ pour la clause. La nécessité de l'élagage a été discutée plus tôt (voir la section 7.3).

En deux mots, retourner s pourrait être incorrecte : elle pourrait être le représentant d'une classe d'équivalence de formes valide dans $E \wedge E'$ mais pas dans E.

Les formes (élaguées) s_i renvoyée par les règles Clause-⇑ pour chaque branche sont unifiées dans Case-⇑, ce qui engendre une forme $\sqcup_i s_i$.

8.1.3 Correction du système *Wob*

Les différences principales entre l'algorithme original de Simon Peyton Jones *et al.* et notre système *Wob* sont

(i) notre utilisation de l'unification pour propager les annotations de type, présente dans l'opérateur de « plus petite borne inférieure » sur les formes, qui rend notre algorithme plus précis (l'unification est aussi présente dans la règle APPN de Simon Peyton Jones *et al.*) ;

(ii) notre insistance sur la connaissance complète des équations qui apparaissent sur les *pattern matching* et notre utilisation de l'élagage.

Tout programme bien typé de MLGI peut être transformé en un programme bien-typé par rapport à la combinaison du système *Wob* et de MLGX en rajoutant suffissament d'annotations de type. Ceci est prouvé par le théorème 6.3.2 de complétude avec assistance de MLGX.

Pour montrer le théorème de correction du système *Wob* qui va suivre, quelques lemmes sont nécessaires. Il s'agit d'abord de montrer que la normalisation des formes inférées est un invariant de l'algorithme d'inférence locale.

Lemme 8.1.1
Si $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$ *alors* s *est normalisée par rapport à* E. ◇

Preuve. Par induction sur les dérivations de $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t$. On discrimine sur la dernière règle appliquée pour produire le jugement.

▷ *Cas* « VAR-⇑, FIX-⇑, ANNOT-⇑ ». On normalise la forme inférée donc la conclusion est triviale.

▷ *Cas* « LAM-⇑ ». La conclusion est de la forme $E, \Gamma \vdash \lambda(x : \theta).t \Uparrow (\theta\downarrow_E \to s) \rightsquigarrow \lambda(x : \theta\downarrow_E).t'$ et l'hypothèse $E, \Gamma; x : \theta\downarrow_E \vdash t \Uparrow s \rightsquigarrow t'$. Par induction, s est normalisée par rapport au système d'équations E donc la forme produite $\theta\downarrow_E \to s$ l'est aussi par application du lemme 7.2.8.

▷ *Cas* « APP-⇑ » La conclusion de la règle est $E, \Gamma \vdash t_1\ (t_2) \Uparrow \mathcal{C}(s) \rightsquigarrow t_1'\ (t_2')$ avec pour hypothèse $E, \Gamma \vdash t_1 \Uparrow s \rightsquigarrow t_1' E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s) \rightsquigarrow t_2'$. Par induction, s est normalisée par rapport au système d'équation E donc $\mathcal{C}(s)$ l'est aussi par application du lemme 7.2.8.

▷ *Cas* « CSTR-⇑ ». L'hypothèse d'induction appliquée aux arguments permet d'affirmer qu'on infère des formes normalisées. La normalisation est préservée par $\mathcal{C}(\bullet)$, l'application de constructeurs de type et l'union entre les formes comme l'affirme le lemme 7.2.8.

▷ *Cas* « CASE-⇑ ». Même argument car on infère l'union des formes inférées pour les branches qui sont, par induction, normalisées et l'union préserve la normalisation par le lemme 7.2.8

▷ *Cas* « LET-⇑, FORALL-⇑, COERCE-⇑ ». Clair par l'hypothèse d'induction appliquée au sous-terme dont on renvoie la forme inférée.

▷ *Cas* « CLAUSE-⇑ ». La conclusion de la règle est $E, \Gamma \vdash (p : \overline{\gamma}.\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2).t \Uparrow s\restriction_{E,E'} \rightsquigarrow p.t'$ et parmi ses hypothèses, on trouve $E \wedge E', \Gamma(\overline{\gamma}.\Gamma') \vdash t \Uparrow s \rightsquigarrow t'$. Par induction, la forme s est normalisée par rapport à $E \wedge E'$. D'après le lemme 7.3.8, cela implique $s\restriction_{E,E'}$ est normalisée par rapport à E. □

Vient maintenant la formalisation de la notion de *connaissance totale* des équations qu'on obtient en forçant les paramètres généralisés des annotations de types sur les matchs à être rigides. Dans l'énoncé suivant, cette propriété est écrite $\overline{\gamma} \# \overline{\tau}_2'$.

Lemme 8.1.2
Si $p : \varepsilon\,\overline{\tau_1}\,\overline{\tau_2} \vdash (\overline{\beta}, E', \Gamma)$, $E \models \overline{\gamma}.\varepsilon\,\overline{\tau_1}'\,\overline{\tau_2}' \preceq \varepsilon\,\overline{\tau_1}\,\overline{\tau_2}$, $\overline{\gamma} \# \overline{\tau}_2'$ *et* $p : \varepsilon\,\overline{\tau_1}'\,\overline{\tau_2}' \vdash (\overline{\beta}, E'', \Gamma')$ *alors* $E \wedge E' \equiv E \wedge E''$ *et* $E \models \Gamma' \preceq \Gamma$. ◇

Preuve. On suppose que $p \equiv K\ \overline{\beta}\ x_1 \ldots x_n$ et $K : \forall\overline{\alpha}\overline{\beta}.\tau_1 \times \ldots \times \tau_n \to \varepsilon\,\overline{\alpha}\,\overline{\tau}$. En inversant l'hypothèse $p : \varepsilon\,\overline{\tau_1}\,\overline{\tau_2} \vdash (\overline{\beta}, E', \Gamma)$, on a

$$K \preceq \forall\overline{\beta}.\tau_1' \times \ldots \times \tau_n' \to \varepsilon\,\overline{\tau}_1\,\overline{\tau}$$

et $\overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}_1, \overline{\tau}_2)$ avec $E' \equiv \overline{\tau}_2 = \overline{\tau}$ et $\Gamma \equiv (x_1 : \tau'_1, \ldots, x_n : \tau'_n)$. En inversant $p : \varepsilon\,\overline{\tau_1}'\,\overline{\tau_2}' \vdash (\overline{\beta}, E'', \Gamma')$, on a

$$K \preceq \forall\overline{\beta}.\tau''_1 \times \ldots \times \tau''_n \to \varepsilon\,\overline{\tau}'_1\,\overline{\tau}$$

et $\overline{\beta} \# \mathbf{ftv}\,(\overline{\tau}'_1, \overline{\tau}'_2)$ avec $E' \equiv \overline{\tau}'_2 = \overline{\tau}$ et $\Gamma \equiv (x_1 : \tau''_1, \ldots, x_n : \tau''_n)$.

Il existe une substitution $\phi \equiv [\overline{\alpha} \mapsto \overline{\tau}_1]$ (respectivement $\phi' \equiv [\overline{\alpha} \mapsto \overline{\tau}'_1]$) telle que pour tout $i \in 1 \ldots n$, $\phi(\tau_i) = \tau'_i$ (respectivement $\phi'(\tau_i) = \tau''_i$). L'hypothèse $E \models \overline{\gamma}.\varepsilon\,\overline{\tau_1}'\,\overline{\tau_2}' \preceq \varepsilon\,\overline{\tau_1}\,\overline{\tau_2}$ nous donne donc pour tout $i \in 1 \ldots n$, $\overline{\gamma}.\tau''_i = \overline{\gamma}.\phi'(\tau_i)$ d'où

$$E \models \overline{\gamma}.\phi'(\tau_i) \preceq \phi(\tau_i)$$

Ce dernier jugement se réécrit $E \models \overline{\gamma}.\tau''_i \preceq \tau'_i$. On a donc $\Gamma' \preceq \Gamma''$.

Comme $\overline{\gamma} \# \overline{\tau}'_2$, les types $\overline{\tau}'_2$ sont rigides. On a donc $E \models \overline{\tau}'_2 = \overline{\tau}_2$. Les équations $\overline{\tau} = \overline{\tau}_2$ et $\overline{\tau} = \overline{\tau}'_2$ sont donc équivalentes sous E par transitivité de l'égalité *modulo* E. □

Sans surprise, on ne change pas le comportement de l'algorithme d'inférence locale en remplaçant le système d'équations par un système d'équations sémantiquement équivalent.

Lemme 8.1.3
Si $E \equiv E'$ et $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$ alors $E', \Gamma \vdash t \Uparrow s \rightsquigarrow t'$. ◇

Preuve. Le système d'équations n'intervient que pour effectuer des normalisations. Or, d'après le lemme 7.2.5, on sait que $s\lfloor_E = s\lfloor_{E'}$ si $E \equiv E'$. Par induction sur les jugements de la forme $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$, il suffit donc de remplacer chaque E par E' pour obtenir une dérivation valide du jugement $E', \Gamma \vdash t \Uparrow s \rightsquigarrow t'$. □

Lemme 8.1.4
Si $E \equiv E'$ et $E, \Gamma \vdash t \Downarrow s \rightsquigarrow t'$ alors $E', \Gamma \vdash t \Downarrow s \rightsquigarrow t'$. ◇

Preuve. Même preuve que le lemme 8.1.3. □

Nous donnons maintenant un énoncé formel de la correction de *Wob*. L'énoncé suppose l'existence d'une dérivation du jugement $E, \Gamma \vdash t : \sigma$ dans MLGI. Naturellement, en pratique, cette dérivation n'est pas connue : elle existe uniquement dans l'esprit du programmeur. Ici, elle est utilisée comme un oracle : l'affirmation qu'une forme s est une approximation correcte du type véritable de t est encodée par l'énoncé $E \models s \preceq \sigma$. L'hypothèse que l'algorithme d'inférence de formes est initialement alimenté par des formes correctes est encodé par l'énoncé $E \models \Gamma' \preceq \Gamma$ où Γ' est l'environnement de forme initial fourni à l'algorithme. L'affirmation que les annotations et les coercions insérées par l'algorithme sont correctes est encodée par le fait que le terme t' a encore le type σ dans MLGI. (Ce dernier point n'implique pas que t' soit bien typé dans MLGX.) Par conséquent, le point 1 du théorème 8.1.5 qui va suivre peut être lu :

> *Si le système* Wob *est évalué en mode inférence et sous des hypothèses correctes, alors il produit une forme correcte et insère des annotations et des coercions correctes.*

Le point 2 affirme une propriété similaire sur le mode inférence.

Nous pouvons enfin énoncer le théorème de correction du système *Wob* tel que nous l'avons décrit plus haut.

Théorème 8.1.5 (Correction du système *Wob*)
Soit $E, \Gamma \vdash t : \sigma$ valide dans MLGI. Supposons $E \models \Gamma' \preceq \Gamma$. Alors,

1. *Si $E, \Gamma' \vdash t \Uparrow s \rightsquigarrow t'$ est valide dans* Wob*, alors $E \models s \preceq \sigma$ et $E, \Gamma \vdash t' : \sigma$ est valide dans MLGI.*
2. *Si $E, \Gamma' \vdash t \Downarrow s \rightsquigarrow t'$ est valide dans* Wob*, $E \models s \preceq \sigma$, et s est normalisée par rapport à E, alors $E, \Gamma \vdash t' : \sigma$ est valide MLGI.* ◇

Le corollaire utilisé sur les programmes clos s'écrit :

Corollaire 8.1.1
Soit $E, \Gamma \vdash t : \sigma$ *valide dans MLGI. Supposons* $E \models \Gamma' \preceq \Gamma$. *Alors,*

1. *Si* $E, \Gamma' \vdash t \Uparrow s \rightsquigarrow t'$ *est valide dans* Wob, *alors* $E \models s \preceq \sigma$ *et* $E, \Gamma \vdash t' : \sigma$ *est valide dans MLGI.*
2. *Si* $E, \Gamma' \vdash t \Downarrow s \rightsquigarrow t'$ *est valide dans* Wob, $E \models s \preceq \sigma$, *et* s *est normalisée par rapport à* E, *alors* $E, \Gamma \vdash t' : \sigma$ *est valide MLGI.* ◇

Preuve. Par induction sur les dérivations dans MLGI.

▷ *Cas* « $E \equiv \mathsf{false}$ ». En mode inférence, la conclusion est $\mathsf{false}, \Gamma' \vdash t \Uparrow \bot \rightsquigarrow t'$. Les jugements $E, \Gamma \vdash t : \sigma$ et $E \models \bot \preceq \sigma$ sont clairement valides puisqu'on peut déduire n'importe quel jugement de typage et n'importe quelle instanciation d'un système d'équations équivalent à false. En mode vérification, le jugement $E, \Gamma \vdash t' : \sigma$ est valide pour les mêmes raisons.

▷ *Cas* « Var ». La conclusion de l'inférence est

$$E, \Gamma' \vdash x \Uparrow s \rightsquigarrow (x \downarrow_E s) \tag{1}$$

Sa prémisse est $(x : s) \in \Gamma'$ (**2**). Le jugement de typage est

$$E, \Gamma \vdash x : \forall \overline{\alpha}.\tau \tag{3}$$

Nous avons aussi le fait que

$$\overline{\alpha} \# \Gamma, E \tag{4}$$

$E \models \Gamma' \preceq \Gamma$ implique $E \models s \preceq \forall \overline{\alpha}.\tau$. Donc, $E \models s \preceq \tau$. Par le lemme 7.1.8, il existe τ' tel que $s \preceq \tau'$ et

$$E \models \tau = \tau' \tag{5}$$

Par la règle Inst appliquée au jugement (3), il vient $E, \Gamma \vdash x : \tau$. Comme le jugement (5) est valide, on applique la règle Conv et on obtient

$$E, \Gamma \vdash x : \tau' \tag{6}$$

Par le lemme 7.2.7, le jugement $(\downarrow_E s) \preceq (\tau' \triangleright \tau' \downarrow_E)$ est valide. Donc, par la règle Coerce et par (6), il vient

$$E, \Gamma \vdash (x \downarrow_E s) : \tau' \downarrow_E$$

Par (5), par le lemme 7.2.4 et par deux applications de Conv, nous avons

$$E, \Gamma \vdash (x \downarrow_E s) : \tau$$

Par la règle Gen et (4), on déduit

$$E, \Gamma \vdash (x \downarrow_E s) : \forall \overline{\alpha}.\tau$$

Les mêmes arguments sont valides pour le mode vérification comme le type attendu n'est pas pris en compte par la règle Var-⇓.

▷ *Cas* « Forall ». Le jugement de type s'écrit $E, \Gamma \vdash \forall \overline{\alpha}.t : \forall \overline{\alpha}.\tau$ et ses prémisses sont $E, \Gamma \vdash t : \tau$ et

$$\overline{\alpha} \# E, \Gamma \tag{1}$$

La conclusion de l'inférence est

$$E, \Gamma' \vdash \forall \overline{\alpha}.t \Uparrow \overline{\alpha}.s \rightsquigarrow \forall \overline{\alpha}.t' \tag{2}$$

et ses prémisses sont $E, \Gamma' \vdash t \Uparrow s \rightsquigarrow t'$ et $\overline{\alpha} \# E, \Gamma'$. Par hypothèse d'induction,

$$E, \Gamma \vdash t' : \tau \tag{3}$$

et

$$E \models s \preceq \tau \tag{4}$$

Par la règle FORALL appliquée à (1) et (3), nous avons

$$E, \Gamma \vdash \forall \overline{\alpha}.t' : \forall \overline{\alpha}.\tau$$

(4) implique

$$E \models \overline{\alpha}.s \preceq \forall \overline{\alpha}.\tau$$

Les arguments pour le mode vérification sont similaires.

▷ *Cas* « ANNOT ». La conclusion du type est $E, \Gamma \vdash (t : \theta) : \tau$ et ses prémisses sont $E, \Gamma \vdash t : \tau$ et

$$\theta \preceq \tau \tag{1}$$

La conclusion de l'inférence est $E, \Gamma' \vdash (t : \theta) \Uparrow (\theta\lfloor_E \sqcup s) \rightsquigarrow (t' : \theta\lfloor_E)$ et ses prémisses sont $E, \Gamma' \vdash t \Uparrow s \rightsquigarrow t'$. Par hypothèse d'induction, il vient

$$E, \Gamma \vdash t' : \tau \tag{2}$$

et

$$E \models s \preceq \tau \tag{3}$$

Par le lemme 7.2.7, (1) implique

$$\theta\lfloor_E \preceq \tau\lfloor_E \tag{4}$$

$E \models \tau\lfloor_E = \tau$ (**5**), (2) et CONV entraine $E, \Gamma \vdash t' : \tau\lfloor_E$. Par la règle ANNOT et par (4), ceci conduit à $E, \Gamma \vdash (t' : \theta\lfloor_E) : \tau\lfloor_E$. Par la règle CONV et par (5), on a

$$E, \Gamma \vdash (t' : \theta\lfloor_E) : \tau$$

Par (4), par (3) et comme la forme s est normalisée, $\theta\lfloor_E \sqcup s$ est définie et $E \models (\theta\lfloor_E \sqcup s) \preceq \tau$.

En mode vérification, la forme s est normalisée donc $(\theta\lfloor_E \sqcup s)$ est définie. Comme $E \models s \preceq \tau$ et (1), $E \models (\theta\lfloor_E \sqcup s) \preceq \tau$. Nous pouvons appliqué l'hypothèse d'induction et les arguments de typage sont similaires à ceux du mode inférence.

▷ *Cas* « LAM ». La conclusion du typage est $E, \Gamma \vdash \lambda(x : \theta).t : \tau_1 \to \tau_2$ et ses prémisses sont $E, \Gamma; (x : \tau_1) \vdash t : \tau_2$ et

$$\theta \preceq \tau_1 \tag{1}$$

Le jugement d'inférence est $E, \Gamma' \vdash \lambda(x : \theta).t \Uparrow (\theta\lfloor_E \to s) \rightsquigarrow \lambda(x : \theta\lfloor_E).t'$ et ses prémisses sont $E, \Gamma'; (x : \theta) \vdash t \Uparrow s \rightsquigarrow t'$. (1) et $E \models \Gamma' \preceq \Gamma$ donne $E \models (\Gamma'; (x : \theta\lfloor_E)) \preceq (\Gamma; (x : \tau_1))$. Par hypothèse d'induction, nous avons

$$E, \Gamma; (x : \tau_1) \vdash t' : \tau_2 \tag{2}$$

et

$$E \models s \preceq \tau_2 \tag{3}$$

(1) et le lemme 7.2.7 implique

$$\theta\lfloor_E \preceq \tau_1\lfloor_E \tag{4}$$

Par la règle LAM appliquée à (2) et (4),

$$E, \Gamma \vdash \lambda(x : \theta\lfloor_E).t' : \tau_1\lfloor_E \to \tau_2$$

On a

$$E \models (\tau_1 \downarrow_E \to \tau_2) = (\tau_1 \to \tau_2)$$

ce qui permet d'appliquer la règle Conv pour obtenir

$$E, \Gamma \vdash \lambda(x : \theta \downarrow_E).t' : \tau_1 \to \tau_2$$

Par (4) et par (3), $E \models (\theta \downarrow_E \to s) \preceq (\tau_1 \to \tau_2)$.

▷ *Cas* « App ». La conclusion du typage est $E, \Gamma \vdash t_1\ (t_2) : \tau_2$ et ses prémisses sont $E, \Gamma \vdash t_1 : \tau_1 \to \tau_2$ et $E, \Gamma \vdash t_2 : \tau_1$. La conclusion de l'inférence est $E, \Gamma' \vdash t_1\ (t_2) \Uparrow \mathcal{C}(s) \rightsquigarrow t'_1\ (t'_2)$ et ses prémisses sont $E, \Gamma' \vdash t_1 \Uparrow s \rightsquigarrow t'_1$ et $E, \Gamma' \vdash t_2 \Downarrow \mathcal{D}(s) \rightsquigarrow t'_2$. Par hypothèse d'induction,

$$E, \Gamma \vdash t'_1 : \tau_1 \to \tau_2 \quad (\mathbf{1})$$

et

$$E, \Gamma \vdash t'_2 : \tau_1 \quad (\mathbf{2})$$

Par la règle App,

$$E, \Gamma \vdash t'_1\ (t'_2) : \tau_2$$

$E \models \mathcal{C}(s) \preceq \tau_2$ parce que $E \models s \preceq \tau_1 \to \tau_2$.

La conclusion du mode vérification est $E, \Gamma' \vdash t_1\ (t_2) \Downarrow s \rightsquigarrow t'_1\ (t'_2)$ et ses prémisses sont $E, \Gamma' \vdash t_1 \Uparrow s_1 \rightsquigarrow t'_1$ et $E, \Gamma' \vdash t_2 \Downarrow \mathcal{D}((\bot \to s) \sqcup s_1) \rightsquigarrow t'_2$. s_1 est normalisée dans le système d'équation E. Dès lors, $(\bot \to s) \sqcup s_1$ est définie. De plus, $E \models s \preceq \tau_2$ et $E \models s_1 \preceq \tau_1 \to \tau_2$ implique $E \models \mathcal{C}(s_1 \sqcup \bot \to s) \preceq \tau_2$. Par hypothèse d'induction sur le jugement de typage du terme t_2, nous avons $E, \Gamma \vdash t'_2 : \tau_1$ et la conclusion suit par l'application de la règle App.

▷ *Cas* « Coerce ». La conclusion du typage est $E, \Gamma \vdash (t : \exists\overline{\gamma}.(\tau'_1 \triangleright \tau'_2)) : \tau_2$. Montrons que cela implique $E, \Gamma \vdash (t : \exists\overline{\gamma}.(\tau'_1 \downarrow_E \sqcup \tau'_2 \downarrow_E)) : \tau_2$, on pourra ainsi utiliser la cas de la règle Annot. Les prémisses du typage sont

$$E, \Gamma \vdash t : \tau_1 \quad (\mathbf{1})$$

$$\kappa \preceq (\tau_1 \triangleright \tau_2) \quad (\mathbf{2})$$

et

$$E \models \kappa \quad (\mathbf{3})$$

(2) et (3) donne $E \models \tau_1 = \tau_2$. Ainsi, par (1) et la règle Conv,

$$E, \Gamma \vdash t : \tau_2$$

On a

$$E \models \tau_2 \downarrow_E = \tau_2 \quad (\mathbf{4})$$

Et donc, une autre application de la règle Conv donne

$$E, \Gamma \vdash t : \tau_2 \downarrow_E$$

Par la règle Annot et comme $(\tau'_1 \downarrow_E \sqcup \tau'_2 \downarrow_E) \preceq \tau_2 \downarrow_E$, on a $E, \Gamma \vdash (t : \exists\overline{\gamma}.(\tau'_1 \downarrow_E \sqcup \tau'_2 \downarrow_E)) : \tau_2 \downarrow_E$. Finalement, par la règle Conv et (4),

$$E, \Gamma \vdash (t : \exists\overline{\gamma}.(\tau'_1 \downarrow_E \sqcup \tau'_2 \downarrow_E)) : \tau_2$$

▷ *Cas* « Let ». La conclusion du typage est $E, \Gamma \vdash \mathsf{let}\, x = t_1\ \mathsf{in}\ t_2 : \tau$ et ses hypothèses sont $E, \Gamma \vdash t_1 : \sigma$ et $E, \Gamma; (x : \sigma) \vdash t_2 : \tau$. La conclusion de l'inférence est $E, \Gamma' \vdash \mathsf{let}\, x = t_1\ \mathsf{in}\ t_2 \Uparrow$

$s \rightsquigarrow \mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2'$ et ses hypothèses sont $E, \Gamma' \vdash t_1 \Uparrow s' \rightsquigarrow t_1'$ et $E, \Gamma'; (x : s') \vdash t_2 \Uparrow s \rightsquigarrow t_2'$. Par induction sur la dérivation du terme t_1, il vient

$$E, \Gamma \vdash t_1' : \sigma \tag{1}$$

et $E \models s' \preceq \sigma$. Ainsi, $E \models \Gamma'; (x : s') \preceq \Gamma; (x : \sigma)$ est valide. Par induction sur la dérivation du terme t_2,

$$E, \Gamma \vdash t_2' : \tau \tag{2}$$

et $E \models s \preceq \tau$. Par la règle Let,

$$E, \Gamma \vdash \mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2' : \tau$$

La caractérisation de la forme s a été donné par l'induction sur la dérivation du terme t_2. Le mode vérification se traite de manière similaire.

▷ *Cas* « Fix-$\star$ ». Notons $\varsigma \equiv \exists\overline{\gamma}.\forall\overline{\alpha}.\tau$. La conclusion du typage est $E, \Gamma \vdash \mu^\star(x : \varsigma).t : \sigma$ et ses hypothèses sont $E, \Gamma; (x : \sigma) \vdash t : \sigma$ et

$$\varsigma \preceq \sigma \tag{1}$$

La conclusion de l'inférence est $E, \Gamma' \vdash \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t \Uparrow \overline{\gamma}.\tau \lfloor_E \rightsquigarrow \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau \lfloor_E).t'$ et ses hypothèses sont $\overline{\alpha} \# E, \Gamma', s$ et $E, \Gamma'; x : \overline{\alpha\gamma}.\tau \lfloor_E \vdash t \Downarrow \overline{\gamma}.\tau \lfloor_E \rightsquigarrow t'$. Par (1) et $E \models \Gamma' \preceq \Gamma$, $E \models \Gamma'; (x : (\exists\overline{\gamma}.\forall\overline{\alpha}.\tau) \lfloor_E) \preceq \Gamma; (x : \sigma \lfloor_E)$. Par hypothèse d'induction, $E, \Gamma; (x : \sigma \lfloor_E) \vdash t' : \sigma \lfloor_E$. Par la règle Fix,

$$E, \Gamma \vdash \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau \lfloor_E).t' : \sigma \lfloor_E$$

Par la règle Conv et $E \models \sigma = \sigma \lfloor_E$, il vient

$$E, \Gamma \vdash \mu(x : \varsigma \lfloor_E).t' : \sigma$$

La preuve pour le mode vérification utilise des arguments semblables.

▷ *Cas* « Conv ». La conclusion du typage est $E, \Gamma \vdash t : \tau_2$ et ses prémisses sont $E, \Gamma \vdash t : \tau_1$ et

$$E \models \tau_1 = \tau_2 \tag{1}$$

Si la forme inférée ou attendue s vérifie $E \models s \preceq \tau_1$ alors elle vérifie aussi $E \models s \preceq \tau_2$ grâce à (1). L'hypothèse d'induction fournit le résultat.

▷ *Cas* « Inst ». La conclusion du typage est $E, \Gamma \vdash t : \tau$ et ses hypothèses sont $E, \Gamma \vdash t : \sigma$ et

$$\sigma \preceq \tau \tag{1}$$

Si la forme inférée ou attendue s vérifie $E \models s \preceq \sigma$ alors elle vérifie aussi $E \models s \preceq \tau$ grâce à (1). L'hypothèse d'induction fournit le résultat.

▷ *Cas* « Gen ». La conclusion du typage est $E, \Gamma \vdash t : \forall\overline{\alpha}.\tau$ et ses hypothèses sont $E, \Gamma \vdash t : \tau$ et $\overline{\alpha} \# E, t, \Gamma$. Si la forme inférée ou attendue s vérifie $E \models s \preceq \tau$ alors elle vérifie aussi $E \models s \preceq \forall\overline{\alpha}.\tau$ car les variables libres du type τ sont interprétées comme des variables de type rigides. L'hypothèse d'induction fournit le résultat.

▷ *Cas* « Case ». La conclusion du typage est $E, \Gamma \vdash \mathsf{match}\, t \,\mathsf{with}\, p_1.t_1 \ldots p_n.t_n : \tau$ et ses hypothèses sont $E, \Gamma \vdash t : \tau_1$ et pour tout i, $E, \Gamma \vdash c_i : \tau_1 \to \tau_2$. La conclusion de l'inférence est $E, \Gamma' \vdash \mathsf{match}\, t \,\mathsf{with}\, p_1.t_1 \ldots p_n.t_n \Uparrow \sqcup_i s_i \rightsquigarrow \mathsf{match}\, (t' : s') \,\mathsf{with}\, p_1.t_1' \ldots p_n.t_n'$ et ses hypothèses sont $E, \Gamma' \vdash t \Uparrow s' \rightsquigarrow t'$ et pour tout i, $E, \Gamma' \vdash (p_i : s').t_i \Uparrow s_i \rightsquigarrow p_i.t_i'$. Par hypothèse d'induction, on a

$$E, \Gamma \vdash t' : \tau_1 \tag{1}$$

$$E \models s' \preceq \tau_1 \tag{2}$$

De même, pour tout i, on a

$$E, \Gamma \vdash p_i.t'_i : \tau_1 \to \tau_2 \tag{3}$$

$$E \models s_i \preceq \tau_2 \tag{4}$$

Par la règle Annot, par (1) et par (2), on obtient

$$E, \Gamma \vdash (t' : s') : \tau_1 \tag{5}$$

Ainsi, la règle Case, (5) et (3) conduisent à

$$E, \Gamma \vdash \mathsf{match}\,(t' : s')\,\mathsf{with}\; p_1.t'_1 \ldots p_n.t'_n : \tau_2$$

(4) implique que $E \models \sqcup_i s_i \preceq \tau_2$. Le mode inférence est traité de la même façon.

▷ *Cas* « Clause ». La conclusion de l'inférence est $E_1, \Gamma'_1 \vdash (p : \overline{\gamma}.\varepsilon\,\overline{\tau}'_1\,\overline{\tau}'_2).t \Uparrow s \restriction_{E_1,E'_2} \rightsquigarrow p.t'$ et ses hypothèses sont $p : \overline{\gamma}.\varepsilon\,\overline{\tau_1}'\,\overline{\tau_2}' \vdash (\overline{\beta}, E'_2, \Gamma'_2)$,

$$E_1 \wedge E'_2, \Gamma'_1\Gamma'_2 \vdash\ t \Uparrow s \rightsquigarrow t' \tag{1}$$

et

$$\overline{\beta} \# E_1, s \restriction_{E_1,E'_2}, \varepsilon\,\overline{\tau}'_1\,\overline{\tau}'_2 \tag{2}$$

La conclusion du typage est $E_1, \Gamma_1 \vdash p.t : \varepsilon\,\overline{\tau_1}\,\overline{\tau_2} \to \tau$ et ses hypothèses sont

$$p : \varepsilon\,\overline{\tau_1}\,\overline{\tau_2} \vdash (\overline{\beta}, E_2, \Gamma_2) \tag{3}$$

$$E_1 \wedge E_2, \Gamma_1\Gamma_2 \vdash t : \tau \tag{4}$$

$$\overline{\beta} \# \Gamma_1, \tau_2, E_1 \tag{5}$$

Le choix des variables de type $\overline{\beta}$ identiques pour l'inférence et le typage n'est pas problématique du moment qu'elles vérifient (5) et (2). Par le lemme 8.1.2,

$$E_1 \wedge E_2 \equiv E_1 \wedge E'_2 \tag{6}$$

et

$$E_1 \wedge E_2 \models \Gamma'_2 \preceq \Gamma_2 \tag{7}$$

(6) et (1) impliquent

$$E_1 \wedge E_2, \Gamma'_1\Gamma'_2 \vdash\ t \Uparrow s \rightsquigarrow t' \tag{8}$$

(7) implique

$$E_1 \wedge E_2 \models \Gamma'_1\Gamma'_2 \preceq \Gamma_1\Gamma_2 \tag{9}$$

Par l'hypothèse d'induction impliquée à (8), Par le lemme 8.1.3, (9) et (4), il vient

$$E_1 \wedge E_2, \Gamma_1\Gamma_2 \vdash t' : \tau \tag{10}$$

et $E_1 \wedge E_2 \models s \preceq \tau$. La règle Clause, (10), (3) et (5) impliquent

$$E_1, \Gamma_1 \vdash p.t' : \varepsilon\,\overline{\tau_1}\,\overline{\tau_2} \to \tau$$

et

$$E_1 \wedge E_2 \models s \preceq \tau \tag{11}$$

(11) et (6) impliquent

$$E_1 \wedge E'_2 \models s \preceq \tau \tag{12}$$

(12) implique

$$E_1 \models s \restriction_{E_1, E_2'} \preceq \tau$$

La conclusion du mode vérification est $E_1, \Gamma_1' \vdash (p : \overline{\gamma}.\varepsilon\, \overline{\tau_1}'\, \overline{\tau_2}').t \Downarrow s \rightsquigarrow p.(t' \uparrow_{E_1 \wedge E_2'} s)$ et ses hypothèses sont $p : \overline{\gamma}.\varepsilon\, \overline{\tau_1}'\, \overline{\tau_2}' \vdash (\overline{\beta}, E_2', \Gamma_2')$, $E_1 \wedge E_2', \Gamma_1\Gamma_2' \vdash t \Downarrow s \downarrow_{E_1 \wedge E_2'} \rightsquigarrow t'$ et $\overline{\beta} \# E, \Gamma, s$. Par hypothèse,

$$E_1 \models s \preceq \tau \tag{13}$$

Ainsi,

$$E_1 \wedge E_2 \models s \downarrow_{E_1 \wedge E_2} \preceq \tau \tag{14}$$

L'hypothèse d'induction impliquent

$$E_1 \wedge E_2, \Gamma_1\Gamma_2 \vdash t' : \tau \tag{15}$$

Comme $E_1 \wedge E_2 \models \tau = \tau \downarrow_{E_1 \wedge E_2}$ et (15), la règle Conv conduit à

$$E_1 \wedge E_2, \Gamma_1\Gamma_2 \vdash t' : \tau \downarrow_{E_1 \wedge E_2} \tag{16}$$

(14) et (13) implique

$$(s \downarrow_{E_1 \wedge E_2} \triangleright s) \preceq (\tau \downarrow_{E_1 \wedge E_2} \triangleright \tau)$$

Par la règle Coerce, ceci donne

$$E_1 \wedge E_2, \Gamma_1\Gamma_2 \vdash (t' \uparrow_{E_1 \wedge E_2} s) : \tau$$

□

Le lecteur peut se demander pourquoi nous appelons cette propriété « correction ». Cet énoncé affirme que la forme inférée est plus générale que le type réel alors que la coutume est de dire qu'un algorithme d'inférence de types est « correct » quand le type inféré est moins général que le type réel. Nos algorithmes d'inférence de forme sont conçus pour calculer *une sous-approximation des types* du programme alors qu'un algorithme d'inférence de types habituel en calcule une sur-approximation (ou, si le système de type possède des types principaux, il calcule un type principal). Ceci explique pourquoi nous renversons la terminologie standard. De plus, dire de nos algorithmes d'inférence de formes qu'ils sont complets mais incorrects serait assez déstabilisant.

La correction n'est pas gratuite. Elle s'appuie sur l'élagage qui doit avoir une connaissance totale du système d'équations courant. Cela revient à demander, dans les règles Clause-⇑ et Clause-⇓, que la forme (inférée) du terme analysé soit explicite sur ses paramètres généralisés. Si on abandonne la correction délibérément, on peut choisir de relâcher ces règles en permettant à cette forme d'être incomplète même sur la partie généralisée ce qui engendre un système d'équations plus faible. Cette voie est empruntée par Simon Peyton Jones *et al.* (Peyton Jones *et al*, 2004) dont l'unification « wobbly » peut « effectuer moins de raffinement de type qu'on pourrait justifier dans une version explicitement typé du programme ». D'un côté, comme le système de Simon Peyton Jones *et al.* est capable de travailler avec une sous-approximation du système d'équation, il accepte quelque fois des programmes que nous rejetons. D'un autre côté, comme il n'effectue aucun élagage, il infère parfois des types incorrects comme l'a montré l'exemple du chapitre 7.

Quelle valeur accorder à la correction ? D'un point de vue théorique, il est certainement préférable d'inférer des formes qui sont des approximations de la vérité plutôt que des formes qui peuvent être parfois fausses. D'un point de vue pratique, cependant, le choix n'est pas si clair. Inférer des formes incorrectes conduit à l'insertion d'annotations et de coercions incorrectes dans le terme transformé. Néanmoins, la correction du système complet n'est pas compromise dans la mesure où ces erreurs sont détectées par MLGX. L'expressivité du système n'est pas non plus mise en cause puisque le

programmeur peut toujours passer outre toutes les déficiences de l'inférence locale en insérant des annotations dans le programme source.

En pratique, la seule différence entre les systèmes inférant des formes correctes ou incorrectes apparaît dans la qualité des messages d'erreurs. D'un côté, un système d'inférence de formes correcte rejette légitimement des programmes dans le cas d'une erreur d'unification. Par ailleurs, il insère des annotations et des coercions qui ont du sens ce qui implique que l'inférence de type de MLGX échoue seulement si il n'y a pas assez d'annotations.

D'un autre côté, un système d'inférence de formes incorrecte peut à tort rejeter un programme à cause d'une erreur d'unification à moins que certaines précautions ne soient prises pour qu'il n'échoue pas (voir la section suivante). De plus, une inférence incorrecte à un certain point du programme peut conduire à l'insertion d'annotations et de coercions incorrectes à un point éloigné par le jeu de la propagation des formes. L'erreur de type détectée par MLGX pourrait alors être rapportée à une distance importante par rapport à l'origine réelle du problème. Enfin, un système propagant des formes potentiellement incorrectes peut avoir le comportement de l'exemple du chapitre 7 et accepter « par chance » une définition de fonction parce que ses annotations nomment les variables de type dans l'ordre correspondant à celui choisi en interne. D'une version à l'autre de l'implémentation du typeur, on pourrait perdre le bon typage si cet ordre est modifié pour une raison arbitraire.

8.1.4 Causes de rejet d'un programme par le système *Wob*

Deux raisons peuvent être à l'origine du rejet d'un terme par le système *Wob* :

- Le programme est mal typé dans MLGI. Dans ce cas, le programme sera aussi rejeté par l'algorithme d'inférence de MLGX.
- Le programmeur n'a pas fourni suffisamment d'annotations. Le programme est alors éventuellement bien typé dans MLGI mais sa traduction est mal typée dans MLGX. En particulier, un *pattern matching* n'a pas été annoté suffisamment pour remplir la condition de connaissance totale du système d'équations E.

Dans les deux cas, on peut choisir de ne pas échouer et de laisser l'inférence de types s'occuper de rejeter le programme. Ce procédé prendra tout son sens dans le cadre de l'algorithme présenté dans la section 8.2.

8.1.5 Exemple

Considérons encore l'exemple du chapitre 3, où on a remplacé μ par $\mu^\star$ (la variable de type α est alors liée dans le corps de la fonction).

Traitons ce terme en mode inférence. La règle Fix-$\Uparrow$ passe en mode vérification en utilisant la forme attendue $\mathit{term}\ \alpha \to \alpha$ dans le terme $\lambda t. \ldots$. La règle Lam-$\Downarrow$ détermine que la variable t a la forme $\mathit{term}\ \alpha$ et vérifie le *pattern matching* à l'aide de la forme α. La règle Case-$\Downarrow$ extrait de l'environnement le fait que t a la forme $\mathit{term}\ \alpha$ ce qui permet d'insérer l'annotation de type $(t : \mathit{term}\ \alpha)$ dans le terme élaboré. Chaque branche est examinée à l'aide de la règle Clause-$\Downarrow$ en sachant que la forme du terme analysé est $\mathit{term}\ \alpha$ et que le corps de la branche a une forme attendue α. Dans la branche *Lit*, par exemple, l'équation $\alpha = \mathit{int}$ devient accessible, donc la forme attendue α est normalisée en *int* dès son entrée, et la coercion $(i : (\mathit{int} \triangleright \alpha))$ est insérée. On vérifie alors avec succès que la variable i a la forme *int*. La variable de type α qui apparaît dans les annotations et les coercions nouvellement insérées est liée par la construction $\mu^\star$ donc aucune nouvelle liaison de la forme « $\forall\alpha$ » n'est nécessaire. Les règles Var-$\Uparrow$ et Var-$\Downarrow$ insèrent parfois des coercions redondantes, comme $(\mathit{int} \triangleright \mathit{int})$, qui peuvent facilement être supprimées si besoin. Dans la branche *Pair*, une phase préliminaire (correspondant à la réécriture des sucres syntaxiques) insère localement les deux variables de type β_1 et β_2 associées à ce constructeur de données. Dans le corps de la branche, l'équation $\alpha = \beta_1 \times \beta_2$ est accessible. On

```
let rec eval : ∀α.term α → α = λt.
  match t with
    | Lit (i) → (* α = int *) i
    | Inc (t) → (* α = int *) eval t + 1
    | IsZ (t) → (* α = bool *) eval t = 0
    | If (b, t, e) → if eval b then eval t else eval e
    | Pair (a, b) → (* ∃β1β2.α = β1 × β2 *) (eval a, eval b)
    | Fst (t) → (* ∃β1β2.α = β1 *) fst (eval t)
    | Snd (t) → (* ∃β1β2.α = β1 *) snd (eval t)

let rec eval : ∀α.term α → α = λt.
  match (t : term α) with
    | Lit i → (i : (int ▷ α))
    | Inc t → (eval t + 1 : (int ▷ α))
    | IsZ t → (eval t = 0 : (bool ▷ α))
    | If b t e → if eval b then eval t else eval e
    | Pair β1 β2 a b → ((eval a, eval b) : (β1 × β2 ▷ α))
    | Fst β1 β2 t → fst (eval t)
    | Snd β1 β2 t → snd (eval t)
```

FIG. 8.3: La fonction *eval*, avant et après l'inférence de forme du système *Wob*.

apprend aussi que $a : term\ \beta_1$ et $b : term\ \beta_2$. La forme attendue α est normalisée en $\beta_1 \times \beta_2$ et on insère la coercion $((eval\ a, eval\ b) : (\beta_1 \times \beta_2 \triangleright \alpha))$ (qui est bien valide comme l'indiquent les deux appels récursifs à *eval*, $(eval\ a, eval\ b) : \beta_1 \times \beta_2$).

Ceci explique comment la fonction *eval* est automatiquement transformée en la version annotée du chapitre 4. Le programme transformé est alors accepté par l'inférence de types de MLGX.

8.2 Le système d'inférence de formes Ibis

Bien que le système *Wob* transforme avec brio l'exemple de la fonction *eval* en un terme MLGX bien typé, il souffre cependant d'un défaut dans son traitement de l'application. Comme on l'a déjà mentionné dans la section 8.1, il infère la forme de la fonction et utilise cette information pour examiner l'argument en mode vérification. Notre règle CSTR-⇑ et la règle APPN de Simon Peyton Jones *et al.* (2004) font un choix opposé et examinent les arguments en mode inférence. En fait, ces deux choix sont *ad hoc*. Idéalement, l'information de forme doit pouvoir être véhiculée de la fonction aux arguments et *vice et versa*. Nous décrivons maintenant un système d'inférence de formes qui subsume le système *Wob* et est conçu pour permettre la propagation dans les deux sens. Le moyen le plus simple pour implémenter cette propagation symétrique est d'effectuer deux passes sur l'ensemble du programme. Le système obtenu est appelé *Ibis*, en référence à son caractère itéré, bidirectionnel et symétrique.

8.2.1 Présentation informelle

Exemple 8.2.1

Le terme suivant illustre le défaut du système *Wob*. Il s'agit de l'utilisation de la fonction standard *map* pour appliquer sur une liste une fonction anonyme faisant usage d'un GADT. Nous supposons que le constructeur de donnée I a le type $ty\ int$ de manière à ce que l'équation $\alpha = int$ soit disponible à l'intérieur du *pattern matching*.

$$\mu^\star(double : \forall\alpha.ty\ \alpha \to list\ \alpha \to list\ \alpha).\lambda t.\lambda l.\\ map\ (\lambda x.\mathsf{match}\ t\ \mathsf{with}\ I \to x + x)\ (l)$$

Trois coercions explicites sont nécessaires pour transformer ce programme en un programme MLGX bien typé. À chaque occurrence de la variable x, la variable de type α doit être convertie en *int*. De plus, l'expression $x + x$ doit être coercée de *int* vers α de façon à satisfaire les annotations fournies par le programmeur qui obligent le type de retour de la fonction anonyme à être α.

Malheureusement, le système *Wob* n'est pas capable d'insérer une seule de ces coercions. Le terme contient une application double de *map*. Le système *Wob* commence le traitement de l'application la plus externe en mode vérification avec le type attendu *list* α. Par la règle App-$\Downarrow$, cela nécessite d'inférer d'abord une forme pour l'application interne *map* $((\lambda x. \ldots))$ et ensuite de vérifier l'argument l. Par conséquent, la règle App-$\Uparrow$ est appliquée à l'application interne. La forme inférée pour *map* est $\gamma_1\gamma_2.(\gamma_1 \to \gamma_2) \to \mathit{list}\ \gamma_1 \to \mathit{list}\ \gamma_2$. Cela conduit à la vérification de $\lambda x. \ldots$ avec la forme $\gamma_1\gamma_2.\gamma_1 \to \gamma_2$. Cette forme imprécise ne fournit pas assez d'informations sur le type de x ou sur le type de retour de la fonction anonyme. Le système *Wob* est donc incapable d'insérer les bonnes coercions dans cette fonction.

Dans cette exemple, la « bonne » chose à faire est d'examiner d'abord l'argument l en mode inférence ce qui donnerait une forme *list* α et d'exploiter cette information pour examiner l'application interne en mode vérification avec la forme attendue *list* $\alpha \to$ *list* α. La « bonne » chose à faire ensuite est d'examiner *map* en mode inférence, comme son type est connu, et d'exploiter cette information en mode vérification sur $\lambda x. \ldots$. On voit ici clairement que tout choix fixé d'une propagation gauche-droite ou droite-gauche sur les applications est une mauvaise idée.

Pour éviter un tel dilemme, nous suggérons de traiter l'application (et, plus généralement, les constructions binaires) de manière plus symétrique. Cela devient possible en découpant l'inférence de formes en deux passes. En première approximation, l'idée est d'examiner les fonctions et les arguments en mode inférence lors de la première passe et de les examiner en mode vérification lors de la seconde passe. La forme inférée pour la fonction pendant la première passe est utilisée pour la vérification des arguments lors de la seconde et réciproquement. L'information est ainsi propagée dans les deux sens.

Une autre idée indépendante est d'abandonner la distinction entre les modes « inférence » et « vérification ». En effet, quand nous sommes en mode inférence, pourquoi refuser de tirer profit des informations de type offertes par le contexte par le biais d'une forme attendue ? Réciproquement, quand nous sommes en mode vérification, pourquoi refuser de produire une forme inférée qui pourrait être plus précise que la forme attendue initialement ? Notre solution est d'effectuer l'inférence et la vérification en même temps. Les jugements du système *Ibis* ont la forme

$$E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'$$

où la forme inférée s', un paramètre de sortie, est toujours au moins aussi précise que la forme attendue s, un paramètre d'entrée. Autrement dit, on a $s \preceq s'$. Chacune de ses formes est normalisée par rapport à E. En deux mots, *Ibis* est bidirectionnel, comme le système *Wob*, mais s'exécute dans les deux sens simultanément. Ce processus rappelle l'inférence locale colorée (Odersky *et al*, 2001) où une partie d'un même type est inférée tandis qu'une autre est issue de la propagation des annotations de type (et donc seulement vérifiée).

Notre premier énoncé approximatif affirmant que la première passe travaille en mode inférence alors que la seconde travaille en mode vérification doit alors être précisé. Les deux passes sont en fait identiques — elles sont définies par le même ensemble de règles — et toutes les deux effectuent simultanément la vérification et l'inférence. La première passe annote les sous-termes immédiats des nœuds d'application avec les formes inférées. La seconde passe exploite ces annotations pour inférer des formes plus précises.

Par conséquent, le système *Ibis* consiste réellement en une unique passe qui peut être itérée autant de fois que nécessaire. En pratique, itérer au moins deux fois est nécessaire pour que les informations soient propagées des fonctions vers les arguments et *vice et versa*. En outre, nous pensons qu'itérer

deux fois est suffisant pour permettre d'obtenir une précision suffisante. Nous ne cherchons donc pas à itérer tant qu'un point fixe n'est pas atteint. Cela permettrait d'accepter plus de programmes mais cela remettrait peut-être l'apparente simplicité et prédictabilité de l'algorithme. Ce nombre arbitrairement grand d'itérations mettrait aussi en cause la complexité linéaire de notre approche stratifiée.

8.2.2 Formalisation

$$\textsc{I-Lam}\quad \frac{s' = s \sqcup (\theta\downarrow_E \to \bot) \qquad E, \Gamma; x : \mathcal{D}(s') \vdash t \Downarrow \mathcal{C}(s') \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash \lambda(x:\theta).t \Downarrow s \Uparrow (s' \sqcup (\bot \to s'')) \rightsquigarrow \lambda(x:\theta\downarrow_E).t'}$$

$$\textsc{I-App}\quad \frac{s' = s_1 \sqcup (s_2 \to s) \qquad E, \Gamma \vdash t_1 \Downarrow s' \Uparrow s'_1 \rightsquigarrow t'_1 \qquad E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s'_1) \Uparrow s'_2 \rightsquigarrow t'_2}{E, \Gamma \vdash (t_1 : s_1)\ ((t_2 : s_2)) \Downarrow s \Uparrow \mathcal{C}(s'_1 \sqcup (s'_2 \to \bot)) \rightsquigarrow (t'_1 : s'_1)\ ((t'_2 : s'_2))}$$

$$\textsc{I-Let}\quad \frac{E, \Gamma \vdash t_1 \Downarrow s_1 \Uparrow s'_1 \rightsquigarrow t'_1 \qquad E, \Gamma; x : s'_1 \vdash t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow t'_2}{E, \Gamma \vdash \mathsf{let}\, x\ =\ (t_1 : s_1)\ \mathsf{in}\ t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow \mathsf{let}\, x\ =\ (t'_1 : s'_1)\ \mathsf{in}\ t'_2}$$

$$\textsc{I-Fix}\quad \frac{\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma, s) \qquad E, \Gamma; x : (\overline{\gamma\alpha}.\tau\downarrow_E \sqcup s) \vdash t \Downarrow (\overline{\gamma}.\tau\downarrow_E \sqcup s) \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau).t \Downarrow s \Uparrow \overline{\alpha}.s'' \rightsquigarrow \mu^\star(x : \exists\overline{\gamma}.\forall\overline{\alpha}.\tau\downarrow_E).t'}$$

$$\textsc{I-Case}\quad \frac{E, \Gamma \vdash t \Downarrow s' \Uparrow s'' \rightsquigarrow t' \qquad \forall i \quad E, \Gamma \vdash (p_i : s'').t_i \Downarrow s \Uparrow s_i \rightsquigarrow p_i.t'_i}{E, \Gamma \vdash \mathsf{match}\,(t : s')\ \mathsf{with}\ p_1.t_1 \ldots p_n.t_n \Downarrow s \Uparrow \sqcup_i s_i \rightsquigarrow \mathsf{match}\,(t' : s'')\ \mathsf{with}\ p_1.t'_1 \ldots p_n.t'_n}$$

$$\textsc{I-Forall}\quad \frac{E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t' \qquad \overline{\alpha} \# \mathbf{ftv}\,(\Gamma, E, s)}{E, \Gamma \vdash \forall\overline{\alpha}.t \Downarrow s \Uparrow \overline{\alpha}.s' \rightsquigarrow \forall\overline{\alpha}.t'}$$

$$\textsc{I-Annot}\quad \frac{E, \Gamma \vdash t \Downarrow (\theta\downarrow_E \sqcup s) \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash (t : \theta) \Downarrow s \Uparrow s'' \rightsquigarrow (t' : \theta\downarrow_E)}$$

$$\textsc{I-Var}\quad \frac{(x : s') \in \Gamma}{E, \Gamma \vdash x \Downarrow s \Uparrow (s \sqcup s'\downarrow_E) \rightsquigarrow (x \downarrow_E s')}$$

$$\textsc{I-Coerce}\quad \frac{E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'}{E, \Gamma \vdash (t : \kappa) \Downarrow s \Uparrow s' \rightsquigarrow t'}$$

$$\textsc{I-Clause}\quad \frac{p : \varepsilon\,\overline{\tau}_1\,\overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\overline{\gamma}.\Gamma') \vdash t \Downarrow s\downarrow_{E\wedge E'} \Uparrow s' \rightsquigarrow t' \qquad s'' = s'\uparrow_{E,E'} \sqcup s \qquad \overline{\beta} \# \mathbf{ftv}\,(E, \Gamma, s'') \qquad \overline{\gamma} \# \mathbf{ftv}\,(E, \Gamma, \overline{\tau}_2, t, s)}{E, \Gamma \vdash (p : \overline{\gamma}.\varepsilon\,\overline{\tau}_1\,\overline{\tau}_2).t \Downarrow s \Uparrow s'' \rightsquigarrow p.(t' \uparrow_{E\wedge E'} s'')}$$

FIG. 8.4: L'inférence de formes pour le système *Ibis*.

Les règles qui définissent le système *Ibis* sont données par la figure 8.4. La règle I-APP s'attend à trouver sur la fonction t_1 et sur l'argument t_2 des annotations explicites de type s_1 et s_2.

Si on ne trouve pas de telles annotations alors la forme $\bot$ est utilisée. En pratique, il n'y a en général aucune annotation avant la première passe. Les formes inférées s'_1 et s'_2 sont enregistrées sous la forme d'annotations de type de manière à ce qu'elles soient exploitées par la passe suivante si il y

en a une.

Par hypothèse, il n'y a pas de coercion dans le programme source. La règle I-Coerce affirme que chaque passe supprime les coercions insérées par la passe précédente. En effet, comme chaque nouvelle passe calcule une forme plus précise que la passe précédente, elle peut engendrer des coercions plus précises.

Nous ne donnons pas une explication détaillée des autres règles. Dans la plupart des cas, les variantes des deux modes du système *Wob* sont superposées pour produire une règle qui effectue à la fois l'inférence et la vérification.

Exemple 8.2.2
Considérons l'exemple de la fonction *double* à nouveau :

$$\begin{array}{l}\mu^\star(\mathit{double} : \forall\alpha.\mathit{ty}\ \alpha \to \mathit{list}\ \alpha \to \mathit{list}\ \alpha).\lambda t.\lambda l.\\ \quad \mathit{map}\ ((\lambda x.\mathsf{match}\ t\ \mathsf{with}\ I \to x + x))\ (l)\end{array}$$

La première passe de l'algorithme *Ibis* débute par l'application englobante avec la forme attendue $\mathit{list}\ \alpha$. Tout d'abord, elle examine sa partie gauche, c'est-à-dire l'application interne, avec la forme attendue $\bot \to \mathit{list}\ \alpha$. La fonction *map* est alors traitée avec la forme attendue $\bot \to \bot \to \mathit{list}\ \alpha$. La règle I-Var combine cette forme avec celle de *map*, déjà connue précisément par l'environnement de forme, ce qui engendre la forme inférée $\gamma.(\gamma \to \alpha) \to \mathit{list}\ \gamma \to \mathit{list}\ \alpha$. Le sous-terme *map* est annoté par cette forme. Alors, la fonction anonyme est traitée avec la forme attendue $\gamma.\gamma \to \alpha$. Le *pattern matching* est examiné avec la forme attendue α ce qui conduit la règle I-Clause à insérer une coercion de *int* vers α autour de la branche. Le résultat de la première passe est le terme :

$$\begin{array}{l}\mu^\star(\mathit{double} : \forall\alpha.\mathit{ty}\ \alpha \to \mathit{list}\ \alpha \to \mathit{list}\ \alpha).\lambda t.\lambda l.\\ \quad \big((\mathit{map} : \gamma.(\gamma \to \alpha) \to \mathit{list}\ \gamma \to \mathit{list}\ \alpha)\ (\\ \quad\ ((\lambda x.\mathsf{match}\ t\ \mathsf{with}\ I \to (x + x : \mathit{int} \triangleright \alpha)) : \gamma.\gamma \to \alpha))\\ \quad\ : \gamma.\mathit{list}\ \gamma \to \mathit{list}\ \alpha\big)\ (\\ \quad (l : \mathit{list}\ \alpha))\end{array}$$

Ici, le système *Ibis* a fait un travail à peine plus poussé que celui de *Wob* : seule une coercion a été insérée. Cette amélioration est due à la capacité du système *Ibis* à effectuer l'inférence et la vérification simultanément. Quand le système *Wob* commence à traiter l'application interne en mode inférence, il oublie tout de la forme attendue $\bot \to \mathit{list}\ \alpha$.

Néanmoins, le terme est annoté par une information que nous pouvons exploiter au sein d'une nouvelle passe.

Dans la seconde passe, sur l'application externe, la forme $\gamma.\mathit{list}\ \gamma \to \mathit{list}\ \alpha$ inférée pour la fonction est combinée avec la forme $\mathit{list}\ \alpha$ inférée pour l'argument. Cela conduit à l'examen de l'application interne avec la forme attendue $\mathit{list}\ \alpha \to \mathit{list}\ \alpha$. Cette information nous permet de déterminer que *map* est utilisée au type $(\alpha \to \alpha) \to \mathit{list}\ \alpha \to \mathit{list}\ \alpha$. La fonction $\lambda x.\ldots$ est considérée avec la forme attendue $\alpha \to \alpha$. La seconde passe du système *Ibis* est maintenant capable de déterminer que la variable x a la forme α ce qui permet à la règle I-Var d'insérer les coercions de α vers *int* sur chacune des utilisations de la variable x. La coercion de *int* vers α qui avait été insérée par la première passe est effacée et une coercion identique est ré-insérée.

Le terme produit par la seconde passe est alors :

$$\begin{array}{l}\mu^\star(\mathit{double} : \forall\alpha.\mathit{ty}\ \alpha \to \mathit{list}\ \alpha \to \mathit{list}\ \alpha).\lambda t.\lambda l.\\ \quad \big((\mathit{map} : (\alpha \to \alpha) \to \mathit{list}\ \alpha \to \mathit{list}\ \alpha)\ (\\ \quad\ ((\lambda x.\mathsf{match}\ t\ \mathsf{with}\\ \qquad\qquad I \to ((x : \alpha \triangleright \mathit{int}) + (x : \alpha \triangleright \mathit{int}) : \mathit{int} \triangleright \alpha))\\ \quad\ : \alpha \to \alpha))\\ \quad\ : \mathit{list}\ \alpha \to \mathit{list}\ \alpha\big)\ (\\ \quad (l : \mathit{list}\ \alpha))\end{array}$$

Ce terme est bien typé dans MLGX ce qui signifie que cette définition de la fonction *double* est acceptée par le système d'inférence stratifiée combinant le système *Ibis* et MLGX.

On montre maintenant la correction de l'algorithme d'inférence locale du système *Ibis* à l'aide d'un énoncé similaire au théorème 8.1.5.

Théorème 8.2.3 (Correction du système *Ibis*)
Soit $E, \Gamma \vdash t : \sigma$ *valide dans MLGI. Supposons* $E \models \Gamma' \preceq \Gamma$. *Si* $E, \Gamma' \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'$ *est dérivable dans le système* Ibis, $E \models s \preceq \sigma$, *et* s *est normalisée par rapport à* E, *alors* $E \models s' \preceq \sigma$ *et* $E, \Gamma \vdash t' : \sigma$ *est dérivable dans MLGI. De plus, la forme* s' *est normalisée par rapport à* E *et* $s \preceq s'$. ◇

Preuve. La preuve se fait par induction sur les dérivations de MLGI. Dans cette démonstration, on dira qu'une forme est normalisée si elle est normalisée par rapport au système d'équations E. Cette preuve s'appuie très largement sur la preuve de correction du système *Wob* car les règles du système *Ibis* sont très souvent une simple superposition du mode inférence et du mode vérification du système *Wob*.

▷ *Cas* « Var ». Le jugement de typage est $E, \Gamma \vdash x : \sigma$ et sa prémisse est $(x : \sigma) \in \Gamma$. Le jugement du système *Ibis* est $E, \Gamma' \vdash x \Downarrow s \Uparrow (s \sqcup s' \lfloor_E) \rightsquigarrow (x \downarrow_E s')$ et sa prémisse est $(x : s') \in \Gamma'$. Pour les mêmes raisons que dans la preuve du théorème 8.1.5, $E, \Gamma \vdash (x \downarrow_E s') : \sigma$ et

$$E \models s' \preceq \sigma \qquad (\mathbf{1}).$$

Par hypothèse,

$$E \models s \preceq \sigma \qquad (\mathbf{2})$$

(1) et (2) implique donc que la forme $s'' \equiv (s \sqcup s' \lfloor_E)$ est telle que $E \models s'' \preceq \sigma$ par application du lemme 7.2.4. Par ailleurs, on a clairement $s \preceq s''$ et s'' est normalisée puisque s l'est par hypothèse.

▷ *Cas* « Lam ».La conclusion du typage est $E, \Gamma \vdash \lambda(x : \theta).t : \tau_1 \to \tau_2$ et ses prémisses sont $E, (\Gamma; x : \tau_1) \vdash t : \tau_2$ et $\theta \preceq \tau_1$. Le jugement du système *Ibis* est $E, \Gamma' \vdash \lambda(x : \theta).t \Downarrow s \Uparrow (s' \sqcup (\bot \to s'')) \rightsquigarrow \lambda(x : \theta \lfloor_E).t'$ et ses prémisses sont $s' \equiv s \sqcup (\theta \lfloor_E \to \bot)$ et $E, \Gamma'; x : \mathcal{D}(s') \vdash t \Downarrow \mathcal{C}(s') \Uparrow s'' \rightsquigarrow t'$. La forme s' est normalisée puisque la forme s l'est par hypothèse. Son domaine $\mathcal{C}(s')$ est donc normalisée. De même, $E \models s' \preceq \tau_1 \to \tau_2$ donc $E \models \mathcal{D}(s') \preceq \tau_1$. On a donc bien $\Gamma'; (x : \mathcal{D}(s')) \preceq \Gamma; (x : \tau_1)$ et on peut donc appliquer l'hypothèse d'induction sur la dérivation de typage du terme t. Pour les mêmes raisons que dans la preuve du théorème 8.1.5, on a $E, \Gamma \vdash \lambda(x : \theta \lfloor_E).t' : \sigma$. La forme inférée est quant à elle plus précise que dans le système *Wob* : la preuve du théorème 8.1.5 en mode vérification nous apprend que

$$E \models s' \preceq \sigma \qquad (\mathbf{1})$$

la preuve en mode inférence nous apprend que $E \models \theta \lfloor_E \to s'' \preceq \sigma$ donc

$$E \models \bot \to s'' \preceq \sigma \qquad (\mathbf{2})$$

(1) et (2) impliquent $E \models (s' \sqcup \bot \to s'') \preceq \sigma$. Puisque $s' \equiv s \sqcup (\theta \lfloor_E \to \bot)$ et $\mathcal{C}(s) \preceq s''$, on a $s \preceq (s' \sqcup \bot \to s'')$. De même, l'hypothèse d'induction implique que la forme s'' est normalisée. Ainsi, la forme $s' \sqcup \bot \to s''$ est normalisée.

▷ *Cas* « App ». Le jugement de typage est $E, \Gamma \vdash (t_1 : s_1)\ ((t_2 : s_2)) : \tau_2$ et ses prémisses sont $E, \Gamma \vdash (t_1 : s_1) : \tau_1 \to \tau_2$ et $E, \Gamma \vdash (t_2 : s_2) : \tau_1$. Le jugement du système *Ibis* est $E, \Gamma' \vdash (t_1 : s_1)\ ((t_2 : s_2)) \Downarrow s \Uparrow \mathcal{C}(s'_1 \sqcup (s'_2 \to \bot)) \rightsquigarrow (t'_1 : s'_1)\ ((t'_2 : s'_2))$ et ses prémisses sont $s' \equiv s_1 \sqcup (s_2 \to s)$, $E, \Gamma' \vdash t_1 \Downarrow s' \Uparrow s'_1 \rightsquigarrow t'_1$ et $E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s'_1) \Uparrow s'_2 \rightsquigarrow t'_2$. $E \models s' \preceq \tau_1 \to \tau_2$ permet d'appliquer l'hypothèse d'induction à la dérivation de typage du terme t_1, on a $E, \Gamma \vdash t'_1 : \tau_1 \to \tau_2$ et $E \models s'_1 \preceq \tau_1 \to \tau_2$. Par application de la règle Annot sur ces deux hypothèses, on a

$$E, \Gamma \vdash (t'_1 : s'_1) : \tau_1 \to \tau_2 \qquad (\mathbf{1})$$

Comme $E \models s'_1 \preceq \tau_1 \to \tau_2$, $E \models \mathcal{D}(s'_1) \preceq \tau_1$. Ceci permet d'appliquer l'hypothèse d'induction à la dérivation de typage du terme t_2, on a $E, \Gamma \vdash t'_2 : \tau_1$ et $E \models s'_2 \preceq \tau_1$. Par application de la règle Annot,

$$E, \Gamma \vdash (t'_2 : s'_2) : \tau_1 \tag{2}$$

Par application de la règle App sur (1) et (2), on a $E, \Gamma \vdash (t'_1 : s'_1)\ ((t'_2 : s'_2)) : \tau_2$. De plus, on a $E \models s'_1 \preceq \tau_1 \to \tau_2$ et $E \models s'_2 \preceq \tau_1$ donc $E \models s'_1 \sqcup s'_2 \to \bot \preceq \tau_1 \to \tau_2$ donc $E \models \mathcal{C}(s'_1 \sqcup s'_2 \to \bot) \preceq \tau_2$. Par hypothèse d'induction, les formes s'_1 et s'_2 sont normalisées par rapport à E donc la forme inférée l'est aussi. Enfin, l'induction nous informe que $s' \preceq s'_1$ donc $s_2 \to s \preceq s'_1$. Étant donné que l'induction nous donne aussi $\mathcal{D}(s'_1) \preceq s'_2$, on obtient $\mathcal{D}(s'_1) \to \bot \preceq s'_2 \to \bot$. Il vient alors que $s_2 \to s \preceq s'_1 \sqcup (s'_2 \to \bot)$ donc $s \preceq \mathcal{C}(s'_1 \sqcup (s'_2 \to \bot))$.

▷ *Cas* « Let ». La conclusion du typage est $E, \Gamma \vdash \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 : \tau$ et ses prémisses sont $E, \Gamma \vdash t_1 : \sigma$ ainsi que $E, (\Gamma; x : \sigma) \vdash t_2 : \tau$. La conclusion du système *Ibis* est $E, \Gamma' \vdash \mathsf{let}\, x = (t_1 : s_1) \,\mathsf{in}\, t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow \mathsf{let}\, x = (t'_1 : s'_1) \,\mathsf{in}\, t'_2$ et ses prémisses sont $E, \Gamma' \vdash t_1 \Downarrow s_1 \Uparrow s'_1 \rightsquigarrow t'_1$ et $E, \Gamma'; x : s'_1 \vdash t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow t'_2$. Par hypothèse d'induction appliquée à la dérivation de typage du terme t_1, on a $E, \Gamma \vdash t'_1 : \sigma$, $E \models s'_1 \preceq \sigma$. Ses deux hypothèses et la règle Annot donnent $E, \Gamma \vdash (t'_1 : s'_1) : \sigma$. On a $E \models \Gamma; (x : \sigma) \preceq \Gamma'; (x : s')$ qui permet d'appliquer l'hypothèse d'induction à la dérivation de typage du terme t_2. On a alors $E, \Gamma \vdash t'_2 : \tau$ et $E \models s'_2 \preceq \tau$. Par application de la règle Let, $E, \Gamma \vdash \mathsf{let}\, x = (t'_1 : s'_1) \,\mathsf{in}\, t'_2 : \tau$. L'hypothèse d'induction appliquée à t_2 fournit immédiatement les propriétés voulues sur s'_2.

▷ *Cas* « Forall ». La conclusion du typage est $E, \Gamma \vdash \forall \overline{\alpha}.t : \forall \overline{\alpha}.\tau$ et les prémisses sont $E, \Gamma \vdash t : \tau$ et

$$\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma) \tag{1}$$

La conclusion du système *Ibis* est $E, \Gamma' \vdash \forall \overline{\alpha}.t \Downarrow s \Uparrow \overline{\alpha}.s' \rightsquigarrow \forall \overline{\alpha}.t'$ et ses prémisses sont $E, \Gamma' \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'$ et $\overline{\alpha} \# \mathbf{ftv}\,(\Gamma', E, s)$. Par application de l'hypothèse d'induction sur la dérivation de typage du terme t, il vient $E, \Gamma \vdash t' : \tau$ et $E \models s' \preceq \tau$. Par (1) et une application de la règle Forall, on a $E, \Gamma \vdash \forall \overline{\alpha}.t' : \forall \overline{\alpha}.\tau$. L'hypothèse d'induction fournit la caractérisation de la forme s'.

▷ *Cas* « Annot ». La conclusion du typage est $E, \Gamma \vdash (t : \theta) : \tau$ et ses prémisses sont $E, \Gamma \vdash t : \tau$ et $\theta \preceq \tau$. La conclusion du système *Ibis* est $E, \Gamma' \vdash (t : \theta) \Downarrow s \Uparrow s'' \rightsquigarrow (t' : \theta \lfloor_E)$ et ses prémisses sont $E, \Gamma' \vdash t \Downarrow (\theta \lfloor_E \sqcup s) \Uparrow s'' \rightsquigarrow t'$. Comme la forme s est normalisée par hypothèse, la forme $(\theta \lfloor_E \sqcup s)$ est aussi normalisée donc on peut appliquer l'hypothèse d'induction à la dérivation de typage du terme t et on obtient

$$E, \Gamma \vdash t' : \tau \tag{1}$$

, $(\theta \lfloor_E \sqcup s) \preceq s''$ et $E \models s'' \preceq \tau$. Ses deux dernières propriétés induisent $E \models \theta \lfloor_E \preceq \tau$ qui, conjointement à (1), impliquent $E, \Gamma \vdash (t' : \theta \lfloor_E) : \tau \lfloor_E$ par application de la règle Annot. Par application de la règle Conv, $E, \Gamma \vdash (t' : \theta \lfloor_E) : \tau$. Les propriétés de la forme s'' découlent directement de l'hypothèse d'induction appliquée au jugement du terme t.

▷ *Cas* « Fix ». La conclusion du jugement de typage est $E, \Gamma \vdash \mu^\star(x : \varsigma).t : \sigma$ et ses prémisses sont $E, (\Gamma; x : \sigma) \vdash t : \sigma$ et

$$\varsigma \preceq \sigma \tag{1}$$

La conclusion du jugement du système *Ibis* est $E, \Gamma' \vdash \mu^\star(x : \exists \overline{\gamma}.\forall \overline{\alpha}.\tau).t \Downarrow s \Uparrow \overline{\alpha}.s'' \rightsquigarrow \mu^\star(x : \exists \overline{\gamma}.\forall \overline{\alpha}.\tau \lfloor_E).t'$ et ses prémisses sont $\overline{\alpha} \# \mathbf{ftv}\,(E, \Gamma, s)$ et $E, \Gamma'; (x : (\overline{\gamma \alpha}.\tau \lfloor_E \sqcup s)) \vdash t \Downarrow (\overline{\gamma}.\tau \lfloor_E \sqcup s) \Uparrow s'' \rightsquigarrow t'$. Par (1), $E \models \varsigma \lfloor_E \preceq \sigma$ et donc $\varsigma \lfloor_E \preceq \sigma \lfloor_E$. La forme $(\overline{\gamma}.\tau \lfloor_E \sqcup s)$ est normalisée donc on peut appliquer l'hypothèse d'induction à la dérivation de typage du terme t. On a alors $E, \Gamma \vdash t' : \sigma$ ainsi que

$$E \models s'' \preceq \sigma \tag{2}$$

et

$$s \preceq s'' \tag{3}$$

On peut appliquer la règle Fix pour en déduire $E, \Gamma \vdash \mu^\star(x : \varsigma \lfloor_E).t : \sigma \lfloor_E$. Par application de la règle Conv, il vient $E, \Gamma \vdash \mu^\star(x : \varsigma \lfloor_E).t : \sigma$. De (2), et en renommant éventuellement les variables de type $\overline{\alpha}$ telles que $\overline{\alpha} \# \mathbf{ftv}(\sigma)$, on a $E \models \overline{\alpha}.s'' \preceq \sigma$. De même, en l'appliquant sur (3), on obtient $s \preceq \overline{\alpha}.s''$.

▷ *Cas* « Case ». Le jugement de typage est $E, \Gamma \vdash \mathsf{match}\, t \,\mathsf{with}\, c_1 \dots c_n : \tau_2$ et ses prémisses sont $E, \Gamma \vdash t : \tau_1$ et $\forall i \quad E, \Gamma \vdash c_i : \tau_1 \to \tau_2$. La conclusion de la dérivation dans le système *Ibis* est $E, \Gamma' \vdash \mathsf{match}\,(t : s')\,\mathsf{with}\, p_1.t_1 \dots p_n.t_n \Downarrow s \Uparrow \sqcup_i s_i \rightsquigarrow \mathsf{match}\,(t' : s'')\,\mathsf{with}\, p_1.t'_1 \dots p_n.t'_n$ et ses prémisses sont $E, \Gamma' \vdash t \Downarrow s' \Uparrow s'' \rightsquigarrow t'$ et $\forall i \quad E, \Gamma' \vdash (p_i : s'').t_i \Downarrow s \Uparrow s_i \rightsquigarrow p_i.t'_i$. Par le même raisonnement que le cas Annot, on a $E, \Gamma \vdash (t' : s'') : \tau_1$. Par application de l'hypothèse d'induction à chacune des branches, pour tout i, $E, \Gamma \vdash (p_i : s'').t_i : \tau_1 \to \tau_2$, $E \models s_i \preceq \tau_2$ et

$$s \preceq s_i \tag{1}$$

La première propriété implique que $E, \Gamma \vdash \mathsf{match}\,(t' : s'')\,\mathsf{with}\, p_1.t'_1 \dots p_n.t'_n : \tau_2$ par application de la règle Case. La seconde propriété implique que la forme $\sqcup_i s_i$ est définie et que la propriété $E \models \sqcup_i s_i \preceq \tau_2$ est validee. Par (1), on a aussi $\sqcup_i s_i \preceq \tau_2$.

▷ *Cas* « Clause ». Le jugement de typage est $E, \Gamma \vdash p.t : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \to \tau_2$ et ses prémisses sont $p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, E', \Gamma'')$, $E \wedge E', \Gamma\Gamma'' \vdash t : \tau_2$ et $\overline{\beta} \# \mathbf{ftv}(E, \Gamma, \tau_2)$. Le jugement du système *Ibis* est $E, \Gamma' \vdash (p : \overline{\gamma}.\varepsilon\, \overline{\tau}_1\, \overline{\tau}_2).t \Downarrow s \Uparrow s'' \rightsquigarrow p.(t' \uparrow_{E \wedge E'} s'')$ et ses prémisses sont $p : \varepsilon\, \overline{\tau}_1\, \overline{\tau}_2 \vdash (\overline{\beta}, E'', \Gamma''')$, $E \wedge E'', \Gamma'(\overline{\gamma}.\Gamma''') \vdash t \Downarrow s \lfloor_{E \wedge E'} \Uparrow s' \rightsquigarrow t', s'' = s' \lceil_{E,E''} \sqcup s, \overline{\beta} \# \mathbf{ftv}(E, \Gamma', s'')$ et $\overline{\gamma} \# \mathbf{ftv}(E, \Gamma', \overline{\tau}_2, t, s)$. Tout d'abord, le lemme 8.1.2 affirme que $E' \equiv E''$ et $E \wedge E' \models \Gamma'' \preceq \Gamma'''$ puisqu'on a exigé que les annotations sur les branches soient rigides sur les paramètres généralisés. Par hypothèse d'induction appliquée à la dérivation de typage du terme t, il vient que

$$E \wedge E', \Gamma\Gamma' \vdash t' : \tau_2 \tag{1}$$

$$E \wedge E' \models s' \preceq \tau_2 \tag{2}$$

$$s \lfloor_{E \wedge E'} \preceq s' \tag{3}$$

et s' est normalisée par rapport au système d'équations $E \wedge E'$. Le lemme 7.3.7 appliqué à (2) induit $E \models s' \lceil_{E,E'} \preceq \tau_2$ donc $E \models s'' \preceq \tau_2$ puisque $E \models s \preceq \tau_2$. Il vient $E \models s'' \lfloor_{E \wedge E'} = \tau_2$. L'application de la règle Coerce est donc valide et on obtient $E, \Gamma \vdash (t' \uparrow_{E \wedge E'} s'') : \tau_2$. Par application de la règle Clause et à l'aide des hypothèses sur le motif inchangé, il vient $E, \Gamma \vdash p.(t' \uparrow_{E \wedge E'} s'') : \tau_2$. Enfin, $s' \lceil_{E,E'}$ est normalisée par rapport à E puisque s' est normalisée par rapport à $E \wedge E'$. $s \sqcup s' \lceil_{E,E'}$ est bien définie et normalisée par rapport à E. On a par ailleurs que $s \preceq s \sqcup s' \lceil_{E,E'}$ par définition.

▷ *Cas* « Inst ». Le jugement de typage est $E, \Gamma \vdash t : \tau$ et ses prémisses sont $E, \Gamma \vdash t : \sigma$ et $\sigma \preceq \tau$. L'hypothèse d'induction peut s'appliquer directement sur le terme t avec un schéma de type plus général. Ses conclusions sont donc immédiatement suffisantes.

▷ *Cas* « Gen ». Le jugement de typage est $E, \Gamma \vdash t : \forall \overline{\alpha}.\tau$ et ses prémisses sont $E, \Gamma \vdash t : \tau$ et $\overline{\alpha} \# \mathbf{ftv}(E, \Gamma, t)$. Le résultat est immédiat par utilisation de l'hypothèse d'induction.

▷ *Cas* « Conv ». Le raisonnement est immédiat puisqu'on demande une forme inférée valide *modulo* le système d'équations E.

▷ *Cas* « Cstr ». Immédiat car c'est une instance de la règle App où on connait parfaitement le membre gauche de l'application. □

8.2.3 Causes de rejet d'un programme par le système *Ibis*

Les causes de rejet d'un programme par le système *Ibis* sont les mêmes que pour le système *Wob* à savoir une erreur de type dans MLGI ou bien un manque d'annotations. Comme nous l'avons fait remarquer, il n'est pas essentiel de rejeter un programme dès qu'une erreur d'unification est detectée : si le programme est réellement mal typé dans MLGI, l'inférence de MLGX saura le détecter aussi. Cette liberté permet d'ignorer une erreur en renvoyant le terme inchangé. Dans le cas où la condition de connaissance totale du système d'équations n'est pas vérifiée par manque d'annotations, on peut espérer qu'à l'itération $n+1$, grâce à l'élaboration effectuée lors de l'itération n, la forme de la valeur analysée du *pattern matching* ait été raffinée et qu'on obtienne ainsi une connaissance totale des équations.

8.3 Comparaison entre les deux algorithmes

L'algorithme d'inférence locale du système *Ibis* accepte strictement plus de programmes que l'algorithme d'inférence locale du système *Wob*. En effet, nous avons vu dans la section 8.2.1 que la fonction *double* est acceptée par *Ibis* et non par *Wob*. Le théorème suivant montre en outre que tout programme accepté par *Wob* l'est aussi par *Ibis* et que les formes qu'infère *Ibis* sont plus précises que celles de *Wob*.

Théorème 8.3.1 (*Ibis* subsume *Wob*)

- *Si* $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$ *est valide dans* Wob*, alors il existe une forme* s' *et un terme* t'' *tels que* $E, \Gamma \vdash t \Downarrow \bot \Uparrow s' \rightsquigarrow t''$ *est valide dans* Ibis *et* $s \preceq s'$.
- *Si* $E, \Gamma \vdash t \Downarrow s \rightsquigarrow t'$ *est valide dans* Wob*, alors il existe une forme* s' *et un terme* t'' *tels que* $E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t''$ *est valide dans* Ibis. ◇

Preuve. La preuve se fait par induction sur les dérivations du système *Ibis*. Elle ne présente pas de difficultés puisque les règles du système *Ibis* sont une superposition des règles du mode inférence et du mode vérification du système *Wob*. □

La comparaison avec l'algorithme d'inférence bidirectionnelle développée dans GHC par Simon Peyton Jones (Peyton Jones *et al*, 2004) est plus complexe car il est mêlé à l'inférence habituelle de ML.

Ainsi, d'un certain côté, l'algorithme de GHC est capable d'inférer un schéma de type pour un identifiant et de l'utiliser dans la suite pour propager les annotations de types alors que dans sa version stricte, notre algorithme d'inférence locale agit en une passe sur tout le programme et ne propage que les annotations de l'utilisateur. Dans l'exemple de la fonction *double*, on suppose connaître le schéma de type de la fonction *map* ce qui aide grandement à la propagation des annotations. Si cette fonction *map* a été définie dans la même unité de compilation, l'inférence locale ne pourra pas connaître son type. Il est donc nécessaire en pratique d'appliquer l'approche stratifiée à une granularité plus fine (par exemple sur toutes les déclarations *toplevel*) pour que l'inférence locale tire parti des schémas de type inférés par l'inférence de ML.

D'un autre côté, la fonction *double* n'est pas acceptée par GHC 6.7 à cause du caractère exclusif des deux modes d'inférence et de vérification comme nous l'avons montré dans la section 8.2.1.

CHAPITRE NEUF

Application : Analyseur LR bien typé

Ce chapitre présente en détail une application des types algébriques généralisés visant à encoder les invariants de bonne formation des analyseurs syntaxiques LR (Knuth, 1965). Comme on va se contenter de générer du code dont on connaît le type, ce chapitre n'a pas de liaison directe avec les chapitres précédents car la synthèse des types n'est pas utilisée à proprement parler.

9.1 Introduction

La transformation automatique de certaines classes de grammaires hors-contextes en analyseurs syntaxiques efficaces et exécutables est maintenant bien comprise (Aho *et al*, 1986). Par exemple, on sait que toute grammaire LALR(1) peut être transformée en un automate à pile déterministe qui reconnaît le langage dénoté par cette grammaire. Cette transformation a été implémentée dans de nombreux outils accessibles au grand public. Le programme POSIX `yacc` (Johnson, 1975), par exemple, change une grammaire LALR(1), décorée par des morceaux de code C appelés actions sémantiques, en un programme C exécutable. Dans le monde de la programmation fonctionnelle, Standard ML (Milner *et al*, 1997), Objective Caml (Leroy *et al*, 2004), et Haskell (Peyton Jones, 2003) possèdent tous un clone de `yacc`. Ces outils sont respectivement connus sous les noms de `ML-Yacc` (Tarditi & Appel, 2000), `ocamlyacc` (Leroy *et al*, 2004), et `happy` (Marlow & Gill, 2004).

D'un côté, les analyseurs syntaxiques générés par `yacc` sont rapides mais écrits en C, un langage de programmation non sûr. De même, les automates produits par `ocamlyacc` sont encodés sous forme de tables (pour les états et les transitions de l'automate) et de fonctions Objective Caml (pour les actions sémantiques), interprétées par un programme C. En examinant ces programmes C, il n'est pas évident de déterminer pourquoi la pile de l'automate contient toujours assez d'éléments ou pourquoi les changements forcés de type qu'on utilise pour stocker et récupérer les valeurs sémantiques sur cette pile sont sûrs. Ainsi, un bug dans `yacc` ou `ocamlyacc` pourrait, en principe, engendrer des analyseurs syntaxiques dont l'évaluation peut échouer. Du point de vue du mainteneur de `yacc` ou d'`ocamlyacc`, obtenir une garantie sur la bonne formation des analyseurs syntaxiques engendrés serait d'une grande utilité.

`ML-Yacc`, d'un autre côté, génère des programmes ML bien typés, ce qui garantit que les accès à la pile de l'automate ne peuvent pas être incohérents. Est-ce plus satisfaisant ? En fait, pas vraiment car bien qu'un programme ML ne puisse pas utiliser ses données de manière incorrecte, il peut échouer lors de son exécution soit à cause d'une exception non rattrappée, soit à cause d'une analyse par cas non exhaustive. Même si ces erreurs sont plus simples à corriger que celles mettant en jeu des incohérences de la mémoire, elles stoppent brutalement le programme ce qui est un problème majeur. En conséquence, les mainteneurs de `ML-Yacc` pourraient vouloir, en plus de la garantie de sûreté, une

garantie de non échec pour augmenter la robustesse des analyseurs syntaxiques générés. Si on rentre un peu plus dans les détails, on s'aperçoit que pour s'assurer de la validité des accès à la pile de l'automate, `ML-Yacc` attache des étiquettes à chaque cellule de la pile, à chaque valeur sémantique et à chaque état de l'automate. Pour s'assurer les bonnes grâces du typeur de ML, l'analyseur syntaxique généré passe beaucoup de temps à analyser ces étiquettes. Or, dans le cas où l'analyseur est correct, ces tests dynamiques ne sont pas nécessaires. On perd donc en performance pour un gain de sûreté très faible (le programme n'exécute pas d'instruction incorrecte pas mais est stoppé).

`happy` laisse ses utiliseurs face à ce choix cornélien en proposant ces deux modes : il peut produire un programme Haskell mal typé (donc sujet à d'éventuels incohérences) mais efficace ou bien un programme Haskell bien typé mais qui peut échouer et qui est inefficace.

Pour remédier à cette situation, nous suggérons d'écrire les analyseurs syntaxiques dans le langage de programmation MLGI décrit dans les chapitres précédents. L'idée est d'encoder des invariants de l'automate à pile dans les types de manière à ce que le typeur puisse prouver automatiquement que l'analyseur syntaxique s'évalue sans erreur et ne peut pas échouer même si nous n'attachons plus d'étiquette aux cellules de la pile et aux valeurs sémantiques. En deux mots, nous obtenons une implémentation efficace, bien typée et sûre des analyseurs syntaxiques.

Si on prend la casquette du prouveur de programme quelques instants, ce codage fournit pour chaque analyseur syntaxique généré une preuve de sa sûreté et non une preuve générale que tout analyseur généré est sûr. Si on reprend la distinction faite par Xavier Leroy dans son article sur la preuve de compilateur (Leroy, 2006), on peut dire que nous avons ici l'ébauche d'un générateur certifiant plutôt que certifié.

Ce résultat est intéressant de plusieurs points de vue. D'abord, il est original et peut être utilisé pour modifier `ML-Yacc`, `ocamlyacc` ou `happy` de manière à ce qu'ils produisent des analyseurs syntaxiques bien typés sans qu'il soit nécessaire d'utiliser des changements forcés de type ou un interprète écrit en C. Cela permet de transporter le générateur d'analyseur syntaxique en dehors de la base de confiance, c'est-à-dire de l'ensemble des outils auxquels on doit faire confiance pour croire en la sûreté de son programme. Bien sûr, le compilateur qui est utilisé pour compiler l'analyseur syntaxique généré en code machine doit encore faire partie de la base de confiance jusqu'à ce qu'on arrive à certifier le compilateur lui-même (Leroy, 2006).

Ensuite, nous pensons que cette application est représentative du pouvoir d'expression des types algébriques généralisés. Implémenter un automate à pile à l'aide de type algébrique ordinaire nécessite la présence physique des étiquettes à l'intérieur de la pile, ce qui est redondant puisque cette information est en fait déjà présente dans l'état de l'automate. Les types algébriques généralisés permettent au contraire de faire accepter au typeur que les étiquettes qui décrivent la forme de la pile sont physiquement à l'extérieur de celle-ci. En d'autres termes, notre travail exploite, et met en lumière, le fait que les types algébriques généralisés permettent de *coordonner des structures de données*, c'est-à-dire d'exprimer à l'aide du typage l'existence de corrélations entre des structures de données physiquement disjointes. Ce fait a déja été noté par Michael Ringenburg et Dan Grossman (Ringenburg & Grossman, 2005) mais ils le considèrent apparemment comme accidentel. Au contraire, nous pensons que ceci constitue l'essence de ce que fournissent les types algébriques généralisés.

9.2 Un exemple de grammaire

À travers ce chapitre, nous nous intéressons à une grammaire particulière pour les expressions arithmétiques (Aho *et al*, 1986) dont la définition apparaît dans la figure 9.1. Montrer comment construire un analyseur syntaxique pour cette spécification de grammaire particulière plutôt que de détailler entièrement l'implémentation du générateur d'analyseur syntaxique suffira pour décrire notre solution.

Les symboles terminaux de la grammaire sont **int**, `+`, `*`, `(`, et `)`. Nous supposons que l'analyseur

$$
\begin{array}{lrcl}
(1) & E\{x\} \texttt{ + } T\{y\} & \rightarrow & E\{x+y\} \\
(2) & T\{x\} & \rightarrow & E\{x\} \\
(3) & T\{x\} \texttt{ * } F\{y\} & \rightarrow & T\{x \times y\} \\
(4) & F\{x\} & \rightarrow & T\{x\} \\
(5) & \texttt{(} E\{x\} \texttt{)} & \rightarrow & F\{x\} \\
(6) & \textbf{int}\{x\} & \rightarrow & F\{x\}
\end{array}
$$

FIG. 9.1: Une grammaire décorée par des actions sémantiques.

lexical associe une valeur sémantique de type *int* au token **int** et une valeur sémantique de type *unit* aux autres terminaux. Les symboles non terminaux de la grammaire sont E (les expressions), T (les termes) et F (les facteurs). Le symbole initial de la grammaire est E.

Il y a six productions, numérotées de (1) à (6). Chaque production est de la forme $S_1 \ldots S_n \rightarrow S$, où $S_1, \ldots, S_n$ sont des symboles terminaux ou non terminaux et S est un symbole non terminal. Cependant, en plus de savoir si une entrée est effectivement une expression arithmétique syntaxiquement bien formée, nous sommes intéressés par le calcul de la valeur de cette expression (la valeur sémantique). Pour ce faire, chaque production est décorée par une action sémantique, c'est-à-dire une expression du langage de programmation cible, qui spécifie comment calculer une valeur sémantique à partir des valeurs sémantiques associées aux producteurs de la règle. Plus précisément, chaque S_i doit être suivi d'une variable x_i, alors que S doit être suivie d'une expression ML e. Les variables x_i et l'expression e sont entourées d'accolades dans la spécification. On interprète un S_i seul comme $S_i\{x_i\}$, où x_i n'apparaît nulle part ailleurs dans la production. Les variables $x_1, \ldots, x_n$ sont liées dans l'expression e.

Souvent, les valeurs sémantiques sont des arbres de syntaxe abstraite. Ici, par souci de simplicité, on préfère associer des valeurs sémantiques de type *int* aux symboles E, T et F. Au final, la grammaire de la figure 9.1 spécifie un évaluateur pour les expressions arithmétiques. Par exemple, sa première production exprime le fait qu'une expression E qui s'évalue en x, suivie du terminal `+`, suivi d'un terme T qui s'évalue en y, forment ensemble une expression arithmétique E qui s'évalue en $x + y$.

9.3 Un analyseur syntaxique LR pour cette grammaire

Nous décrivons maintenant un analyseur syntaxique pour notre exemple de grammaire. Cet analyseur syntaxique, extrait du livre de Aho *et al.* (1986), est présenté sous la forme d'un automate à pile. Cet automate consomme un flux d'entrée formé des symboles terminaux **int**, `+`, `*`, `(` et `)` et du symbole spécial `$` qui signale la fin de l'entrée. L'évaluation de l'automate nécessite le maintien d'un état courant. Les états sont modélisés par des entiers de l'ensemble $\{0, \ldots, 11\}$. On doit aussi maintenir une pile courante. Les piles sont de la forme : $\sigma ::= \epsilon \mid \sigma s v$, où s est la méta-variable dénotant les états et v dénote une valeur sémantique. ϵ représente la pile vide et $\sigma s v$ dénote une pile obtenue en empilant s et v sur la pile σ.

Initialement, le flux d'entrée consiste en la liste des symboles terminaux à analyser suivis par le symbole `$` ; l'état courant est 0 et la pile est vide.

Les transitions de l'automate sont définies par deux tables, *action* et *goto* qu'on peut trouver dans la figure 9.2. À chaque étape, l'automate consulte l'état courant ainsi que le symbole terminal en cours d'analyse. Ces deux informations déterminent une case dans le tableau à double entrée *action*. Le code inscrit dans cette case s'interprète ainsi :

1. Si le code est « shift s » (écrit « ss » dans la figure 9.2) où s est un état, alors l'état courant et le symbole terminal analysé sont poussés sur la pile, on passe au symbole terminal suivant et s

State	*action*						*goto*		
	int	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 9.2: Table issue de l'analyse de la grammaire.

devient l'état courant de l'automate.

2. Si le code est « reduce k » (écrit « rk » dans la figure 9.2), où la production numéro k de la grammaire est $S_1\{x_1\}\dots S_n\{x_n\} \to S\{e\}$, alors la pile courante *doit être de la forme* $\sigma s_1 v_1 \dots s_n v_n$. L'expression ML $e[v_1/x_1, \dots, v_n/x_n]$ est évaluée en une nouvelle valeur sémantique v. L'état s_1 et le non terminal S déterminent alors une case dans *goto* qui *doit être un état* s. Alors, $\sigma s_1 v$ devient la pile courante de l'automate et s son état courant. Dans cette situation, on ne passe pas au symbole terminal suivant.
3. Si le code est « accept » (écrit « **acc** » dans la figure 9.2) alors la pile courante *doit être de la forme* $\sigma s v$. L'évaluation de l'automate est alors terminée et on retourne v.
4. Si le code est indéfini alors cela signifie que la reconnaissance de l'entrée a échoué.

Le processus qui permet la construction de *action* et *goto* n'est pas notre propos ici (on consultera (Aho *et al*, 1986) pour une description).

Le point essentiel, c'est que ces tables sont construites d'une telle façon que les conditions où notre description informelle précédente utilise de l'expression « doit être » sont effectivement vérifiées. Cela signifie que tester ces conditions est superflu.

9.4 Une implémentation en ML

On décrit maintenant une implémentation en ML sans GADT de l'automate. Cette implémentation est bien typée mais elle effectue les tests superflus mentionnés dans la section précédente.

Pour commencer, on doit spécifier les moyens d'interaction entre l'analyse lexicale et notre analyseur syntaxique. On déclare le type des symboles terminaux :

```
type token =
  KPlus | KStar | KLeft | KRight | KEnd | KInt of int
```

On suppose que l'analyseur lexicale fournit deux fonctions, l'une permettant de lire le symbole en cours d'analyse et l'autre de passer au terminal suivant :

```
val peek : unit → token
val discard : unit → unit
```

On se plonge maintenant dans la conception de l'analyseur syntaxique. L'ensemble des états est décrit par une énumération, c'est à dire un type algébrique ordinaire dont tous les constructeurs ont une arité nulle :

type *state* = *S0* | *S1* | ... | *S11*

Ensuite, on déclare un autre type algébrique ordinaire pour implémenter la pile de l'automate. L'approche canonique consiste à mimer la définition des piles ($\sigma ::= \epsilon \mid \sigma sv$, voir la section 9.3) :

```
(* Définition temporaire. *)
type stack =
  SEmpty | SCons of stack × state × semantic_value
and semantic_value = ...
```

Cette déclaration affirme qu'une pile est une liste de paires formées d'un état et d'une valeur sémantique.

Il ne nous reste plus qu'à définir le type des valeurs sémantiques. Ici, les symboles `+`, `*`, `(`, `)` et `$` ont des valeurs sémantiques valant de type *unit* alors que **int**, E, T et F ont des valeurs sémantiques de type *int*. Il est clair que cette hétérogénéité des types des valeurs sémantiques force la définition d'un type somme, c'est-à-dire une nouvelle définition de type algébrique ordinaire.

Bien que naturelle et simple, cette solution introduit une certaine redondance puisqu'à la fois les cellules et les valeurs sémantiques contiennent des étiquettes. Pour éviter cette redondance, on utilise une méthode un peu moins directe qui fusionne les étiquettes des valeurs sémantiques et des cellules en intégrant les différents types de valeurs sémantiques directement à l'intérieur de la définition des piles :

```
type stack =
  | SEmpty
  | SP of stack × state
  | SS of stack × state
  | SL of stack × state
  | SR of stack × state
  | SI of stack × state × int
  | SE of stack × state × int
  | ST of stack × state × int
  | SF of stack × state × int
```

(Dans les noms choisis pour les constructeurs de données, P, S, L, R, et I sont des raccourcis pour *Plus*, *Star*, *Left*, *Right*, et *Int*.)

En examinant l'étiquette attachée à une valeur de type *stack*, on peut répondre en même temps à deux questions : est-ce que la pile est vide ? si elle n'est pas vide, quel type de valeurs sémantiques contient-elle à son sommet ?

Par ailleurs, à des fins d'optimisation, on choisit de ne pas représenter à l'exécution les valeurs de type *unit*. Par conséquent, les cellules associées aux symboles `+`, `*`, `(` et `)` contiennent seulement un état et non une paire contenant un état et la valeur unit.

Enfin, aucune cellule n'est construite avec le symbole `$` parce que, par construction, l'automate n'effectue jamais une action « shift » lorsqu'il rencontre ce symbole.

Pour résumer, chaque valeur du type *stack* contient une étiquette qui doit être examinée pour que le contenu réel soit accessible. Si, grâce à un raisonnement externe, l'étiquette est connue par avance alors ce test dynamique est redondant.

C'est cette approche où les valeurs sémantiques et/ou les cellules de la pile sont étiquetés qui est adoptée par `ML-Yacc` et `happy`.

La fonction centrale de l'analyseur syntaxique, *run* implémente l'évaluation de l'automate à pile. Elle est paramétré par l'état courant et la pile courante. Il peut s'arrêter soit en lançant l'exception

```
1  exception SyntaxError
2
3  let rec run : state → stack → int =
4    fun s stack →
5      match s, peek() with
6      | ...
7      | S9, KStar →
8          (* Décale vers l'état 7. *)
9          discard ();
10         run S7 (SS (stack, S9))
11     | S9, KPlus →
12         (* Réduit E{x} + T{y} → E{x + y}. *)
13         (* Dépile trois cellules . *)
14         let ST (SP (SE (stack, s, x), _), _, y) =
15           stack in
16         (* Empile l'état courant et la nouvelle valeur sémantique sur la pile . *)
17         let stack = SE (stack, s, x + y) in
18         (* Choisit un successeur sur la pile en utilisant la colonne E de la table goto. *)
19         gotoE s stack
20     | ...
21     | _, _ →
22         raise SyntaxError
23
24 and gotoE : state → stack → int =
25   fun s →
26     match s with
27     | S0 →
28         run S1
29     | S4 →
30         run S8
```

FIG. 9.3: Une implémentation bien typée en ML.

SyntaxError ce qui signifie que l'entrée n'est pas conforme à la grammaire, ou bien en retournant une valeur sémantique entière correspondant à la valeur de l'expression arithmétique E qui a été analysée. À l'exception des fonctions *peek* et *discard* qui font des effets de bord pour manipuler le flux d'entrée, la totalité de ce programme est purement fonctionnel.

La définition de *run* apparaît dans la figure 9.3. La fonction examine l'état courant s ainsi que l'entrée courante obtenue par la fonction *peek* et détermine quelle action effectuer. Il y a plusieurs cas, on en détaille deux.

Quand l'état courant vaut 9 et le symbole à analyser est ∗ (ligne 7), la table *action* de la figure 9.2 indique que l'automate doit effectuer une action « shift 7 ». On passe donc au symbole suivant, l'état courant 9 est placé au sommet de la pile et l'état courant est maintenant 7 (ligne 10). Dans ce cas particulier, aucune valeur sémantique n'est poussée sur la pile parce qu'aucune valeur sémantique n'est associée au symbole ∗. De même, le constructeur de données n'attend pas un troisième argument. L'état courant et la pile sont changés en suivant l'idiome de programmation fonctionnelle pure qui consiste à effectuer un appel récursif à *run* avec ces nouveaux paramètres.

Quand l'état courant est 9 et le symbole à analyser est + (ligne 11), la table *action* indique que l'automate doit réduire la production 1, c'est-à-dire $E\{x\} + T\{y\} \rightarrow E\{x+y\}$. Comme on l'a expliqué dans la section précédente, la structure des trois éléments placés au sommet de la pile est connue en

ce point. La première cellule de la pile contient l'étiquette *ST* et la valeur sémantique y associée au terme T. La seconde cellule doit être celle associée au symbole +, elle doit donc contenir l'étiquette *SP*. L'étiquette de la troisième cellule est nécessairement *SE* et est accompagnée par la valeur sémantique x associée à l'expression E. Grâce au *pattern matching*, x et y sont extraites de la pile (lignes 14–15). Simultanément, les variables *stack* et *s* sont liées à de nouvelles valeurs (masquant ainsi leurs anciennes valeurs). Ceci revient à dépiler trois cellules de la pile. L'action sémantique $x + y$ est ensuite évaluée et une nouvelle valeur sémantique est construite (ligne 17). Pour finir, la fonction auxiliaire *gotoE* est appelée (ligne 19) (il y a en fait une fonction *goto* par non terminale mais *gotoT* et *gotoF* ne sont pas montrées ici).

Le rôle de *gotoE* (ligne 24) est de déterminer le nouvel état de l'automate lorsque l'on vient de reconnaître un E. Pour cela, on regarde à l'intérieur de la colonne E de la table *goto*. La ligne de cette table correspond à l'état *s* qui a initié l'analyse du non terminal E. Ici, cette colonne n'a que deux entrées donc *s* est nécessairement *S0* (et l'état de l'automate doit alors devenir *S1* (ligne 28)) ou *S4* (et l'état de l'automate doit alors devenir *S8* (ligne 30)).

Notons que *gotoE* ne consulte et ne modifie pas la pile. En réalité, cette fonction n'a pas accès à la pile. On pourrait donc lui donner le type $state \to state$ et remplacer les appels à *gotoE* à l'aide de *run (gotoE s) stack*. Cependant, ceci briserait une propriété utile par la suite (voir la section 9.9) : dans la version que nous avons choisie, on appelle toujours *run* avec un état constant.

Comme annoncé, cette implémentation effectue des tests superflus à cause des *pattern matchings* non exhaustifs comme "***let*** *ST (...) = stack*" (lignes 14–15). On vérifie dynamiquement que la pile contient au moins trois cellules et que ces cellules sont associées aux symboles T, + et E.

De plus, dans la fonction *gotoE*, "***match*** *s* ***with***" (ligne 26) vérifie si *s* est effectivement *S0* ou *S4*.

Ces deux *pattern matching* ne sont pas exhaustifs donc le compilateur doit insérer du code qui lance une exception si jamais ils échouent. Ceci est à la fois une perte de temps et de taille de code : si l'analyseur syntaxique est correct alors, par construction de l'automate, ces tests ne peuvent pas échouer.

Avant de continuer, terminons cette section par deux remarques.

D'abord, dans cette implémentation, les tables *action* et *goto* sont compilées sous la forme d'un programme et non sous la forme de données (comme par exemple des tableaux d'entiers). Cette approche engendre des analyseurs syntaxiques plus gourmands en espace mais elle permet leur analyse par un système de type général. Un autre avantage, c'est de pouvoir spécialiser le code généré et donc d'obtenir éventuellement des gains en performance par rapport à un interprète de tables (voir la section 9.9).

Ensuite, nous représentons une pile comme une structure de données purement fonctionnelle c'est-à-dire une liste chaînée de cellules allouées dans le tas et non modifiables. Une paire de tableaux modifiables extensibles, l'un pour les états et l'autre pour les valeurs sémantiques serait peut-être plus efficace. Cependant, ce dernier tableau nécessiterait un type très fin témoignant de la variabilité des types des cases du tableau au cours de l'exécution car il s'agit d'une structure de données modifiable et hétérogène. Le système de type de ML ne supporte pas de genre de structures de données. Au minimum, une notion de linéarité doit être introduite pour garantir qu'aucun pointeur vers des données désallouées ne peut intervenir. Le choix d'une structure de données purement fonctionnelle est donc imposé par la discipline de type naïve de ML. Il existe quelques systèmes de type permettant d'exprimer les piles modifiables comme par exemple (Jia *et al*, 2005).

9.5 Comprendre l'invariant de l'automate

Nous avons affirmé plus haut que, par construction des tables *action* et *goto*, quand une action « reduce » est effectuée, alors le contenu du sommet de la pile est connu et peut être accédé sans

Forme de la pile							État
ϵ							0
ϵ	$\{0\}$	E					1
σ	$\{0,4\}$	T					2
σ	$\{0,4,6\}$	F					3
σ	$\{0,4,6,7\}$	(					4
σ	$\{0,4,6,7\}$	**int**					5
σ	$\{0,4\}$	E	$\{1,8\}$	+			6
σ	$\{0,4,6\}$	T	$\{2,9\}$	*			7
σ	$\{0,4,6,7\}$	(	$\{4\}$	E			8
σ	$\{0,4\}$	E	$\{1,8\}$	+	$\{6\}$	T	9
σ	$\{0,4,6\}$	T	$\{2,9\}$	*	$\{7\}$	F	10
σ	$\{0,4,6,7\}$	(	$\{4\}$	E	$\{8\}$	)	11

FIG. 9.4: L'invariant de l'automate.

vérification. Avant de modifier le programme pour tirer parti de cela, nous devons comprendre pourquoi il en est ainsi.

La raison est simple. Bien que les piles soient définies par des séquences arbitraires de paires formées d'un état et d'une valeur sémantique, les piles qui apparaissent effectivement durant l'exécution ont une forme plus déterminée. L'ensemble des formes possibles pour les piles est énuméré par la figure 9.4. La colonne de gauche décrit la forme de la pile σ alors que la colonne de droite indique dans quel état de l'automate on trouve cette forme de pile. Dans la colonne de gauche, ϵ représente la pile vide alors que σ représente une pile inconnue. Les entiers dénotent des états donc les ensembles d'entiers dénotent des ensembles d'états potentiellement présents dans la pile. Un terminal ou un non terminal dénote une valeur sémantique accompagnant ce symbole. Par exemple, la sixième ligne de la table affirme que quand l'automate est dans l'état 5 alors la pile est non vide et possède à son sommet un état appartenant à l'ensemble $\{0,4,6,7\}$ ainsi qu'une valeur sémantique pour le symbole **int**. Ceci implique en particulier que le sommet de la pile contient l'étiquette *SI* donc l'information contenue dans l'étiquette est bien redondante.

Définition 9.5.1
Une pile σ et un état s sont cohérents si et seulement si

(i) σ et s sont d'une des formes décrites par la figure 9.4 ;

(ii) si σ est de la forme $\sigma' s' v'$ alors σ' et s' sont consistants. ◇

Alors, on a l'invariant suivant :

Invariant 9.6 *À tout instant, la pile courante de l'automate et son état courant sont consistants.*

On peut faire cette preuve par une induction sur les exécutions possibles de l'automate défini dans la section 9.3. La pile initiale est ϵ et l'état initial est 0, ils sont consistants puisqu'ils apparaissent dans la première ligne de la figure 9.4. Il reste à prouver que toute transition possible préserve cet invariant ce qui est une exploration fastidieuse mais simple.

En fait, une fois que nous aurons introduit l'invariant dans le type de la fonction *run*, cette preuve sera faite automatiquement par le typeur.

On pourrait avoir l'impression que nous avons découvert l'invariant après avoir construit l'automate par le biais d'une analyse attentive (et peut-être délicate) de ses transitions. Il n'en est rien : déterminer l'invariant d'un automate se fait par une simple observation de ses *items*(Knuth, 1965, Aho *et al*, 1986). Aucune inférence complexe d'invariant n'est donc nécessaire.

9.7 Garder trace de la structure de la pile

Grâce à l'invariant de l'automate, examiner l'état courant de l'automate est suffisant pour déterminer sans ambiguïté la forme de la pile. Par exemple, si l'état courant est 9 alors la pile contient à son sommet trois cellules étiquetées par *SE*, *SP*, et *ST*. C'est exactement l'information nécessaire pour prouver que la construction "***let*** *ST (SP (SE (...))) = stack*" (figure 9.3) ne peut échouer.

Mais comment persuader le compilateur d'un tel fait ? Nous devons lui faire prendre conscience de la corrélation entre les états et la forme des piles.

Pour ce faire, nous paramétrons le type des états par une variable de type α. L'idée, c'est que si l'état courant de l'automate est de type α *state* alors la pile courante est de type α. La représentation des états ne change pas car le type α ne décrit pas la structure des états. Ce paramètre est donc un paramètre fantôme (Hinze, 2003).

Dans la suite, nous supposons que le type *state* a une définition qui satisfait cette intuition et nous expliquons comment cela permet de changer la manière d'implémenter les piles (section 9.7.1). Ensuite, on définit concrètement le type *state* comme un type algébrique généralisé (section 9.7.2). Enfin, nous discutons comment ces changements dans nos déclarations de type modifie le code qui implémente l'évaluation de l'automate (section 9.7.3).

9.7.1 Types des cellules de la pile

Pour convaincre le typeur que l'accès à la pile ne peut pas échouer, nous n'avons plus à définir le type *stack* à l'aide de données étiquetées. À la place, notre vocabulaire sera assez précis pour exprimer, disons, « le type de toutes les piles qui ont trois cellules associées aux symboles E,+ et T à leur sommet ». Pour cette raison, nous ne définissons plus un unique type *stack* mais une famille de types (figure 9.5, lignes 2–10).

Le type *empty* (figure 9.5, ligne 2) est le type de la pile vide. L'unique valeur qui l'habite est *SEmpty*. Ce type est isomorphe à *unit*.

Le type α *cP* (figure 9.5, ligne 3) est le type des piles non vides dont la cellule au somment est associée au symbole + et dont le reste des cellules est une pile de type α. Un point clé est que le type α apparaît deux fois dans cette définition, une fois en tant que type du reste de la pile et une seconde fois en tant que paramètre de *state*. Ceci encode le point (ii) dans la définition de la cohérence (section 9.5) et indique au système de type que dans chaque cellule, il y a une relation entre l'état qui est dans la pile et la structure de la pile qui se trouve en dessous. Les définitions sur les lignes 4–10 sont analogues.

Ainsi, une pile qui consiste en une cellule unique associée au symbole E a le type *empty cE*. Une pile formée de deux cellules, respectivement associées aux symboles E et + a le type *empty cE cP* et ainsi de suite. Il peut sembler désormais nécessaire d'utiliser des types d'une taille arbitraire de manière à dénoter l'ensemble de toutes les piles possibles. Par chance, pour parvenir à exprimer l'invariant de manière correcte, une information incomplète sur leur forme est suffisante. Par exemple, toute pile non vide dont la cellule au sommet est associée au symbole E doit avoir un type de la forme τ *cE* pour un certain type τ. En conséquence, une telle pile est un argument valide pour une fonction de type $\forall\alpha.\alpha\ cE \to \ldots$. Autrement dit, grâce aux variables de type et aux abstractions de type, un simple schéma de type suffit pour décrire toute une collection de piles.

Même si par souci de simplicité syntaxique, nous avons maintenu les étiquettes *SEmpty*, *SP*, etc., aucun de ces types définis dans lignes 2–10 de la figure 9.5 ne sont réellement des données étiquetées. En fait, ce sont des n-uplets de taille 0 (*empty*), deux (*cP*, etc.) ou 3 (*cI*, etc.). Le fait de n'avoir qu'un constructeur de données pour chacun de ces types supprime la nécessité d'avoir une étiquette pour discriminer parmi plusieurs choix. Une pile est encore une liste chaînée de cellules mais aucune cellule ne contient d'étiquette puisque c'est en observant l'état de l'automate qu'on détermine leur

```
1  (* Le type des cellules de la pile. *)
2  type empty = SEmpty
3  type α cP = SP of α × α state
4  type α cS = SS of α × α state
5  type α cL = SL of α × α state
6  type α cR = SR of α × α state
7  type α cI = SI of α × α state × int
8  type α cE = SE of α × α state × int
9  type α cT = ST of α × α state × int
10 type α cF = SF of α × α state × int
11
12 (* Le type des états. *)
13 type state : ⋆ → ⋆ =
14 | S0 : empty state
15 | S1 : empty cE state
16 | S2 : ∀α.α cT state
17 | S3 : ∀α.α cF state
18 | S4 : ∀α.α cL state
19 | S5 : ∀α.α cI state
20 | S6 : ∀α.α cE cP state
21 | S7 : ∀α.α cT cS state
22 | S8 : ∀α.α cL cE state
23 | S9 : ∀α.α cE cP cT state
24 | S10 : ∀α.α cT cS cF state
25 | S11 : ∀α.α cL cE cR state
```

FIG. 9.5: Codage d'une partie des invariants à l'intérieur des types.

forme. La pile de l'automate et son état sont donc des structures de données coordonnées dans le sens de Michael Ringenburg et Dan Grossman (2005).

9.7.2 Types des états

Nous revenons maintenant à la définition du type paramétré α *state* (figure 9.5, lignes 13–25). La ligne 13 affirme que le constructeur de type *state* a pour sorte $\star \to \star$ ce qui veut dire que c'est un constructeur de type unaire. Les lignes 14–25 spécifient ses constructeurs de données. Tous ces constructeurs n'attendent aucun argument. Il s'agit donc encore d'une énumération. La différence avec la version précédente, c'est la variation des instances choisies pour le paramètre. Cette variation est acceptée grâce à l'extension de ML avec des GADT.

Ces différentes instances reflètent la connaissance de la forme de la pile pour chacun des états. Considérons l'état 0. D'après la figure 9.4, quand l'automate est dans cet état, la pile est vide. Nous donnons donc à *S0* le type *empty state* (ligne 14). Ici, le paramètre est contraint à avoir le type *empty*. De même, quand l'automate est dans l'état 0, la pile est une unique cellule contenant la valeur sémantique du symbole E. Ainsi, *S1* a pour type *empty cE state* (ligne 15). Dans le cas de l'état 2, les choses sont légèrement plus complexes : la figure 9.4 nous apprend que lorsque l'automate est dans l'état 2 alors la pile a une cellule associée au symbole T à son sommet et le reste de la pile est inconnu. *S2* se voit donc associer le type $\forall\alpha.\alpha$ *cT state* (ligne 16). La variable de type α est quantifiée universellement de manière à dénoter le reste inconnu de la pile. C'est pourquoi n'importe qu'elle valeur pour α est compatible avec l'état *S2*. Savoir que l'automate est dans l'état *S2* permet donc de conclure que la pile a la forme τ *cT* pour un certain type τ. Les déclarations pour les états de *S3*

```
1  let rec run : ∀α.α state → α → int =
2    fun s stack →
3      match s, peek() with
4      | ...
5      | S9, KStar →
6          discard ();
7          run S7 (SS (stack, S9))
8      | S9, KPlus →
9          let ST (SP (SE (stack, s, x), _), _, y) =
10           stack in
11         let stack = SE (stack, s, x + y) in
12         gotoE s stack
13     | ...
14     | _, _ →
15         raise SyntaxError
16
17 and gotoE : ∀α.α state → α cE → int =
18   fun s →
19     match s with
20     | S0 →
21         run S1
22     | S4 →
23         run S8
```

FIG. 9.6: Une implémentation plus précise (en ne changeant que les types).

à *S11* (lignes 17–25) sont déduites par le même raisonnement. Cette déclaration de type algébrique généralisé encode donc le point (i) de la définition de la consistance (section 9.5).

Les définitions de *cP*, *cS*, etc. et de *state* sont mutuellement récursives.

Il nous reste à montrer comment ces types plus spécifiques enrichissent le typage de l'analyse par cas sur les états.

9.7.3 Implementation

Étudions la définition de la fonction *run* qui est fournie dans la figure 9.6. Il est remarquable que seules les annotations de type de *run* et de *gotoE* soient modifiées. Le programme en lui-même n'a pas changé. Ceci montre que l'apport en précision des déclarations de GADT est l'unique raison pour laquelle le programme devient plus sûr.

Le type de *run* qui était $state \to stack \to int$ devient $\forall\alpha.\alpha\ state \to \alpha \to int$ (ligne 1). En clair, la structure de la pile, représentée par le type α peut être arbitraire tant qu'elle reste consistance avec l'état courant. Partager la variable de type α entre le type de état et celui de la pile nous permet de refléter la corrélation (on pourrait aussi dire la dépendance) entre l'état courant et la pile courante.

Par hypothèse, s a le type $\alpha\ state$. Grâce à notre définition de *state* sous la forme d'un type algébrique généralisé, examiner s (ligne 3) raffine notre connaissance de α. Par exemple, si il s'avère que s coïncide avec *S9*. Par définition du type *state* (figure 9.5, ligne 23), *S9* a tous les types de la forme $\tau'\ cE\ cP\ cT\ state$ et seulement ceux-là. Ainsi, si s est égal à *S9*, alors le type dénoté par α doit être de la forme $\tau'\ cE\ cP\ cT$ pour un certain type τ'. Du point de vue du typage, il est donc parfaitement licite d'enrichir le système d'équations entre types par :

$$\alpha = \alpha'\ cE\ cP\ cT, \tag{1}$$

où α' est une variable de type choisie fraîche par rapport à l'environnement.

***Les actions* « *shift* k »** Expliquons maintenant pour quelles raisons une action « shift k » (lignes 6–7) est bien typée. Étant donné que *stack* et s ont les types α et α *state*, la cellule *SS (stack, S9)* en ligne 7 a le type α *cS*.

Posons $\tau \equiv \alpha'$ *cE cP* où α' est la variable de type abstraite introduite dans le paragraphe précédent. Alors, l'équation (1) peut être écrite $\alpha = \tau$ *cT* qui par congruence implique

$$\alpha\ cS = \tau\ cT\ cS.$$

Ceci permet au typeur de déduire que *SS (stack, S9)*, dont le type est α *cS*, a aussi le type τ *cT cS*.

Pour finir, en instanciant la variable de type α en le type τ dans la définition de *S7* (figure 9.5, ligne 21), on montre que τ *cT cS state* est un type valide pour *S7*. Il vient par l'instanciation de α en τ *cT cS* dans le type de *run* que *run S7 (SS (stack, S9))* (ligne 7) est bien typé.

En résumé, nous avons vérifié que les deux cellules au sommet de la pile existent et sont associées aux symboles T et $*$. C'est suffisant pour garantir que la nouvelle pile est cohérente avec l'état 7. Le fait qu'on est instancié α en τ dans la définition de *S7* où τ vaut α' *cE cP* signie que l'automate oublie l'existence de ces deux cellules quand il effectue une action « shift k ».

***Les actions* « *reduce* E »** Expliquons ici pourquoi les actions « reduce E » (lignes 9–12) sont bien typées. La variable *stack* a le type α donc par l'équation (1), elle a aussi le type α' *cE cP cT*. Il est donc licite de filtrer *stack* par le motif :

ST (SP (SE (stack, s, x), _), _, y), et cela a pour effet de lier *stack*, s, x et y à des valeurs de types α', α' *state*, *int* et *int* respectivement. Ce *pattern matching* ne peut de surcroît pas échouer car il s'agit seulement de la déconstruction d'un n-uplet dont on connaît parfaitement le nombre de composantes. Comme x et y ont tous les deux le type *int*, l'expression $x + y$ (ligne 11) est bien typée et la nouvelle pile *stack* (ligne 11) a le type α' *cE*. Donc, l'appel à *gotoE* (ligne 12) est valide.

***Les fonctions* « *Goto* »** La tâche de *gotoE* (lignes 17–23) est de recouvrer l'information qui a été perdue lors de l'action « shift k ». Quand *gotoE* est appelée, on sait que la cellule qui est au sommet de la pile est associée au symbole E. En effet, nous sommes alors en pleine réduction d'une production d'un symbole E donc nous venons de créer une cellule, dans le code *run*, avant d'invoquer *gotoE*. Cependant, rien de plus n'est connu sur la pile. Pour récupérer l'information nécessaire sur le reste de la pile, nous devons examiner l'état qui se trouve dans la cellule plaçée au sommet de la pile. Cet état est passé à la fonction *gotoE* sous le nom s (ligne 18). Il s'agit de l'état de l'automate dans lequel on a initié l'analyse de E.

Imaginons que s est *S4* (ligne 22). Conformément au type énoncé pour *gotoE* (ligne 17), s a le type α *state*. Si on confronte cette information avec la définition de *S4* (figure 9.5), on en conclut que l'équation

$$\alpha = \alpha'\ cL, \tag{2}$$

est licite dans le contexte de typage (en choisissant la variable de type α' suffisamment fraîche). Dès lors, nous retrouvons l'information concernant la cellule suivante dans la pile. On apprend que la pile n'est pas vide et que la cellule suivante est associée au symbole (. Ici, le succès d'un test dynamique, l'analyse par cas sur s, fournit une information statique sur la forme de la pile. Cette propriété est caractéristique des types algébriques généralisés.

Finalement, en instanciant α en α' dans les types de *run* et *S8*, on obtient que *run S8* a le type α' *cL cE* $\rightarrow$ *int* ce qui peut être écrit α *cE* $\rightarrow$ *int* à l'aide de l'équation (2). La valeur retournée satisfait alors le type énoncé pour *gotoE* (ligne 17).

Le cas où s est *S0* est similaire.

9.7.4 Résumé et remarques

Nous avons encodé une part de l'invariant de la figure 9.4 au sein des définitions de type de la figure 9.5. Cela a permis la suppression des étiquettes attachées aux cellules de la pile et par conséquent la suppression des tests redondants. Plus importante peut-être est la garantie obtenue : le code qui extrait les valeurs sémantiques de la pile lors des actions « reduce E » est maintenant sûr.

On pourrait s'interroger sur la réalité de la suppression du test dynamique car le programme en lui-même n'a pas changé. En fait, c'est la nouvelle information de typage qui permet de transformer une analyse par cas non exhaustive en une analyse exhaustive. Comme il n'y a ici qu'un seul cas, il n'est plus nécessaire pour le compilateur d'introduire une vérification.

La fonction *gotoE* effectue encore une analyse par cas non exhaustive (ligne 19) qui est donc traduite en un test dynamique parce que le typeur n'a aucune information lui indiquant que les seuls états ayant pu initier l'analyse d'un E sont *S0* et *S4*. En conséquence, le compilateur doit émettre un avertissement lors de la compilation et engendrer le code lançant une exception lors de l'exécution si jamais s n'est pas l'un de ces deux états. Il reste donc encore une faille permettant à notre analyseur syntaxique d'échouer si un bug est présent dans le générateur. Nous allons nous attaquer à ce problème dans la section 9.8.

Le code de la ligne 9 de la figure 9.6 doit être capable d'accéder à la troisième cellule qui contient *stack*, s et x sans examiner les deux cellules placées au sommet de la pile (en utilisant le motif _). Ce prérequis semble interdire une représentation des piles comme une type algébrique ordinaire où l'état contenu par chaque cellule sert d'étiquette qui doit être observée pour pouvoir accéder au reste de la pile.

Dans cette implémentation, la pile de l'automate est explicitement allouée. Il existe une implémentation alternative, qui rend la pile implicite à l'aide de la currification. Dans cette implémentation, dépiler une cellule s'implémente en retournant une fonction qui consomme deux arguments (une valeur sémantique et un état) alors qu'empiler une cellule sur la pile est effectué par un appel récursif à la fonction *run* avec deux arguments supplémentaires (encore une fois, une valeur sémantique et un état). Les types qui décrivent les piles sont définis ainsi :

type (α, β) *cell* $= \beta \rightarrow \alpha$ *state* $\rightarrow \alpha$

Ici, le type β et le type α *state* représentent respectivement la valeur sémantique et l'état qui sont trouvés en haut de la pile. Le α final encode la structure du reste de la pile. Le type de la fonction *run* devient $\forall\alpha.\alpha$ *state* $\rightarrow \alpha$. On remarque que la définition du type de *state* est inchangée. On peut vérifier que, avec ces définitions, l'application *run S0* a le type *empty*, c'est-à-dire, *svE*, comme désiré.

9.8 Garder trace des états présents dans la pile

De manière à éliminer le dernier test dynamique redondant de la fonction *gotoE*, il est nécessaire de prouver que le paramètre s doit valoir soit *S0*, soit *S4*. Comme s est originellement trouvé dans la pile durant une action « reduce E », nous devons garder une trace de l'identité des états présents dans la pile.

Pour cela, on rajoute un nouveau paramètre, ρ, au constructeur de type *state*. Informellement, on peut dire que ρ est à valeur dans l'ensemble des états et que, par conséquent, un état a le type (τ, ρ) *state* seulement si il est membre de l'ensemble ρ. Par exemple, si un état a le type $(\alpha, \{0, 4\})$ *state* alors il peut être ou bien *S0*, ou bien *S4* mais aucun autre.

Si on revient à un discours plus technique, on doit se rendre à l'évidence que les ensembles d'entiers ne sont pas des types (en ML) donc ρ ne peut strictement pas être un ensemble d'entiers. On doit donc passer par le chemin détourné d'un codage des ensembles d'entiers dans les types de ML. Nous allons discuter cet codage (section 9.8.1) et ensuite nous allons expliquer comment définir le nouveau

constructeur de type binaire *state* (section 9.8.2) pour finir sur l'étude des modifications apportées sur le code de l'évaluation de l'automate (section 9.8.3).

9.8.1 Encoder des ensembles d'entiers dans les types

Posons *pre* et *abs*, deux types abstraits qui vont représenter la notion de présence et d'absence respectivement (ils peuvent être définis par deux définitions de type algébrique sans constructeur). On peut alors encoder les ensembles d'états sous la forme de 12-uplets dont les composantes vaudraient *pre*, *abs* ou des variables de type.

Par exemple, l'ensemble constant $\{0, 4\}$ peut être encodé par le type $\mathit{pre} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{pre} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs}$. Pour gagner en lisibilité, on écrira $\{0, 4\}$ pour ce type. Plus généralement, si S est un ensemble arbitraire d'états, nous écrivons $\{S\}$ pour le type produit dont la composante numéro i est *pre* (respectivement *abs*) si et seulement si $i \in S$ (respectivement $i \notin S$).

Un sous-ensemble arbitraire de $\{0, 4\}$ peut être encodé par le type $\gamma_0 \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \gamma_4 \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs} \times \mathit{abs}$, où γ_0 et γ_4 sont des variables fraîches. Nous écrirons $\langle 0, 4\rangle$ pour un tel type. Plus généralement, si S est un ensemble arbitraire d'état, nous écrivons $\langle S\rangle$ pour le type produit dont le composante numéro i est une variable fraîche γ_i (respectivement *abs*) si et seulement si $i \in S$ (respectivement $i \notin S$). Cette notation est concise mais particulièrement informelle puisqu'elle ne spécifie pas clairement comment les γ_i sont choisies. Nous nous contentons de cette approximation pour simplifier la présentation. Pour être plus formel, il faudrait indiquer où ces variables sont liées et par rapport à quel contexte exactement elles doivent être choisies fraîches.

Soient S et S' deux ensembles d'états. Deux propriétés clés sont

(i) le type $\{S\}$ est une instance du type $\langle S'\rangle$ si et seulement si $S \subseteq S'$,

(ii) le type $\langle S\rangle$ est une instance du type $\langle S'\rangle$ si et seulement si $S \subseteq S'$.

Nous sommes donc capables d'encoder la relation « est un sous-ensemble de » entre ensembles d'états dans le système de type de ML bien qu'il ne s'appuie que sur l'unification et non sur une notion de sous-typage. Cette méthode est inspirée du travail de Didier Rémy concernant la simulation du sous-typage des enregistrements à l'aide d'unification (Rémy, 1994). L'idée au centre de cette méthode a été indépendamment étudiée par Matthew Fluet et Riccardo Pucella (Fluet & Pucella, 2002).

Ce codage fonctionne mais est extrêmement peu concis quand l'automate possède beaucoup d'états puisque la taille de la représentation d'un ensemble S est toujours proportionnelle au cardinal de l'ensemble des états de l'automate et non à celui de S. Si le système de type du langage hôte possède des rangées (Rémy, 1994), on peut utiliser un codage plus économe en espace.

L'idée est d'encoder les ensembles d'états à l'aide de rangées dont les étiquettes sont les états et dont les composantes sont *pre*, *abs* ou des variables de type. Nous écrivons maintenant $\{0, 4\}$ pour la rangée $(0 : \mathit{pre}\,; 4 : \mathit{pre}\,; \partial\mathit{abs})$ qui associe *pre* à 0 et 4 et *abs* à tous les autres états. Nous écrivons $\langle 0, 4\rangle$ pour la rangée $(0 : \gamma_0; 4 : \gamma_4; \partial\mathit{abs})$. L'égalité des rangées est définie *modulo* les règles suivantes de commutation et d'expansion :

$$\begin{aligned}(s_1 : \tau_1; s_2 : \tau_2; \tau) &= (s_2 : \tau_2; s_1 : \tau_1; \tau)\\ (s : \tau; \partial\tau) &= \partial\tau\end{aligned}$$

Cet codage vérifie les propriétés (i) et (ii) décrites plus haut. Bien sûr, ces règles doivent être intégrées dans le typeur lorsqu'il décide l'égalité des types et l'implication entre égalités de type (pour la partie du typeur liée aux types algébriques généralisés).

On peut reconnaître qu'aucun de ces deux codages n'est extrêmement naturel et qu'il s'agit d'un accident si le système de type ML le permet. Des objets plus riches ne sont d'ailleurs tout simplement pas encodables dans les types de ML. La seconde partie de cette thèse propose un langage de programmation permettant l'expression, sans codage, d'invariants et de structures mathématiques arbitrairement complexes.

```
1  type empty = SEmpty
2  type (α, ρ) cP = SP of α × (α, ρ) state
3  type (α, ρ) cS = SS of α × (α, ρ) state
4  type (α, ρ) cL = SL of α × (α, ρ) state
5  type (α, ρ) cR = SR of α × (α, ρ) state
6  type (α, ρ) cI = SI of α × (α, ρ) state × int
7  type (α, ρ) cE = SE of α × (α, ρ) state × int
8  type (α, ρ) cT = ST of α × (α, ρ) state × int
9  type (α, ρ) cF = SF of α × (α, ρ) state × int
10
11 type state : (⋆, row) → ⋆ where
12 | S0 : (empty, {0}) state
13 | S1 : ∀γ̄.((empty, ⟨0⟩) cE, {1}) state
14 | S2 : ∀αγ̄.((α, ⟨0, 4⟩) cT, {2}) state
15 | S3 : ∀αγ̄.((α, ⟨0, 4, 6⟩) cF, {3}) state
16 | S4 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cL, {4}) state
17 | S5 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cI, {5}) state
18 | S6 : ∀αγ̄.(((α, ⟨0, 4⟩) cE, ⟨1, 8⟩) cP, {6}) state
19 | S7 : ∀αγ̄.(((α, ⟨0, 4, 6⟩) cT, ⟨2, 9⟩) cS, {7}) state
20 | S8 : ∀αγ̄.(((α, ⟨0, 4, 6, 7⟩) cL, ⟨4⟩) cE, {8}) state
21 | S9 : ∀αγ̄.((((α, ⟨0, 4⟩) cE, ⟨1, 8⟩) cP, ⟨6⟩) cT, {9}) state
22 | S10 : ∀αγ̄.((((α, ⟨0, 4, 6⟩) cT, ⟨2, 9⟩) cS, ⟨7⟩) cF, {10}) state
23 | S11 : ∀αγ̄.((((α, ⟨0, 4, 6, 7⟩) cL, ⟨4⟩) cE, ⟨8⟩) cR, {11}) state
24
25 val run : ∀αρ.(α, ρ) state → α → int
26 val gotoE : ∀αγ̄.(α, ρ) state → (α, ρ) cE → int
27                                   where ρ = ⟨0, 4⟩
```

FIG. 9.7: Codage de la totalité de l'invariant dans les types.

9.8.2 Types des états

Une fois muni de la notation pour encoder les ensembles d'états à l'aide de types, nous pouvons donner une définition du type *state* (figure 9.7). Il y a de grands changements par rapport à la définition précédente (figure 9.5).

Le premier changement est présent dans chaque ligne : le second paramètre de *state* est contraint de manière à refléter l'identité de l'état de manière exacte. Par exemple, *S0* a un type de la forme $(\ldots, \{0\})$ *state* (ligne 12) ; *S1* a un type de la forme $(\ldots, \{1\})$ *state* (ligne 13) et ainsi de suite. Un type de la forme $(\tau, \{0\})$ *state* peut alors être habité seulement par *S0* ; un type de la forme $(\tau, \{1\})$ *state* est habité uniquement par *S1* et ainsi de suite. Cette technique qui consiste à créer un type pour chaque valeur s'appelle « type singleton ».

Le second changement qu'on retrouve aussi sur chaque ligne est la précision accrue de la description de la forme de la pile. Désormais, le premier paramètre est modifié pour garder la trace de l'identité des états contenus à l'intérieur de la pile. Pour chaque cellule, une borne supérieure de l'identité de l'état qu'elle contient est dénotée par un type de la forme $\langle S \rangle$. Par exemple, le type déclaré pour *S2* est

$$\forall \alpha \bar{\gamma}.((\alpha, \langle 0, 4 \rangle)\ cT, \{2\})\ state$$

(ligne 14), qui reflète le fait que, dès lors que l'état courant est 2, la cellule au sommet de la pile contient une valeur sémantique associée au symbole T et un état de l'ensemble $\{0, 4\}$. Par convention, sur chaque ligne, le vecteur $\bar{\gamma}$ représente l'ensemble des variables de type utilisées implicitement dans

la abréviation $\langle S \rangle$ lorsque l'on l'expanse en sa définition.

9.8.3 Implémentation

Le dernier changement est celui du type de *run* et de *gotoE* (lignes 25–27).

Le premier paramètre de *run* a maintenant le type (α, ρ) *state* à la place de α *state*. La variable ρ n'est pas contrainte puisque *run* accepte n'importe quel état.

Le premier paramètre de *gotoE* a maintenant le type (α, ρ) *state*, où ρ est une notation pour $\langle 0, 4 \rangle$, *i.e.* $(0 : \gamma_0; 4 : \gamma_4; \partial abs)$. Nous utilisons consciemment une notation plutôt que d'écrire simplement $\langle 0, 4 \rangle$ deux fois parce que cette dernière écriture introduirait quatre variables de type fraîches $\gamma_0, \gamma_4, \gamma'_0$ et γ'_4 ce qui n'est pas le but recherché (on décorrélerait les deux types). Les variables de type γ_0 et γ_4 sont universellement quantifiées : en effet, ici, $\bar{\gamma}$ signifie $\gamma_0\gamma_4$.

Étant donné que γ_0 et γ_4 sont universellement quantifiées, elles peuvent être instanciées à volonté avec *pre* et *abs*. L'application *gotoE S0* est donc bien typée : en effet, par la propriété (i), le type $\{0\}$ est une instance du type $\langle 0, 4 \rangle$. De même, *gotoE S4* est bien typé. Cependant, aucun autre état ne peut être passé à *gotoE*. Par exemple, l'application *gotoE S1* est mal typée parce que le type $\{1\}$ n'est pas une instance du type $\langle 0, 4 \rangle$.

Un typeur pour les types algébriques généralisés peut peut tirer profit de cette propriété pour déduire que l'analyse par cas à l'intérieur de *gotoE* est exhaustive et ne peut pas échouer. Considérons, par exemple, le cas de *S1* qui est un cas syntaxiquement manquant. Supposons que *S1* soit présent dans *gotoE*, alors l'équation suivante devrait être prise en compte pour vérifier le bon typage du corps de la branche :

$$\{1\} = \langle 0, 4 \rangle,$$

c'est-à-dire :

$$(1 : pre; \partial abs) = (0 : \gamma_0; 4 : \gamma_4; \partial abs),$$

où γ_0 et γ_4 sont fraîches. Par la propriété (i), cette équation est non satisfiable ce qui prouve qu'une telle branche serait morte.

Le même raisonnement peut être utilisé pour tout état différent de 0 et 4 ce qui permet de conclure que l'analyse par cas est réellement exhaustive même si elle ne met en jeu explicitement que deux cas. Aucun avertissement n'apparaît lors de la compilation. Cette propriété est l'élimination du code mort décrite par Hongwei Xi (Xi, 1999) et par François Pottier et Vincent Simonet (Simonet & Pottier, 2005).

Nous laissons le lecteur vérifier que le code des actions « shift k » et « reduce E » demeure bien typé après ces changements. La propriété (ii) est utilisée pour vérifier les actions « shift k ». Encore une fois, en passant de la section 9.7 à la section 9.8, nous n'avons eu à modifier qu'un ensemble restreint de déclarations de type et d'annotations mais le programme lui-même n'a pas été modifié. Le seul effet de ce supplément d'information de typage est de permettre au compilateur de gérer de manière plus satisfaisante l'analyse par cas à l'intérieur de *gotoE*. Aucun avertissement n'est produit ce qui signifie que le compilateur garantit désormais que le programme s'évalue sans erreur et ne peut pas échouer.

Toute l'information contenue dans le figure 9.4 est maintenant encodée dans les types *state*. Quand le typeur analyse le programme, il vérifie donc automatiquement que l'invariant de l'automate est maintenu par le programme.

9.9 Optimisations

Nous concluons avec une liste d'optimisations optionnelles. Elles sont immédiates mais le point important est la robustesse de nos types : ils sont toujours valables à travers ces transformations du programme.

```
1  type empty = SEmpty
2  type (α, ρ) cL = SL of α × (α, ρ) state
3  type (α, ρ) cE = SE of α × (α, ρ) state × int
4  type (α, ρ) cT = ST of α × (α, ρ) state × int
5
6  type state : (⋆, row) → ⋆ where
7  | S0 : (empty, {0}) state
8  | S4 : ∀αγ̄.((α, ⟨0, 4, 6, 7⟩) cL, {4}) state
9  | S6 : ∀αγ̄.((α, ⟨0, 4⟩) cE, {6}) state
10 | S7 : ∀αγ̄.((α, ⟨0, 4, 6⟩) cT, {7}) state
11
12 let rec gotoT : ∀αγ̄.(α, ρ) state → (α, ρ) cT → int
13 where ρ = ⟨0, 4, 6⟩ =
14   fun s →
15     match s with
16     | S0 → ...
17     | S4 → ...
18     | S6 →
19         (* Version inlinée de run9. *)
20         fun stack →
21           match peek() with
22           | KStar →
23               discard ();
24               run7 stack
25           | KPlus →
26               let ST (SE (stack, s, x), _, y) =
27                 stack in
28               let stack = SE (stack, s, x + y) in
29               gotoE s stack
30           | ...
31           | _ →
32               raise SyntaxError
33
34 and gotoE : ∀αγ̄.(α, ρ) state → (α, ρ) cE → int where ρ = ⟨0, 4⟩ =
35   fun s →
36     match s with
37     | S0 →
38         run1
39     | S4 →
40         (* Version inlinée de run8. *)
41         ...
```

FIG. 9.8: Une implémentation optimisée.

Certains des états qui sont poussés sur la pile ne sont jamais observés. Une observation de la table *goto* montre que les seuls états qui sont consultés par l'opération « reduce S » sont 0, 4, 6 et 7. En effet, tous les autres états ont des lignes vides dans cette table. En résumé, il n'est pas pertinent de pousser les états 1, 2, 8 et 9 sur la pile. (Les états 3, 5, 10 et 11 ne sont pas non poussés sur la pile car aucune action « shift k » ne les considère). Ainsi, comme nous effectuons une action « shift k » à partir des états 1, 2, 8 et 9, nous allouons une cellule qui ne contient pas d'état. Nigel Horspool et Michael Whitney (Horspool & Whitney, 1990) nomme cette idée « l'optimisation de l'empilement

minimal ».

Cette optimisation, ajoutée à notre décision précédente de ne pas stocker les valeurs sémantiques de type *unit* sur la pile, entraîne que certaines actions « shift k » ne demandent aucune allocation. Dans notre exemple, les transitions de ce type qui sortent des états 1, 2, 8 et 9 sont associées aux symboles terminaux dont les valeurs sémantiques sont de type *unit*. Quand l'une de ces transitions est empruntée, on ne modifie donc pas la pile, seulement l'état courant.

Une autre optimisation consiste à définir une version spécialisée de *run* pour chaque état : *run0*, *run1* et ainsi de suite. Ces fonctions spécialisées se voient attribuer des types qui reflètent la connaissance de la forme de la pile : par exemple, *run9* a le type $\forall\alpha.\alpha\ cE\ cP\ cT \to int$. Le paramètre s disparaît : les appels à *run S9* sont remplacés par des appels à *run9*. Cette optimisation est rendue possible par le fait que la fonction *run* est toujours appliquée à un état constant.

Une fois cette optimisation appliquée, les représentations de tous les états autres que 0, 4, 6 et 7 ne sont plus utiles donc les constructeurs de données associés sont supprimés. En fait, certaines de ces fonctions n'ont qu'un unique site d'appel et leur corps peut donc être mis en ligne. C'est le cas de *run8*, *run9*, *run10* et *run11*.

Finalement, nos définitions de types se réduisent à celles présentées dans la figure 9.8 (lignes 1–10). Tous les constructeurs de données exceptés *S0*, *S4*, *S6* et *S7* ont disparu. En outre, les formes de pile associées à ces quatres états ont été très largement simplifiées. Les cellules qui ne contenaient pas de valeurs sémantiques et contenaient les états de l'ensemble $\{1, 2, 8, 9\}$ ont disparu par la même occasion.

Un fragment du programme final est donné dans la figure 9.8 (lignes 13–41). La définition de *run9* (lignes 19–32) est mise en ligne en son unique site d'appel à l'intérieur de *gotoT*. L'action « shift k » effectue une transition vers l'état 7 (ligne 24) en invoquant *run7*. Aucune nouvelle celulle de pile n'est allouée parce que ni l'état 9, ni la valeur sémantique *()* ne sont utiles. Dans l'action « reduce », seules deux cellules de la pile sont dépilées (lignes 26–27) parce que la cellule intermédiaire était associée au symbole + et ne contenait aucune information utile. La fonction auxiliaire *gotoE* reste inchangée excepté l'appel *run S1* qui est remplacé par *run1* (ligne 38) et l'appel *run S8* qui est remplacé par la version mise en ligne de *run8* (ligne 40), encore une fois parce que *run8* n'a pas d'autre site d'appel.

L'optimisation « goto direct » de Nigel Horspool et Michael Whitney (Horspool & Whitney, 1990) est obtenue sans effort : quand une fonction *goto* contient un *pattern matching* qui ne discrimine que sur un seul cas, le compilateur supprime tout test dynamique.

9.10 Conclusion

Nous avons expliqué comment l'extension des types algébriques généralisés permet d'exprimer des analyseurs syntaxiques LR à la fois efficaces et sûrs. La sûreté obtenue est plus forte que celle considérée habituellement en ML puisqu'elle garantit que le programme généré s'évalue sans erreur et ne peut pas échouer. La performance, quant à elle, est obtenue par la suppression des tests dynamiques redondants qui sont nécessaires pour rendre le programme bien typé si on n'a pas les types algébriques généralisés. Notons, par ailleurs, que notre codage des propriétés dans les types est stable à travers les optimisations que nous avons implémentées, ce qui montre sa pertinence. Nous trouvons ce résultat particulièrement amusant et caractéristique de l'expressivité acquise grâce aux types algébriques généralisés.

9.11 Quelques utilisations connues des GADT

Dans cette section, nous faisons une liste non exhaustive des applications connues des GADT dans la littérature.

► **Origines des GADT (Pfenning & Lee, 1989)** Si la formalisation des GADT a dû attendre l'article de Hongwei Xi, les idées qui consistent à indicer le type des termes par leur type objet dans un

évaluateur et d'avoir un raffinement des types dans chaque branche de cet évaluateur en fonction du cas syntaxique traité apparaît dans un article de Franck Pfenning et Peter Lee (1989). Ils utilisent un codage de Church classique pour encoder le match dans un langage polymorphique d'ordre 2 mais ils donnent à chaque branche un type plus précis pour effectuer un raffinement.

▶ **λ-calcul simplement typé (Xi et al, 2003, Sheard, 2004)** On peut généraliser le langage $\mathcal{T}$ de notre introduction pour traiter le λ-calcul simplement typé. Cette implémentation repose sur une représentation des variables à l'aide des indices de De Bruijn (de Bruijn, 1972) et sur le codage de l'environnement de typage dans un indice (sous la forme d'une liste de types, le type en position i correspondant au type de la variable i). La littérature mentionne des implémentations utilisant la syntaxe d'ordre supérieur (Xi *et al*, 2003, Donnelly & Xi, 2005) ou des sortes spécifiques pour les variables (Sheard, 2005a) mais, assez curieusement, on ne trouve pas l'implémentation suivante pourtant très simple.

$$
\begin{array}{l}
\mathsf{type}\ term : \star \to \star \to \star = \\
\quad \mid Var : \forall env\ ty.indice\ env\ ty \to term\ env\ ty \\
\quad \mid App : \forall env\ ity\ oty.term\ env\ (ity \to oty) \to term\ env\ ity \to term\ env\ oty \\
\quad \mid Fun : \forall env\ a\ b.term\ (a \times env)\ b\ \to term\ env\ (a \to b)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{type}\ indice : \star \to \star \to \star = \\
\quad \mid Zero : indice\ (h \times t)\ t \\
\quad \mid Shift : indice\ h\ t \to indice\ (h \times s)\ t
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let\ rec}\ lookup : \forall env\ ty.indice\ env\ ty \to env \to ty = \lambda i\ env. \\
\quad \mathsf{match}\ i\ \mathsf{with} \\
\quad\quad \mid Zero \to fst\ env \\
\quad\quad \mid Shift\ i \to lookup\ i\ (snd\ env)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let\ rec}\ eval : \forall env\ ty.term\ env\ ty \to env \to ty = \lambda t\ env. \\
\quad \mathsf{match}\ t\ \mathsf{with} \\
\quad\quad \mid Var\ i \to lookup\ i\ env \\
\quad\quad \mid App\ t1\ t2 \to (eval\ t1\ env)\ (eval\ t2\ env) \\
\quad\quad \mid Fun\ t \to \lambda x.eval\ t\ (x, env)
\end{array}
$$

La fonction *lookup* sert à obtenir la valeur d'un indice dans l'environnement d'évaluation. Comme le type de l'environnement est synchronisé avec l'ensemble des indices apparaissant dans le terme, il suffit d'itérer sur l'indice et sur l'environnement pour obtenir la valeur associée à l'indice. La fonction *eval* maintient l'invariant de synchronisation de l'environnement et des variables libres du terme en augmentant l'environnement à chaque λ traversé.

▶ **Typage de Yampa (Nilsson, 2005)** Yampa est un langage spécifique embarqué dans Haskell à l'aide de combinateurs. Il fournit un ensemble d'outils pour la manipulation de signaux dans le cadre de la programmation fonctionnelle réactive. Un traitement sur les signaux est de type $\mathsf{SF}\ \alpha\ \beta$ où α et β correspondent respectivement aux types d'entrée et de sortie des signaux. Les traitements sont composés à l'aide de combinateurs. Le plus simple de ces combinateurs est $\mathsf{>>>} :: \mathsf{SF}\ \alpha\ \beta \to \mathsf{SF}\ \beta\ \gamma \to \mathsf{SF}\ \alpha\ \gamma$, il construit une séquence formée de deux traitement mis bout à bout. Le type SF est un GADT ce qui permet par exemple de donner au constructeur de données de l'identité SFId le schéma de type $\mathsf{SF}\ \alpha\ \alpha$. Dès lors, dans la définition du combinateur >>>, une optimisation qui supprime l'identité dans une séquence est bien typée :

$$
\begin{array}{l}
\mathsf{>>>} :: \mathsf{SF}\ \alpha\ \beta \to \mathsf{SF}\ \beta\ \gamma \to \mathsf{SF}\ \alpha\ \gamma \\
\ldots \\
\mathsf{SFId} \mathsf{>>>} s = s
\end{array}
$$

Cette définition est bien typée car dans la branche montrée ci-dessus $\alpha = \beta$ donc s a le type $\mathsf{SF}\ \alpha\ \gamma$.

▶ **Défonctionnalisation typée (Pottier & Gauthier, 2004)** La défonctionnalisation est la transformation d'un programme d'ordre supérieur en un programme du premier ordre par la suppression des fonctions de première classe. Ces dernières sont remplacées par des fermetures, des données décrivant, à l'aide d'une étiquette, une fonction *toplevel* à appeler et contenant un environnement de valeurs pour les variables libres et les arguments effectifs. Chaque appel de fonctions est remplacé par un appel à une fonction *apply* qui interprète les données de la fermeture pour appeler la fonction voulue dans le bon contexte. Pour maintenir le bon typage du programme transformé, les étiquettes des fermetures sont définies par un GADT de manière à coder le type des fonctions de première classe qu'on remplace.

Cette approche s'appuyant sur des GADT a été utilisée pour obtenir des garanties de préservation du typage de la transformation *closure conversion* par Stefan Monnier dans l'implémentation d'un compilateur préservant les types en Haskell (Monnier & Guillemette, 2007).

▶ **Polymorphisme intensionnel et programmation générique (Xi et al, 2003, Crary et al, 2002, Hinze, 2003)** Une des utilisations les plus courantes des GADT est de fournir une représentation dynamique des types statiques. Un test dynamique sur la représentation du type α permet d'obtenir une information statique α. On peut ainsi écrire des fonctions génériques, ou polytypiques, en observant un descripteur dynamique de type.

▶ **Simulation d'objets (Xi et al, 2003)** On peut modéliser un objet comme une fonction qui interprète sous la forme de valeur les messages qu'on lui envoie. Pour donner un type à cette fonction, on utilise le GADT $\mathsf{msg}\ \alpha$ des messages dont la réponse est de type α. Un objet est alors de type $\forall\alpha.\mathsf{msg}\ \alpha \to \alpha$. On implémente très simplement à l'aide d'un *pattern matching* sur le message reçu.

▶ **Arbres rouges-noirs (Sheard, 2005a)** Notre application sur les analyseurs LR (Pottier & Régis-Gianas, 2006b) montre qu'il est possible de coder des invariants algorithmiques à l'intérieur des indices des GADT. Une autre illustration est l'implémentation des arbres rouges-noirs (Cormen *et al*, 2001) à l'aide de GADT. Voici le type des arbres rouges-noirs :

$$
\begin{array}{l}
\mathsf{type}\ \mathit{zero} : \star \\
\mathsf{type}\ \mathit{succ} : \star \to \star \\
\mathsf{type}\ \mathit{red} : \star \\
\mathsf{type}\ \mathit{black} : \star \\
\\
\mathsf{type}\ \mathit{subtree} : \star \to \star \to \star = \\
\quad \mid \mathit{Leaf} : \mathit{subtree\ black\ zero} \\
\quad \mid \mathit{RedNode} : \forall n.\mathit{subtree\ black\ n} \to \mathit{int} \to \mathit{subtree\ black\ n} \to \mathit{subtree\ red\ n} \\
\quad \mid \mathit{BlackNode} : \forall n\ l\ r.\mathit{subtree\ l\ n} \to \mathit{int} \to \mathit{subtree\ r\ n} \to \mathit{subtree\ red\ (succ\ n)} \\
\quad \mid \mathit{Fix} : \forall n.\mathit{subtree\ red\ n} \to \mathit{subtree\ black\ n} \\
\\
\mathsf{type}\ \mathit{tree} : \star = \mathit{Tree} : \forall n.\mathit{subtree\ black\ n} \to \mathit{tree}
\end{array}
$$

On peut vérifier que ce type code les invariants des arbres rouges-noirs suivants :
- Un nœud est soit rouge, soit noir.
- La racine d'un arbre est toujours noire.
- Toutes les feuilles sont noires.
- Si un nœud est rouge alors tous ses descendants sont noirs.
- Tous les chemins des feuilles à la racine contiennent le même nombre de nœuds noirs.

▶ **File purement fonctionnelle a la Kaplan** Avec Nadji Gauthier, nous avons implémenté une version de la première structure de données de file purement fonctionnelle proposée par Haim Kaplan (1999). Pour parvenir à garantir des opérations en temps constant (push, pop, inject, eject), cette structure de données utilise des invariants assez complexes. En particulier, le type des éléments stockés par la file n'est pas homogène dans ses sous-structures. Pour être plus précis, la file est implémenté par une pile de bloc. Chaque bloc est une liste dont le type des éléments croît de manière exponentielle (le premier niveau est de type α, le second de type $\alpha \times \alpha$, le troisième $(\alpha \times \alpha) \times (\alpha \times \alpha)$ ... etc). Il est important que chaque bloc connaisse le type des éléments du bloc suivant. Or, ce type est l'exponentielle de la longueur de ce bloc. Grâce à un GADT, on peut coder cet invariant statiquement. Nous avons implémenté une version totalement certifiée de cet algorithme dans un prototype de système de preuves pour la programmation fonctionnelle.

▶ **Structure de données coordonnées (Ringenburg & Grossman, 2005)** La coordination entre l'état de l'automate et la forme de sa pile vue dans le chapitre 9 est un exemple de structures coordonnées décrites par Michael Ringenburg et Dan Grossman (2005). Dans le cas général, il s'agit de maintenir un invariant entre des données physiquement disjointes. Les auteurs prennent l'exemple de deux listes. L'une contient des éléments de types hétérogènes et l'autre contient des fonctions dont *les types des arguments sont synchronisés avec les types des éléments de la première liste*. Ils proposent un système de types dédiés à ce problème qui introduit une forme généralisée de types existentiels. Nous avons montré que leur système est subsumé par les GADT.

DEUXIÈME PARTIE

Preuve de programmes fonctionnels

CHAPITRE DIX

Introduction

Comme nous l'avons expliqué dans l'introduction de cette thèse, le pouvoir d'expression des GADT sont limitées pour dénoter des invariants. Pour pallier à cette limitation, plutôt que d'enrichir le langage des types, nous avons choisi d'étudier l'ajout de formules logiques à un langage de programmation fonctionnel pur à l'aide d'une logique de Hoare, méthode habituellement appliquée aux langages impératifs.

La logique de Hoare (Floyd, 1967, Hoare, 1969, Cousot, 1990) est une discipline qui consiste à annoter les programmes avec des formules logiques, appelées « assertions », et à extraire des formules logiques, appelées « obligations de preuve » à partir d'un tel programme annoté. La validité des obligations de preuve, qui peut être vérifiée soit manuellement, soit mécaniquement, entraîne la correction du programme annoté. Cela signifie qu'elle garantit que les assertions sont des prédictions statiques correctes du comportement dynamique du programme. Le processus de construction et de vérification des obligations de preuve est quelques fois connu sous le nom de « vérification statique étendue » (Detlefs *et al*, 1998b).

La logique de Hoare était conçue initialement pour un langage « à boucles while » c'est-à-dire un langage impératif équipé d'une construction d'itération et d'un nombre fixé de variables globales et modifiables. Les fonctions récursives et d'ordre supérieur ont été le sujet de beaucoup d'attention dans les années 70 et au début des années 80 (Clarke, 1979, Apt, 1981, Damm & Josko, 1983, German *et al*, 1983, Goerdt, 1985). Plus récemment, les structures de données modifiables et allouées dynamiquement ainsi que les fonctionnalités liées aux langages orientés objets ont été énormément étudiées. Ceci a conduit au développement d'outils dédiés à des langages comme C (Filliâtre & Marché, 2004), Java (Burdy *et al*, 2005, Flanagan *et al*, 2002, Marché *et al*, 2004, Beckert *et al*, 2007), et C# (Barnett *et al*, 2004a).

Nous voulons mettre en avant la thèse que les systèmes de preuve pour les langages impératifs sont plus complexes à mettre en oeuvre que ceux dédiés aux langages fonctionnels. La simplicité accrue des langages fonctionnels permet de dépenser plus d'énergie pour s'attaquer à des propriétés clés nécessaires au passage à l'échelle de ces méthodes comme la modularité ou l'abstraction. Une question résume notre point de vue : étant donné que les programmes fonctionnels sont connus pour être plus simples à prouver corrects, pourquoi cette activité n'est pas devenu une routine quotidienne, quarante ans après les papiers fondateurs de Robert Floyd et C.A.R. Hoare ?

10.1 Sur le coût de la programmation impérative

La thèse affirmant que la programmation fonctionnelle est supérieure à la programmation impérative n'est pas nouvelle (Hughes, 1989). Cependant, il est bon de rappeler quelques difficultés inhérentes

au raisonnement sur les programmes travaillant sur un état modifiable.

Dans un langage impératif moderne, tous les objets alloués dans le tas sont modifiables. Il en découle qu'un raisonnement à l'aide d'entités abstraites comme des paires, des listes ou des arbres, etc., est remplacé par un raisonnement de plus bas niveau s'appuyant sur des graphes, des chemins, etc. Plus concrètement, le programmeur est contraint d'écrire et de prouver des formules mettant en jeu des associations des adresses en mémoire vers les blocs en mémoire (Mehta & Nipkow, 2005, Filliâtre & Marché, 2004).

La possible présence d'*alias* signifie que, du moment où un bloc de mémoire quelconque est modifié, la mémoire qui est accessible à travers n'importe quel pointeur sur ce bloc est potentiellement affectée. De manière à garder sous contrôle ces effets de bord, on introduit en général des mécanismes pour affirmer des propriétés de séparation, indiquant que certaines paires de pointeurs ne travaillent pas dans la même zone mémoire. Ces assertions peuvent être soit explicites (auquel cas très lourdes) ou bien implicites (Reynolds, 2002).

Quand la modularité et les traits orientés objet rentrent en scène, les choses deviennent encore plus compliquées. Il est souvent agréable pour un module ou un objet de maintenir un état privé (local) conjointement à un invariant privé. Vérifier que l'état privé reste privé et que cet invariant ne peut pas être violé est un problème non trivial connu sous le nom d'« exposition de la représentation » (Detlefs *et al*, 1998a, Leino & Nelson, 2002). Quelques systèmes de type basés par exemple sur la notion de « type d'appartenance » (Clarke *et al*, 1998) ou sur les types linéaires et les régions (Fähndrich & DeLine, 2002) ont été conçus pour proposer des solutions à ces problèmes.

10.2 Objectifs

Nous n'affirmons pas que les problèmes mentionnés plus hauts ne méritent pas d'être étudiés : au contraire, ils sont fascinants. Néanmoins, il est dommage que nous n'ayons pas aujourd'hui des outils vérifiant la correction des programmes fonctionnels. Voilà pourquoi nous avons décidé d'étudier la logique de Hoare dans le cadre des programmes sans état. Traiter les programmes ML idiomatiques qui exploitent l'état modifiable avec parcimonie et discipline est une voie de recherche possible par la suite.

Les programmes que nous voulons prouver ici s'appuient fortement sur des fonctions (éventuellement d'ordre supérieur), des structures de données algébriques et du polymorphisme. Nous pensons qu'il est plus simple d'extraire des obligations de preuve courtes et naturelles à partir de ces programmes quand ils sont annotés par des spécifications.

L'élimination de l'état a deux intérêts principaux. Pour l'utilisateur, cela conduit à des spécifications et des obligations de preuve plus simples. Le développeur du système peut se focaliser sur le polymorphisme, l'abstraction de type et la modularité. L'importance de ces propriétés ne doit pas être sous-estimée car la capacité à développer et à vérifier des composants de programmes indépendamment augmente l'efficacité du développement.

Aujourd'hui, il n'existe pas à notre connaissance d'outil qui permet d'extraire des obligations de preuve à partir de programmes fonctionnels décorés par des assertions logiques. On reviendra sur les travaux existants dans le chapitre 16. Nous voulons combler ce manque.

10.3 Contribution

Cette partie va présenter la conception d'un langage polymorphiquement typé et d'ordre supérieur où les programmes sont annotés par des assertions exprimées dans une logique typée, polymorphe et d'ordre supérieur. Nous définissons une procédure pour extraire des obligations de preuve des programmes et nous montrons que ce processus est correct. Cette procédure a été implémentée dans un prototype qui a servi à prouver des programmes non triviaux.

10.4 Grandes lignes de notre approche

On décrit maintenant quelques aspects techniques de notre approche.

Nous nous focalisons sur la correction partielle. En particulier, on accepte des programmes qui ne terminent pas forcément. On pourrait très bien restreindre le langage de programmation pour contraindre la terminaison mais nous estimons que forcer la terminaison de tout programme est abusif dans le cas d'un langage de programmation généraliste. C'est à l'utilisateur de déterminer quelles sont les propriétés du code qui nécessite une preuve et d'insérer des assertions logiques le cas échéant. Un programmeur peut très bien écrire des programmes sans assertion et ne pas avoir d'obligations de preuve à traiter. Il n'y a donc aucun coût à utiliser notre méthodologie.

Les valeurs, les programmes, les types et les formules logiques appartiennent à des catégories syntaxiques distinctes. Les preuves n'apparaissent pas dans les programmes : les obligations de preuve sont déléguées à un prouveur externe (qui doit donc faire partie de la base de confiance si il ne produit pas de termes de preuve).

Nous n'immisçons pas les valeurs, les programmes ou les formules dans les types. Nos types sont donc des termes du premier ordre : ils comptent parmis eux les variables de type, les types algébriques paramétrés, les types fonctionnels, tout comme en ML. En conséquence, l'inférence de type dans le style de Milner (Milner, 1978) est possible. L'inférence de type ne produit donc aucune obligation de preuve.

Le corollaire de ce choix est l'absence de types dépendants, tels que le type des listes indexé par leurs tailles (Xi, 1998) mais nous les simulons comme suit : plutôt que déclarer qu'une variable x a le type $\mathit{list}\, n$, nous disons que x a le type list et nous affirmons la formule logique $\mathit{length}(x) = n$. Ici, la fonction length doit être définie inductivement au niveau logique.

Les formules peuvent faire référence aux valeurs mais pas aux programmes. C'est important car les valeurs sont pures alors que les programmes peuvent être potentiellement impurs. Bien que notre logique ne puisse raisonner explicitement sur l'état, il est toutefois correct d'évaluer des programmes qui mettent en jeu de la non-terminaison, du non-déterminisme, des entrées-sorties ou un état modifiable (lire un flux d'entrée ou déréférencer un pointeur peuvent être vues comme des opérations non-déterministes). Dans ce cas, il est toujours possible d'établir des propriétés qui ne dépendent pas du comportement des opérations impures. Cela signifie que, par exemple, on peut prouver la correction d'un programme fonctionnel même si il a été instrumenté avec des instructions impures de débogage, chronométrage ou affichage.

Dans notre langage de programmation, les fonctions qui sont potentiellement impures sont des valeurs donc elles peuvent apparaître dans les formules. Que signifie la référence dans une formule à une fonction calculatoire f du type, disons, $\tau_1 \dot{\longrightarrow} \tau_2$? Notre réponse est de voir f dans le monde logique sous la forme d'une paire de prédicats qui représente la précondition et la postcondition de f. Autrement dit, quand on l'utilise à l'intérieur d'une formule, f a le type $(\tau_1 \to \mathsf{prop}) \times (\tau_1 \to \tau_2 \to \mathsf{prop})$. Les deux projections de paire, écrite pre et post, peuvent être utilisée pour faire référence à chaque composante de la paire. Ainsi, $\mathsf{pre}(f)$ and $\mathsf{post}(f)$ sont des notations légères pour faire référence à la précondition et à la postcondition de f. Quand f est une fonction liée localement par un let, ce mécanisme peut être vu comme le nommage des spécifications de la fonction par des abréviations. Quand f est inconnue car liée par un λ ou un $\forall$, ces opérateurs sont très utiles pour écrire des spécifications de fonctions d'ordre supérieur.

Pour résumer, bien que les outils techniques que nous exploitons soient standard, nous croyons qu'il y a à gagner à attirer l'attention sur la combinaison de simplicité et de puissance qu'offrent nos choix techniques. Quand il est étendu avec un système de module adapté et avec un schéma de compilation vers, par exemple, O'Caml alors notre outil peut être utilisé pour vérifier la correction de composants purement fonctionnels qui peuvent être utilisés dans de plus grands programmes éventuellement impératifs.

10.5 Plan

Cette partie est organisée comme suit. Tout d'abord, nous introduisons rapidement la logique d'ordre supérieur dans laquelle les assertions et les obligations de preuve vont être exprimées (chapitre 11). Ensuite, nous présentons la syntaxe et la sémantique d'un langage noyau fonctionnel dont les expressions portent des assertions logiques (chapitre 12). Nous décrivons son système de type ainsi qu'une procédure d'extraction des obligations de preuve à partir des programmes (chapitre 13) et nous discutons de la manière dont ces obligations de preuve sont transformées pour être convoyées vers des prouveurs externes. Nous donnerons ensuite quelques exemples. Enfin, nous discuterons de quelques extensions utiles de ce langage noyau (chapitre 14) que nous utiliserons dans une application concrète implémentant des arbres binaires équilibrés (chapitre 15). Nous présenterons enfin les travaux existants dans ce domaine (chapitre 16).

CHAPITRE ONZE

La logique sous-jacente

Dans ce chapitre, on décrit la logique d'ordre supérieure polymorphiquement typée (notée PHOL) dans laquelle sont exprimées les assertions logiques du système. Cette logique est standard et ses propriétés méta-théoriques ont été étudiées dans la littérature (Andrews, 1986). Le fait de ne pas utiliser une logique exotique (contrairement à d'autres approches (Kieburtz, 2002, Honda & Yoshida, 2004)) permet le traitement des obligations de preuve par des prouveurs automatiques pour la logique du premier ordre (après un codage standard de l'ordre supérieur vers le premier ordre) et par des assistants de preuve comme Coq.

11.1 Syntaxe

Nous nous appuyons sur une logique standard d'ordre supérieur (Andrews, 1986) dont les types et les termes sont donnés par la figure 11.1. La seule particularité de notre logique est son typage polymorphe. Les types θ incluent les variables de type, les types inductifs paramétrés, les types fonctionnels, les types produits et le type prop des propositions logiques. Dans la suite, la syntaxe des termes est étendue par les sucres syntaxiques habituels pour la proposition fausse, la disjonction, l'implication, l'équivalence, la quantification existentielle, etc.

Les règles de typage apparaissent dans la figure 11.3. En général, nous écrivons t pour les termes d'un type arbitraire. Nous écrivons F pour les formules, c'est-à-dire les termes de type prop et P pour les prédicats c'est-à-dire les termes de type $\theta \to \mathsf{prop}$.

Notre logique n'est pas simplement typée. Comme notre langage de programmation (chapitre 12) est polymorphe et comme nous voulons injecter toutes les valeurs calculatoires au niveau logique, la logique doit elle aussi être polymorphe. Pour cette raison, nous avons des schémas de type logique $\varsigma ::= \forall \overline{\alpha}.\, \theta$ et chaque utilisation d'une variable x est annotée explicitement par un vecteur de type $\overline{\theta}$ indiquant la manière dont le schéma de type associé à x doit être instancié.

La syntaxe des formules intègre la quantification universelle sur les variables de type qui permet d'énoncer une propriété valide quelque soit le type. Par exemple, si la relation logique id implémente l'identité polymorphe et si elle a le schéma de type $\forall \alpha.\, \alpha \to \alpha \to \mathsf{prop}$, la formule suivante est bien formée et valide :

$$\forall \alpha. \forall (x_1 : \alpha). \forall (x_2 : \alpha). \big(id\ \alpha\ (x_1)\ (x_2) \Leftrightarrow x_1 = x_2 \big)$$

Une telle quantification universelle peut être instanciée arbitrairement : par exemple, une des conséquences de la formule précédente est :

$$\forall (x : int). id\ int\ (x)\ (x).$$

			Types logiques
θ	$::=$	α	*Variable*
	$\mid$	$\varepsilon\ \overline{\theta}$	*Données*
	$\mid$	$\theta \to \theta$	*Fonction*
	$\mid$	$\theta \times \theta$	*Produit*
	$\mid$	prop	*Proposition*
ς	$::=$	$\forall\overline{\alpha}.\,\theta$	*Schéma de type*
			Environnement de type logique
Δ	$::=$	$\emptyset$	*Nil*
	$\mid$	$\Delta, (x : \varsigma)$	*Variable*
	$\mid$	$\Delta, \overline{\alpha}$	*Variables de type*
			Termes logiques
t, F, P	$::=$	$x\,\overline{\theta}$	*Variable*
	$\mid$	$D\,\overline{\theta}\ (t, \ldots, t)$	*Données*
	$\mid$	$\lambda(x : \theta).t$	*Abstraction*
	$\mid$	$t\ (t)$	*Application*
	$\mid$	(t, t)	*Produit*
	$\mid$	π_1	*Projection (écrite aussi* pre*)*
	$\mid$	π_2	*Projection (écrite aussi* post*)*
	$\mid$	true	*Vérité*
	$\mid$	$t = t$	*Égalité*
	$\mid$	$t \wedge t$	*Conjonction*
	$\mid$	$\neg\, t$	*Négation*
	$\mid$	$\forall(x : \varsigma).t$	*Quantification universelle*
	$\mid$	$\forall\overline{\alpha}.t$	*Quantification sur les types*

FIG. 11.1: La logique (syntaxe).

Par souci de clarté, nous écrivons $id\,\alpha$ et $id\,int$ à la place de id uniquement. Dans le langage réel, ces annotations de type peuvent être omises grâce à l'inférence de type dans le style de Hindley-Milner qui est appliquée à la fois sur le code et sur les formules.

La logique possède des types inductifs. Nous supposons que chaque constructeur de type inductif ε est accompagné d'une arité entière fixée et que chaque application $\varepsilon\ \overline{\theta}$ la respecte. Nous supposons de plus que ε est associé à un nombre fini de constructeurs de données D auxquels sont associés des schémas de type de la forme :

$$\forall\overline{\alpha}.\,\theta_1 \to \ldots \to \theta_n \to \varepsilon\ \overline{\alpha}$$

Nous forçons les schémas de type associés à ces constructeurs à remplir une condition de positivité (Paulin-Mohring, 1992) qui est informellement résumée comme suit : dans le schéma de type suivant, le constructeur de type ε (ou n'importe quel constructeur de type dont la définition est mutuellement récursive avec la définition de ε) ne doit pas apparaître à gauche d'une flèche à l'intérieur de $\theta_1, \ldots, \theta_n$. La caractérisation formelle de la stricte positivité s'appuie sur la définition des occurrences dans un type θ, positives et négatives, des constructeurs de type $\varepsilon_1, \ldots \varepsilon_n$ définis mutuellement.

Définition 11.1.1 (Occurrences positives et négatives de constructeurs de type)
L'ensemble des occurrences positives (respectivement négatives), noté Occ^+_T *(respectivement* Occ^-_T*),*

$$\begin{array}{rcll}(\Delta,(x:\varsigma))(x) & = & \varsigma & \\ (\Delta,(x_1:\varsigma))(x_2) & = & \Delta(x_2) & \text{if } x_1 \# x_2 \\ (\Delta,\overline{\alpha})(x) & = & \Delta(x) & \text{if } \overline{\alpha} \# \Delta(x)\end{array}$$

FIG. 11.2: Accès à l'environnement logique.

$$\frac{\Delta(x) = \forall\overline{\alpha}.\,\theta}{\Delta \vdash x\,\overline{\theta} : [\overline{\alpha} \mapsto \overline{\theta}]\theta} \qquad \frac{\begin{array}{c}D : \forall\overline{\alpha}.\,\theta_1 \to \ldots \to \theta_n \to \varepsilon\ \overline{\alpha} \\ \forall i \qquad \Delta \vdash t_i : [\overline{\alpha} \mapsto \overline{\theta}]\theta_i\end{array}}{\Delta \vdash D\,\overline{\theta}\ (t_1,\ldots,t_n) : \varepsilon\ \overline{\theta}} \qquad \frac{\Delta,(x:\theta_1) \vdash t : \theta_2}{\Delta \vdash \lambda(x:\theta_1).t : \theta_1 \to \theta_2}$$

$$\frac{\Delta \vdash t_1 : \theta_1 \to \theta_2 \quad \Delta \vdash t_2 : \theta_1}{\Delta \vdash t_1\ (t_2) : \theta_2} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \theta_i}{\Delta \vdash (t_1,t_2) : \theta_1 \times \theta_2} \qquad \frac{\Delta \vdash t : \theta_1 \times \theta_2}{\Delta \vdash \pi_i\ (t) : \theta_i} \qquad \frac{}{\Delta \vdash \mathsf{true} : \mathsf{prop}} \qquad \frac{\forall i \qquad \Delta \vdash t_i : \theta}{\Delta \vdash t_1 = t_2 : \mathsf{prop}}$$

$$\frac{\forall i \qquad \Delta \vdash t_i : \mathsf{prop}}{\Delta \vdash t_1 \wedge t_2 : \mathsf{prop}} \qquad \frac{\Delta \vdash t : \mathsf{prop}}{\Delta \vdash \neg\, t : \mathsf{prop}} \qquad \frac{\Delta,(x:\varsigma) \vdash t : \mathsf{prop}}{\Delta \vdash \forall(x:\varsigma).t : \mathsf{prop}} \qquad \frac{\Delta,\overline{\alpha} \vdash t : \mathsf{prop}}{\Delta \vdash \forall\overline{\alpha}.t : \mathsf{prop}}$$

FIG. 11.3: La logique (système de type).

d'un ensemble de constructeurs de type $\mathcal{T} = \varepsilon_1,\ldots,\varepsilon_n$ au sein d'un type θ sont définies inductivement ainsi :

$$\begin{array}{lcll}Occ^+_{\mathcal{T}}(\alpha) & = & \emptyset & \\ Occ^+_{\mathcal{T}}(\varepsilon\,\overline{\theta}) & = & \{\epsilon\} & \textit{si } \varepsilon \in \mathcal{T} \\ Occ^+_{\mathcal{T}}(\varepsilon\,\overline{\theta}) & = & \emptyset & \textit{sinon} \\ Occ^+_{\mathcal{T}}(\theta_1 \to \theta_2) & = & 1.Occ^-_{\mathcal{T}}(\theta_1) \cup 2.Occ^+_{\mathcal{T}}(\theta_2) & \\ Occ^+_{\mathcal{T}}(\theta_1 \times \theta_2) & = & 1.Occ^+_{\mathcal{T}}(\theta_1) \cup 2.Occ^+_{\mathcal{T}}(\theta_2) & \\ Occ^-_{\mathcal{T}}(\alpha) & = & \emptyset & \\ Occ^-_{\mathcal{T}}(\varepsilon\,\overline{\theta}) & = & \emptyset & \\ Occ^-_{\mathcal{T}}(\theta_1 \to \theta_2) & = & 1.Occ^+_{\mathcal{T}}(\theta_1) \cup 2.Occ^-_{\mathcal{T}}(\theta_2) & \\ Occ^-_{\mathcal{T}}(\theta_1 \times \theta_2) & = & 1.Occ^-_{\mathcal{T}}(\theta_1) \cup 2.Occ^-_{\mathcal{T}}(\theta_2) & \end{array}$$

On peut maintenant définir la stricte positivité requise pour la bonne formation de nos définitions de types algébriques.

Définition 11.1.2 (Bonne formation des définitions mutuelles d'ADTs)
Soit $\mathcal{T} = \{\varepsilon_1,\ldots,\varepsilon_n\}$, un ensemble de constructeurs de type définis mutuellement à l'aide de n familles de constructeurs de données notées $(D^k_i)_{\substack{k\in1,\ldots,n \\ i\in1,\ldots,m_k}}$ telles que $D^k_i : \forall\overline{\alpha}.\theta^{i,k}_1 \to \ldots \to \theta^{i,k}_{p_{i,k}} \to \varepsilon_k\ \overline{\alpha}$. Ces définitions sont bien formées si et seulement si $\forall i,j,k.Occ^-_{\mathcal{T}}(\theta^{i,k}_j) = \emptyset$ ◇

Bien qu'il y ait une forme d'introduction pour les types inductifs – l'application des constructeurs de données D – il n'y a pas de forme d'élimination. Un mécanisme pour définir des fonctions sur ces types est essentiel en pratique mais il n'est pas nécessaire pour générer les obligations de preuve donc il est omis pour le moment.

11.2 Interprétation

Nous supposons que la logique est munie d'une interprétation cohérente à l'aide d'un modèle (Andrews, 1986) de manière à ce que toute formule close, valide et bien formée s'évalue dans le modèle en la valeur vraie.

Nous supposons que le modèle est compatible avec l'égalité. En particulier, il est toujours compatible avec la réflexivité.

Propriétés du modèle 1 *Dans le modèle, l'égalité est réflexive, transitive et symétrique.*

Nous supposons que le modèle est compatible avec la $\beta\eta$-équivalence.

Propriétés du modèle 2 *Si le modèle valide* F_1 *et* $F_1 =_{\beta\eta} F_2$ *alors il valide aussi* F_2*.*

Enfin, le modèle interprète les valeurs de types algébriques de façon à être compatible avec les axiomes suivants :

Propriétés du modèle 3 *Dans le modèle,* $D\ (v_1, \ldots, v_n) = D\ (v'_1, \ldots, v'_n)$ *est équivalent à pour tout* $i \in 1 \ldots n$*,* $v_i = v'_i$*.*

Propriétés du modèle 4 *Si le type algébrique* ε *est défini par* n *constructeurs de données* $D_1, \ldots, D_n$ *de schémas de types respectifs* $\forall\overline{\alpha}.\theta_1^1 \times \ldots \theta_{m_1}^1 \to \varepsilon(\overline{\alpha}), \ldots, \forall\overline{\alpha}.\theta_1^n \times \ldots \theta_{m_n}^n \to \varepsilon(\overline{\alpha})$ *alors le modèle valide*

$$\forall\overline{\alpha}.\forall(x : \varepsilon(\overline{\alpha})).\bigcup_{i=1}^{n} \exists(x_1 : \theta_1^i, \ldots, x_{m_i} : \theta_{m_i}^i).x = D_i(x_1, \ldots, x_{m_i})$$

Propriétés du modèle 5 *Pour les tous les constructeurs* D_i *du type algébrique* ε*, on a pour tout* i *et* j *distincts, le modèle qui valide* $D_i\ v_1 \ldots v_n \neq D_j\ v'_1 \ldots v'_m$

En somme, le modèle doit donc se comporter comme une Σ-algèbre multi-sortée.

11.3 Définition de fonctions logiques et de prédicats

Les règles de génération des obligations de preuve que nous verrons dans le chapitre 13 ne nécessitent pas l'introduction à proprement parler de prédicats inductifs ou de fonctions logiques. En effet, les prédicats seront traités de manière opaque, sans interprétation.

Néanmoins, pour pouvoir traiter les obligations de preuves à l'aide d'un prouveur externe, on doit fournir la définition des prédicats et des fonctions logiques. Pour être compatible avec Coq, nous traitons un sous-ensemble des fonctions et des prédicats inductifs du calcul des constructions inductives.

Les fonctions logiques sont définies par la syntaxe :

$$\begin{array}{l}\mathsf{logic\ fun}\ f : \forall\overline{\alpha}.\theta_1 \to \theta_2 = \\ \quad \mid p_1 \to t_1 \\ \quad \mid \ldots \\ \quad \mid p_n \to t_n\end{array}$$

Plusieurs propriétés doivent être vérifiées pour que cette définition soit bien fondée. D'une part, la récursion doit être structurelle (les appels récursifs à la fonction f doivent être faits sur des sous-termes stricts de l'argument). D'autre part, on doit s'assurer que la fonction est totale. Pour cela, on exige une clause pour chaque cas possible du type algébrique. Ces restrictions sont assez fortes mais elles ne sont pas une limitation très importante pour une première version du système.

La syntaxe des prédicats inductifs est :

$$\begin{array}{l}\mathsf{inductive}\ p : \forall\overline{\alpha}.\overline{\theta} \to \mathsf{prop} = \\ \quad \mid F_1 \\ \quad \mid \ldots \\ \quad \mid F_n\end{array}$$

Chaque cas F_i du prédicat inductif doit avoir comme conclusion une application du prédicat p. De plus, chaque occurrence de p dans ces formules doit être strictement positive à la manière des prédicats inductifs de Coq (Paulin-Mohring, 1992).

11.4 Interfaçage avec COQ

Notre logique typée et d'ordre supérieur est incluse dans le calcul des constructions inductives qui est la logique sous-jacente de Coq (The Coq development team, 2006). Exporter une obligation de preuve vers Coq est donc une simple affaire de formatage.

Coq est un prouveur interactif. Pour décharger l'obligation de preuve, l'utilisateur doit écrire un script de preuve. Un problème ouvert est la maintenance de ces scripts tout au long de l'évolution du programme. La localisation des obligations de preuve peut changer légèrement ainsi que leur énoncé. Peut-être qu'une solution consiste à ne prouver manuellement que les lemmes qui sont explicitement énoncés et explicitement nommés et à utiliser un prouveur automatique pour toutes les obligations de preuve restantes. Il est alors plus simple de maintenir une correspondance entre les obligations de preuve et les fichiers Coq créés par l'utilisateur.

11.5 Interfaçage avec la logique du premier ordre polymorphique

11.5.1 Motivations

Ergo (Conchon & Contejean, 2006) est un prouveur totalement automatique de théorèmes énoncés dans une logique typée, polymorphe et du premier ordre (notée PFOL). Sa conception est en partie inspirée de Simplify (Detlefs *et al*, 2005). Par contre, la logique d'Ergo est typée et polymorphe contrairement à celle de Simplify qui est non typée. De notre point vue, cela rend la logique plus intéressante que celle de Simplify. En effet, les obligations de preuve qui sont dans le fragment du premier ordre de notre logique peuvent être envoyées directement à Ergo.

En plus de la logique du premier ordre, Ergo a un support natif de l'arithmétique linéaire et de la théorie des constructeurs (c'est-à-dire des symboles de fonction tels que $f(x) = f(y)$ implique $x = y$). Ce point rend plus performant le raisonnement sur des structures de données algébriques.

Dans le cas général, nos obligations de preuve sont naturellement exprimées dans une logique d'ordre supérieur comme le montre notre présentation. Néanmoins, la logique d'ordre supérieur peut se coder dans la logique du premier ordre. Un codage standard introduit un prédicat « apply » qui aide à simuler la β-conversion (Kerber, 1991).

Dans notre cas, ce codage peut être ajusté pour sembler très naturel. Les symboles pre et post qui, jusqu'à maintenant, ont été vus comme des projections pour les paires peuvent être réinterprétés comme des prédicats et simuler l'application en plus de la projection. De plus, nous pouvons transformer pre en un prédicat binaire et post en un prédicat ternaire pour éviter l'application currifiée des fonctions. Pour récapituler, à la place des formules d'ordre supérieur :

$$f = (\lambda(x_1 : \theta_1).F_1, \lambda(x_1 : \theta_1).\lambda(x_2 : \theta_2).F_2),$$

on peut écrire :

$$\begin{array}{r}\forall(x_1 : \theta_1).(\mathsf{pre}(f, x_1) \Leftrightarrow F_1) \\ \wedge\ \forall(x_1 : \theta_1).\forall(x_2 : \theta_2).(\mathsf{post}(f, x_1, x_2) \Leftrightarrow F_2)\end{array}$$

			Types des objets
T	$::=$	α	*Variable de type*
	$\mid$	$\varepsilon\ (\overline{T})$	*Application de type*
			Prédicats
P	$::=$	true	*Proposition vraie*
	$\mid$	false	*Proposition fausse*
	$\mid$	$p(o, \ldots, o)$	*Application*
	$\mid$	$v = v$	*Égalité*
	$\mid$	$P \wedge P$	*Conjonction*
	$\mid$	$P \vee P$	*Disjonction*
	$\mid$	$\neg P$	*Négation*
	$\mid$	$\exists(x : T).P$	*Quantification existentielle*
	$\mid$	$\forall(x : T).P$	*Quantification universelle*
			Objets
o	$::=$	x	*Variables*
	$\mid$	$f\ (\overline{o})$	*Applications*
			Déclarations
$\mathcal{D}$	$::=$	$\mathsf{pred}\ p : \forall\overline{\alpha}.\overline{T} \to \mathsf{prop}$	*Prédicat*
	$\mid$	$\mathsf{val}\ v : \forall\overline{\alpha}.\overline{T} \to T$	*Valeur*
	$\mid$	$\mathsf{axiom}\ \forall\overline{\alpha}.P$	*Axiome*
	$\mid$	$\mathsf{type}\ \varepsilon\,[n]$	*Constructeur de type (et son arité)*

FIG. 11.4: Syntaxe de la logique du premier ordre polymorphe.

La paire et les trois λ-abstractions ont été η-expansées, et les projections et les applications de symboles ont été fusionnés dans les applications de pre et post. Si on suppose que F_1 et F_2 sont du premier ordre alors la formule obtenue est du premier ordre.

La section 11.5.2 présente la syntaxe et le système de type de la logique du premier ordre polymorphe que nous ciblons.

11.5.2 La logique du premier ordre polymorphe

La syntaxe de la logique du premier ordre polymorphe est rappelée par la figure 11.4. La différence essentielle entre PFOL et PHOL est la quantification qui ne peut pas se faire sur des prédicats dans PFOL. Notons aussi que la quantification sur les types de PFOL ne permet évidemment pas la quantification sur les types de prédicats.

11.5.3 Codage de PHOL vers PFOL

Codage du noyau

Nous utilisons la traduction décrite par Gilles Dowek, Thérèse Hardin et Claude Kirchner (Dowek *et al*, 2001). Il s'agit d'une sorte de défonctionnalisation (Reynolds, 1990, Pottier & Gauthier, 2004) , encore appelé λ-*lifting* : les prédicats utilisés comme valeurs de première classe vont être codés sous

la forme d'objets de la logique. Un prédicat noté ε permet leur interprétation.

Notre contribution est d'adapter cette traduction classique à un monde polymorphiquement typé et de faire quelques ajustements pour que le résultat du codage soit adapté aux prouveurs automatiques. Pour mener à bien ce codage, nous avons donc besoin de plusieurs constantes et constructeurs de types pour dénoter les prédicats comme des objets :

- le type o correspond au type des dénotations des termes au niveau objet ;
- le constructeur de type ofun d'arité 2 est le type des objets dénotant des fonctions (logiques ou des prédicats) ;
- le symbole de fonction apply de type $\forall\alpha\beta.(\mathsf{ofun}\ (\alpha,\beta),\alpha) \to \beta$ correspond à l'application d'une fonction au niveau objet.
- le symbole de prédicat ε de type $\mathsf{o} \to \mathsf{prop}$ correspond à l'interprétation de la dénotation objet.
- les constantes correspondant aux connecteurs logiques.

Comme indiqué dans (Dowek *et al*, 2001), il est nécessaire de rajouter des axiomes d'extensionnalité pour que ce codage soit correct et complet.

Codage de l'égalité

Même si le codage est correct et complet, les prouveurs automatiques du premier ordre ont des stratégies de résolution peu enclines à utiliser les théorèmes d'interprétation. Les méthodes utilisées cherchent à contrôler le nombre d'instanciations des lemmes, passer par une couche d'interprétation n'est donc pas très efficace en général.

Beaucoup de prouveurs automatiques utilisent l'algorithme *congruence closure* pour maintenir efficacement les classes d'équivalence entre termes (et factoriser ainsi le travail d'instanciation). Coder les égalités handicape donc le prouveur puisqu'il doit passer par des lemmes d'interprétation pour les obtenir.

Quand c'est possible, on essaie donc de traduire les égalités entre des objets d'ordre supérieur sans utiliser le passage par les combinateurs d'interprétation. En effet, lorsque l'on a par exemple deux fonctions logiques égalisées ainsi :

$$\lambda x.f(x) = \lambda y.g(y)$$

Il suffit de renommer la variable y en x pour réécrire cette égalité en

$$\forall x.f(x) = g(x)$$

Le même raisonnement s'applique aux prédicats mais on utilise alors le connecteur d'équivalence logique plutôt que l'égalité.

Codage des types inductifs

Nous utilisons le même codage que celui proposé par Jean-Christophe Filliâtre et Nicolas Ayache (Ayache & Filliâtre, 2006). Il s'agit de traduire les lemmes d'injection, d'inversion et d'algèbre libre pour chaque déclaration de types algébriques.

Codage des prédicats inductifs

Chaque cas des prédicats inductifs est traduit au premier ordre par la procédure indiquée plus haut. Le prouveur automatique peut ainsi trouver le cas du prédicat inductif à employer. Nous ne traduisons pas les schémas d'induction car comme ils sont du second ordre, il y a peu de chance qu'ils soient utilisés.

Par contre, on détecte les prédicats inductifs dirigés par la syntaxe d'un type algébrique (typiquement lorsqu'on trouve un cas d'inductif pour chaque constructeur de données). C'est un cas particulièrement commun et on génère un lemme d'inversion qui s'avère très utile en pratique.

Codage des fonctions logiques

Les fonctions logiques sont des fonctions récursives discriminant sur un argument d'un type algébrique. C'est le cas typique mis en lumière par le codage de Nicolas Ayache et Jean-Christophe Filliâtre (Ayache & Filliâtre, 2006). Nous utilisons la même méthode : on produit une égalité pour chaque cas de la fonction logique.

11.5.4 Codage de PFOL vers FOL

Nous avons mentionné l'utilisation du prouveur automatique Ergo (Conchon & Contejean, 2006) car il travaille directement dans PFOL. Nous pensons que c'est une solution plus prometteuse que les logiques non typées car le typage peut diriger les instanciations à effectuer et rendre la recherche de preuve plus efficace. Ergo (Conchon & Contejean, 2006) est un système très récent et, à ce jour, il lui manque quelques propriétés essentielles comme l'intégration de symboles commutatifs et associatifs dans le raisonnement. Il a donc été nécessaire, dans l'implémentation, d'encoder la logique polymorphiquement typée du premier ordre vers une logique du premier ordre non typée pour pouvoir utiliser un plus grand nombre de prouveurs comme par exemple Simplify (Detlefs *et al*, 2005).

Si on oublie tout simplement les types, le codage est incorrect. En effet, la propriété

$$\forall(x, y : unit).x = y$$

est valide dans PFOL. Par contre,

$$\forall x, y.x = y$$

est évidemment invalide dans FOL. Ce problème a été traité de manière approfondie par Stéphane Lescuyer (Lescuyer, 2006, Couchot & Lescuyer, 2007). La solution consiste à encoder les types à l'aide de termes du premier ordre. On utilise ensuite un prédicat « sort » d'arité 2 qui prend un terme et une représentation de type pour contraindre la forme des termes à avoir le bon type. On encode ainsi la vérification de typage à l'intérieur de la logique en embarquant les types à l'intérieur des termes. L'énoncé problématique se réécrit alors

$$\forall x, y.sort(x, unit) = sort(y, unit)$$

L'instanciation de ce théorème est alors limitée aux termes de la forme $sort(t, unit)$ et le codage s'assure que ce sont seulement les termes qui ont le type $unit$ dans le langage source.

CHAPITRE DOUZE

Le langage calculatoire

Ce chapitre formalise le langage de programmation du système. Il s'agit d'un langage fonctionnel pur polymorphiquement typé, muni de types algébriques et dans lequel le programmeur peut insérer des assertions logiques.

12.1 Syntaxe

La syntaxe de notre langage de programmation apparaît dans la figure 12.1. Il est muni d'un système de type à la ML (Milner, 1978) ce qui implique une distinction entre les types τ et les schémas de type σ. Comptent, parmi les types, les variables de type, les types algébriques paramétrés et les types fonctionnels. Nous écrivons $\dot{\longrightarrow}$ pour le constructeur de type des fonctions calculatoires de façon à le distinguer du constructeur de type des fonctions logiques $\rightarrow$ (figure 11.1).

Nous imposons une séparation syntaxique entre valeurs et expressions et nous demandons que les deux sous-termes des applications ainsi que les termes analysés par le *pattern matching* soient des valeurs. Ce style rappelle les formes A-normales (Flanagan *et al*, 1993) où le résultat de tout calcul intermédiaire doit être nommé par la construction let. Bien sûr, un tel style n'est pas confortable pour le programmeur et, en pratique, le langage de surface permet les constructions habituelles qui sont traduites automatiquement vers le langage présenté ici.

Le langage admet une inférence de type dans le style de Hindley-Milner. Cependant, dans cette présentation, nous ne sommes pas intéressés par l'inférence de type donc nous travaillons sur des programmes explicitement typés comme en témoignent

(i) la syntaxe des valeurs et des motifs où les variables et les constructeurs de données sont annotés par des vecteurs de types qui indiquent comment les schémas de type polymorphes doivent être instanciés,

(ii) les constructions fun et let dont les variables liées sont annotés par des types, et

(iii) la construction let où les variables de type $\overline{\alpha}$ sont explicitement liées.

Une définition de fonction suit la forme générale :

$$\mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t$$

Le mot-clé / doit être lu « où ». Toute fonction est récursive : f est liée à l'intérieur de t. Le paramètre formel x_1 est lié dans la précondition F_1, dans la postcondition F_2 et dans t. La variable x_2 est le résultat de la fonction et est uniquement liée dans la postcondition F_2. Toute fonction doit être annotée par une précondition et une postcondition (si l'une d'elles manque, on suppose qu'elle vaut true).

Une variable locale est définie comme suit :

$$\mathsf{let}\,(x\,\overline{\alpha} : \tau/F) = t_1\ \mathsf{in}\ t_2$$

			Types calculatoires
τ	$::=$	α	*Variable*
	$\mid$	$\varepsilon\ \overline{\tau}$	*Données*
	$\mid$	$\tau \dot{\longrightarrow} \tau$	*Fonctions*
σ	$::=$	$\forall \overline{\alpha}.\, \tau$	*Schema*
			Valeurs
v	$::=$	$x\,\overline{\tau}$	*Variable*
	$\mid$	$D\,\overline{\tau}\ (v, \ldots, v)$	*Données*
	$\mid$	$\mathsf{fun}\, f(x : \tau/F) : (x : \tau/F) = t$	*Fonction récursive*
			Motifs
p	$::=$	$x\,\overline{\tau}$	*Variable*
	$\mid$	$D\,\overline{\tau}\ (p, \ldots, p)$	*Données*
			Expressions
t	$::=$	v	*Valeur*
	$\mid$	$v\ (v)$	*Application de fonction*
	$\mid$	$\mathsf{let}\,(x\,\overline{\alpha} : \tau/F) = t\ \mathsf{in}\ t$	*Liaison locale*
	$\mid$	$\mathsf{match}\, v\ \mathsf{with}\ c$	*Analyse par cas*
			Cas
c	$::=$	$\emptyset$	*Nil*
	$\mid$	$p.t \,[]\, c$	*Cons*

FIG. 12.1: Le langage calculatoire (syntaxe).

La variable locale x est liée dans F et dans t_2. Les variables de type α sont liées dans τ, F et dans t_1. La proposition F sert de postcondition à t_1. Si la proposition est manquante alors on suppose qu'elle vaut une postcondition par défaut dont la définition sera donnée dans la section 12.3.

Une analyse par cas s'écrit :

$$\mathsf{match}\, v\ \mathsf{with}\ c$$

Ici, c est une séquence éventuellement vide de branches. Chaque branche est de la forme $p.t$ où les variables qui apparaissent dans le motif p sont liées dans t. Les motifs doivent être linéaires.

12.2 Reflet des entités calculatoires dans le monde logique

Dans la logique de Hoare, les formules font références aux valeurs. Cela signifie que, si x est liée au niveau calculatoire par un des opérateurs fun, let ou match, alors il est possible qu'une formule F présente dans la portée lexicale de x fasse référence à la variable x. Deux questions sont alors inévitables : d'abord, si la variable x a un type calculatoire τ, quel est son type logique à utiliser pour vérifier le bon typage de la formule F ? Ensuite, si au cours de l'évaluation, la variable x est substituée par la valeur v, alors quelle est la valeur logique correspondante à utiliser pour interpréter la formule F ?

Nous répondons à ces questions en injectant les types calculatoires et les valeurs calculatoires dans le monde logique (figure 12.2). Plus précisément, chaque type calculatoire τ est injecté *via* le type logique $\lceil\tau\rceil$ et chaque valeur calculatoire v est associée au terme logique $\lceil v\rceil$ avec l'idée que si v a le type τ alors $\lceil v\rceil$ a le type $\lceil\tau\rceil$.

$$
\begin{array}{rcl}
\textbf{\textit{Types}} & & \\
\lceil \alpha \rceil & = & \alpha \\
\lceil \varepsilon\ \overline{\tau} \rceil & = & \varepsilon\ \lceil \overline{\tau} \rceil \\
\lceil \tau_1 \dot{\longrightarrow} \tau_2 \rceil & = & (\lceil \tau_1 \rceil \to \mathsf{prop}) \times (\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil \to \mathsf{prop}) \\
 & & \\
\textbf{\textit{Schéma de types}} & & \\
\lceil \forall \overline{\alpha}.\, \tau \rceil & = & \forall \overline{\alpha}.\, \lceil \tau \rceil \\
 & & \\
\textbf{\textit{Valeurs}} & & \\
\lceil x\, \overline{\tau} \rceil & = & x\, \lceil \overline{\tau} \rceil \\
\lceil D\, \overline{\tau}\ (v_1, \ldots, v_n) \rceil & = & D\, \lceil \overline{\tau} \rceil\ (\lceil v_1 \rceil, \ldots, \lceil v_n \rceil) \\
\lceil \mathsf{fun}\, f(x_1 : \tau_1 / F_1) : (x_2 : \tau_2 / F_2) = t \rceil & = & (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2)
\end{array}
$$

FIG. 12.2: Reflet des types calculatoires et des valeurs dans le monde logique.

Comme annoncé (dans le chapitre 10), les fonctions calculatoires sont reflétées, au niveau logique, par une paire formée d'une précondition et d'une postcondition. On le constate en observant l'injection logique des types des fonctions calculatoires :

$$\lceil \tau_1 \dot{\longrightarrow} \tau_2 \rceil = (\lceil \tau_1 \rceil \to \mathsf{prop}) \times (\lceil \tau_1 \rceil \to \lceil \tau_2 \rceil \to \mathsf{prop})$$

La première composante de la paire, qui représente la précondition de la fonction, est abstraite par rapport à l'argument de la fonction alors que le second argument, qui représente la postcondition, est abstraite par rapport à l'argument et au résultat.

Il suit de cette définition que, si f est liée, au niveau calculatoire, à une fonction du type $\tau_1 \dot{\longrightarrow} \tau_2$, alors une formule dans la portée lexicale de f la verra sous la forme d'une paire de prédicats. (Souvenez-vous que pre et post sont des sucres syntaxiques pour les projections π_1 et π_2, comme mentionné dans la section 11.1.)

Les valeurs fonctionnelles calculatoires (les λ-abstractions) sont injectées dans le monde logique d'une façon cohérente avec cette définition. La précondition et la postcondition d'une fonction déterminent seules son reflet logique : le code de la fonction est ignoré. La motivation principale de ce choix de conception, c'est que le raisonnement sur le comportement d'une fonction doit être fait *via* sa spécification uniquement. Avec notre approche, il n'est pas possible de raisonner *a posteriori* sur le code d'une fonction.

Pour les types algébriques, nous reflétons chaque définition par un type inductif isomorphe. Ainsi, pour chaque constructeur de type algébrique ε calculatoire, il doit exister un constructeur de type inductif logique de même arité et qu'on écrira aussi ε. Pour chaque constructeur de données calculatoire :

$$D : \forall \overline{\alpha}.\, \tau_1 \times \ldots \times \tau_n \to \varepsilon\ \overline{\alpha},$$

il existe un constructeur de données logique :

$$D : \forall \overline{\alpha}.\, \lceil \tau_1 \rceil \times \ldots \times \lceil \tau_n \rceil \to \varepsilon\ \overline{\alpha}.$$

Pour que la condition de positivité (voir le chapitre 11) soit remplie, il faut que le constructeur de type ε (ou un constructeur de type dont la définition est mutuellement récursive avec celle de ε) n'apparaisse pas à gauche ou à droite d'une flèche à l'intérieur des types $\tau_1, \ldots, \tau_n$. Cette contrainte supplémentaire est due à la définition du reflet d'une valeur fonctionnelle. En effet, celui-ci transporte le type de retour de la fonction à gauche de la flèche et multiplie ainsi les occurences strictement positives potentiellement dangereuses du constructeur de type ε.

12.3 Inférer les postconditions les plus fortes

Pour simplifier la définition de la procédure d'extraction de preuve, nous imposons que chaque construction let soit annotée par une postcondition. En pratique, cependant, annoter systématiquement chaque let n'est pas souhaitable car dans certains cas, on peut inférer une postcondition raisonnable.

Idéalement, la formule que nous devons inférer dans cette situation est la postcondition la plus forte pour le membre gauche du let. Notre logique est suffisamment puissante pour exprimer la postcondition la plus forte de n'importe quelle construction du langage. Par exemple, la postcondition pour la valeur v est $\lambda x.(x = \lceil v \rceil)$. La postcondition la plus forte pour une application de fonction $v_1\ (v_2)$ est $\mathsf{post}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil)$. Nous pourrions aller plus loin et expliquer comment la postcondition la plus forte peut être inférée pour les constructions let et match. Mais, dans ces deux cas, les formules deviendraient complexes en faisant intervenir des quantifications existentielles et des disjonctions.

Au final, les postconditions portées par les constructions let interviennent dans les obligations de preuve, où elles apparaissent sous la forme d'hypothèses. Pour cette raison, nous ne voulons pas qu'elles soient complexes : nous voulons produire des obligations de preuve simples et compréhensibles.

Notre réponse à ce problème est de construire la postcondition la plus forte pour les valeurs et les applications de fonction, comme suggéré plus haut, mais de ne pas le faire pour les constructions let et match : à la place d'une postcondition manquante sur ces constructions, nous utilisons la postcondition true.

Il est possible, *via* la conversion en une forme A-normale, de s'assurer que le membre gauche d'une construction let n'est jamais une autre construction let. Dès lors, le seul cas où notre approche naïve demande une annotation explicite est celui d'un let dont le membre gauche est un match. Le caractère obligatoire de cette annotation tue dans l'œuf toute explosion combinatoire de la postcondition qui pourrait survenir par l'interaction entre les séquences et les alternatives (Flanagan & Saxe, 2001).

12.4 Notions de substitution

La sémantique opérationnelle que nous allons expliciter un peu plus loin réécrit des expressions qui contiennent à la fois des types et des formules explicites. Pour s'assurer que ces annotations s'accordent au cours de la transformation des expressions, nous devons définir quelques définitions légèrement non standard des substitutions.

Une même variable de type α peut apparaître à la fois dans les types logiques et dans les types calculatoires. De même, une variable x donnée peut intervenir dans une formule et dans une expression. Pour cette raison, nous écrivons $[\alpha \mapsto \tau]$ pour la substitution qui remplace toutes les occurrences libres de α au niveau calculatoire dans τ et toutes les occurrences libres de α au niveau logique dans $\lceil \tau \rceil$. Dans le même esprit, nous écrirons $[x \mapsto v]$ pour la substitution qui remplace les occurrences libres de x au niveau calculatoire par v et les occurrences libres de x au niveau logique par $\lceil v \rceil$.

Nous avons annoté les constructions let avec une abstraction de type explicite et les occurrences de variables avec des applications de type explicites. Réduire un let-redex suppose donc simultanément une réduction au niveau des types des β-redexes. Pour cela, nous écrivons $[x \mapsto \Lambda\overline{\alpha}.v]$ pour la substitution qui remplace chaque occurrence de variable de la forme $x\,\overline{\tau}$ par $[\overline{\alpha} \mapsto \overline{\tau}]v$. Ce remplacement est effectué à la fois dans le monde calculatoire et dans le monde logique *modulo* une injection dans le cas logique.

On étend l'application des substitutions aux environnements calculatoires en les appliquant seulement aux formules. Ainsi :

$$\begin{array}{rcl} \rho(\emptyset) & = & \emptyset \\ \rho(\Gamma, (x : \sigma)) & = & \rho(\Gamma), (x : \sigma) \\ \rho(\Gamma, \overline{\alpha}) & = & \rho(\Gamma), \overline{\alpha} \\ \rho(\Gamma, F) & = & \rho(\Gamma), \rho(F) \end{array}$$

$$
\begin{array}{rcll}
\text{(E-Beta)} & & & \\
v_1\ (v_2) & \to & [x \mapsto v_2][f \mapsto v_1]t & \text{si } v_1 \text{ est } \mathsf{fun}\, f(x : \tau/F) : (\ldots) = t \\
\\
\text{(E-Let)} & & & \\
\mathsf{let}\,(x\,\overline{\alpha} : \tau/F) = v\ \mathsf{in}\ t & \to & [x \mapsto \Lambda\overline{\alpha}.v]t & \\
\\
\text{(E-Case-Cons-Match)} & & & \\
\mathsf{match}\ v\ \mathsf{with}\ p.t \,[]\, c & \to & [p \mapsto v]t & \text{si } [p \mapsto v] \text{ est définie} \\
\\
\text{(E-Case-Cons-Fail)} & & & \\
\mathsf{match}\ v\ \mathsf{with}\ p.t \,[]\, c & \to & \mathsf{match}\ v\ \mathsf{with}\ c & \text{sinon} \\
\\
\text{(E-Let-Context)} & & & \\
\mathsf{let}\,(x\,\overline{\alpha} : \tau/F) = t_1\ \mathsf{in}\ t_2 & \to & \mathsf{let}\,(x\,\overline{\alpha} : \tau/F) = t_1'\ \mathsf{in}\ t_2 & \text{si } t_1 \to t_1'
\end{array}
$$

FIG. 12.3: Sémantique opérationnelle.

Pour finir, la notation $[x \mapsto v]$, qui dénote une substitution d'une valeur pour une variable, est étendue à la notation $[p \mapsto v]$ qui est indéfinie quand le motif p ne filtre pas la valeur v mais qui dénote une substitution simultanée de valeurs pour les variables de p dans le cas contraire. Cette substitution simultanée est définie formellement comme suit :

$$[D\,\overline{\tau}\ (p_1, \ldots, p_n) \mapsto D\,\overline{\tau}\ (v_1, \ldots, v_n)]$$

correspond à

$$[p_1 \mapsto v_1] \cup \ldots \cup [p_n \mapsto v_n]$$

Comme les motifs sont linéaires, c'est une union de substitutions dont les domaines sont disjoints deux à deux.

12.5 Sémantique opérationnelle

Le langage de programmation est muni d'une sémantique opérationnelle à petits pas et en appel par valeur qui est spécifiée par la figure 12.3. Elle est parfaitement standard. On y trouve trois types de *redexe*s (β, let et match) et un contexte d'évaluation (le membre gauche des lets).

Il est trivial de vérifier qu'une expression irréductible, éventuellement nichée au sein de contextes d'évaluation, peut soit être une valeur, soit une expression de la forme $\mathsf{match}\ v\ \mathsf{with}\ \emptyset$ ou une application $v_1\ (v_2)$ où v_1 n'est pas une fonction. Dans les deux derniers cas, on dit que l'expression est bloquée.

Adopter une stratégie d'évaluation par nom poserait un problème que nous décrivons dans le chapitre 16.

CHAPITRE TREIZE

Le système de type et de preuve

Nous munissons maintenant le langage calculatoire d'un système de type dans le style de ML et d'un système de preuve (une logique de Hoare), qui peut être vu comme un algorithme pour extraire des obligations de preuve des programmes bien typés. Pour être plus synthétique, nous décrivons ces deux systèmes à l'aide d'un unique jugement qui énonce qu'un programme est bien typé et est annoté par une formule cohérente.

En pratique, notre système vérifie d'abord si le programme est bien typé et infère en même temps les annotations de type manquantes. Ensuite, on extrait un ensemble d'obligations de preuve exprimées dans notre logique d'ordre supérieur typée. Le fait que le programme (y compris les assertions logiques) est bien typé garantit que les obligations de preuve le sont aussi.

13.1 Environnement de typage

La syntaxe des environnements de type Γ apparaît dans la figure 13.1. Comme d'habitude, les environnements de type lient des variables et des variables de types. Les environnements contiennent aussi des formules qui deviennent des hypothèses quand les obligations de preuve sont émises. Un environnement de la forme Γ, F est bien formé quand F a le type prop sous $\lceil\Gamma\rceil$. Enfin, dans la figure 13.2, on étend enfin le reflet logique des types calculatoires aux environnements en remplaçant les types des variables par leurs reflets logiques.

13.2 Obligations de preuve

Une obligation de preuve est un jugement de la forme $\Gamma \models F$ où F a le type prop sous $\lceil\Gamma\rceil$. Les règles de la figure 13.3 traduisent une obligation de preuve en une formule close de type prop qui doit être valide pour que le jugement le soit. La validité d'une formule est décidée par un prouveur externe.

13.3 Jugements

Le système de preuve est défini par trois jugements qui énoncent les propriétés des valeurs, des motifs et des expressions :

Valeurs	$\Gamma \vdash v : \tau$	(figure 13.5)
Motifs	$\Gamma \vdash p : \tau$	(figure 13.6)
Expressions	$\Gamma \vdash t : \tau\ \{P\}$	(figure 13.7)

$$
\begin{array}{rcll}
\Gamma & ::= & \emptyset & \textit{Nil} \\
 & | & \Gamma, (x : \sigma) & \textit{Variable} \\
 & | & \Gamma, \overline{\alpha} & \textit{Variables de type} \\
 & | & \Gamma, F & \textit{Hypothèse}
\end{array}
$$

FIG. 13.1: Syntaxe des environnements de typage.

$$
\begin{array}{rcl}
\lceil \emptyset \rceil & = & \emptyset \\
\lceil \Gamma, (x : \sigma) \rceil & = & \lceil \Gamma \rceil, (x : \lceil \sigma \rceil) \\
\lceil \Gamma, \overline{\alpha} \rceil & = & \lceil \Gamma \rceil, \overline{\alpha} \\
\lceil \Gamma, F \rceil & = & \lceil \Gamma \rceil
\end{array}
$$

FIG. 13.2: Reflet logique des environnements de typage.

13.4 Valeurs

Le jugement $\Gamma \vdash v : \tau$ (figure 13.5) dénote le fait que sous l'environnement de typage Γ, la valeur v a le type τ. Aucune précondition ou postcondition n'apparaît dans le jugement. En effet, comme les valeurs ne demandent aucun calcul, elles n'ont pas besoin de précondition. De plus, comme toutes les valeurs peuvent être injectées dans le monde logique, elles n'ont pas besoin d'une postcondition explicite : la postcondition la plus forte pour une valeur v est la simple égalité à $\lceil v \rceil$.

Les règles VAR et DATA sont immédiates. La règle FUN est plus complexe. Deux prémisses forcent la précondition F_1 et la postcondition F_2 a être des formules bien formées sous les environnements appropriés. La dernière prémisse vérifie que le corps de la fonction est conforme à la spécification de la fonction. À cet effet, l'environnement de typage est étendu par la liaison de f et x_1 et aussi par l'hypothèse :

$$f = \lceil \mathsf{fun}\, f \ldots \rceil,$$

qui par définition de l'injection (12.2) est un synonyme de :

$$f = (\lambda(x_1 : \lceil \tau_1 \rceil).F_1, \lambda(x_1 : \lceil \tau_1 \rceil).\lambda(x_2 : \lceil \tau_2 \rceil).F_2).$$

Cette hypothèse donne du sens aux occurrences de $\mathsf{pre}(f)$ et $\mathsf{post}(f)$ à l'intérieur du corps de la fonction et permettent la vérification des appels récursifs à f. Enfin, l'environnement est aussi étendu avec la précondition F_1 qui signifie qu'à l'intérieur du corps de la fonction, la précondition est supposée vraie. Sous cet environnement étendu, la corps de la fonction doit produire une valeur qui vérifie la postcondition $\lambda(x_2 : \lceil \tau_2 \rceil).F_2$.

Il n'est pas difficile de voir que $\Gamma \vdash v : \tau$ implique $\lceil \Gamma \rceil \vdash \lceil v \rceil : \lceil \tau \rceil$. Cette propriété est requise pour que les règles de types ne construisent que des formules bien formées.

13.5 Motifs

Le jugement $\Gamma \vdash p : \tau$ (13.6) affirme qu'une valeur de type τ peut être filtrée par le motif p en introduisant les liaisons décrites par Γ. Ces liaisons sont monomorphes (voir PAT-VAR). Comme les motifs sont linéaires, les environnements de type $\Gamma_1, \ldots, \Gamma_n$ dans la règle PAT-DATA ont des domaines disjoints.

$$\frac{F \text{ valide}}{\emptyset \models F} \qquad \frac{\Gamma \models \forall(x : \lceil\sigma\rceil).F}{\Gamma, (x : \sigma) \models F} \qquad \frac{\Gamma \models \forall\overline{\alpha}.F}{\Gamma, \overline{\alpha} \models F} \qquad \frac{\Gamma \models F_1 \Rightarrow F_2}{\Gamma, F_1 \models F_2}$$

FIG. 13.3: Émission des obligations de preuve.

$$\begin{array}{rcll} (\Gamma, (x : \sigma))(x) & = & \sigma & \\ (\Gamma, (x_1 : \sigma))(x_2) & = & \Gamma(x_2) & \text{si } x_1 \# x_2 \\ (\Gamma, \overline{\alpha})(x) & = & \Gamma(x) & \text{si } \overline{\alpha} \# \Gamma(x) \\ (\Gamma, F)(x) & = & \Gamma(x) & \end{array}$$

FIG. 13.4: Accès à l'environnement calculatoire.

13.6 Expressions

Le jugement $\Gamma \vdash t : \tau \ \{P\}$ (13.7) énonce que, sous l'environnement de typage Γ, l'expression t a le type τ et que (si son évaluation termine alors) elle produit une valeur dont le reflet logique satisfait le prédicat P. Dans un tel jugement, P a le type $\lceil\tau\rceil \to \mathsf{prop}$ sous $\lceil\Gamma\rceil$.

La règle VALUE reflète directement la signification voulue : le jugement $\Gamma \vdash v : \tau \ \{P\}$ est valide si et seulement si v a le type τ sous Γ et on peut prouver que son reflet logique $\lceil v\rceil$ satisfait P sous les hypothèses trouvées dans Γ. La prémisse $\Gamma \models P\ (\lceil v\rceil)$ est une obligation de preuve.

La règle APP demande à la fonction v_1 et à son argument réel v_2 d'avoir des types calculatoires concordants. De plus, elle émet deux obligations de preuve affirmant que

(i) l'argument effectif doit vérifier la précondition de la fonction et

(ii) la postcondition de la fonction doit impliquer la postcondition P désirée.

Dans la dernière prémisse, nous écrivons $P' \Rightarrow P$ où P' et P ont les types $\lceil\tau_2\rceil \to \mathsf{prop}$ pour $\forall(x : \lceil\tau_2\rceil).(P'\ (x) \Rightarrow P\ (x))$, avec x choisie fraîche pour P' et P.

La règle LET vérifie que t_1 a le type τ_1 et est compatible avec la postcondition F fournie par l'utilisateur. Ensuite, elle effectue une généralisation de type, dans le style de Milner (1978), de manière à ce que t_2 soit vérifiée par rapport à $(x : \forall\overline{\alpha}.\,\tau_1)$. L'hypothèse est changée en $\forall\overline{\alpha}.[x \mapsto x\,\overline{\alpha}]F$ pour prendre en compte le fait que x est désormais polymorphe.

La règle CASE-NIL émet l'obligation de preuve $\Gamma \models \mathsf{false}$ qui impose l'ensemble des hypothèses trouvées dans Γ à être inconsistant. Ceci est suffisant pour s'assurer qu'une analyse par cas avec aucune branche n'est jamais exécutée.

La règle CASE-CONS demande à la valeur v et au motif p d'avoir le même type τ. L'environnement Γ' collecte les variables liées dans p avec leurs types. Sous l'hypothèse qu'une certaine instance de p concorde avec v, ce qui est exprimé par l'extension de Γ en Γ' avec l'hypothèse supplémentaire que $\lceil v\rceil = \lceil p\rceil$, on doit vérifier que le corps de la branche t a le type désiré τ' et vérifie la postcondition P. De plus, sous l'hypothèse qu'aucune instance de p ne coïncide avec v, ce qui est écrit $\forall\Gamma'.\lceil v\rceil \neq \lceil p\rceil$, le reste des branches doit aussi avoir le type τ' et vérifier la postcondition P.

Quand on vérifie une construction match avec n branches, la $(k+1)$-ième branche est vérifiée sous l'hypothèse qu'aucun des motifs $p_1, \ldots, p_k$ ne capture la forme de la valeur v. En particulier, pour $k = n$, la conjonction de toutes ces hypothèses de la forme $(\forall\Gamma'_i.\lceil v\rceil \neq \lceil p_i\rceil)$ doit être incohérente ce qui assure que le contrôle ne peut pas parvenir à la fin d'une analyse par cas ou, dit autrement, que l'analyse par cas est exhaustive. Aujourd'hui les compilateurs ML et Haskell implémentent une approximation conservative de ce test en utilisant un critère purement syntaxique. Nous implémentons aussi ce critère

$$\textsc{Var}\ \frac{\Gamma(x) = \forall\overline{\alpha}.\,\tau}{\Gamma \vdash x\,\overline{\tau} : [\overline{\alpha} \mapsto \overline{\tau}]\tau} \qquad \textsc{Data}\ \frac{D : \forall\overline{\alpha}.\,\tau_1 \times \ldots \times \tau_n \to \varepsilon\ \overline{\alpha} \quad \forall i \quad \Gamma \vdash v_i : [\overline{\alpha} \mapsto \overline{\tau}]\tau_i}{\Gamma \vdash D\,\overline{\tau}\ (v_1, \ldots, v_n) : \varepsilon\ \overline{\tau}}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} f \,\#\, F_1, F_2 \\ \lceil \Gamma, (x_1 : \tau_1) \rceil \vdash F_1 : \mathsf{prop} \qquad \lceil \Gamma, (x_1 : \tau_1), (x_2 : \tau_2) \rceil \vdash F_2 : \mathsf{prop} \\ \Gamma, (f : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2), f = \lceil \mathsf{fun}\, f \ldots \rceil, (x_1 : \tau_1), F_1 \qquad \vdash t : \tau_2\ \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\} \end{array}}{\Gamma \vdash \mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t : \tau_1 \stackrel{\cdot}{\longrightarrow} \tau_2}$$

FIG. 13.5: Le langage calculatoire (fragment du système de preuve concernant les valeurs).

$$\textsc{Pat-Var}\ (x : \tau) \vdash x : \tau \qquad \textsc{Pat-Data}\ \frac{D : \forall\overline{\alpha}.\,\tau_1 \times \ldots \times \tau_n \to \varepsilon\ \overline{\alpha} \quad \forall i \quad \Gamma_i \vdash p_i : [\overline{\alpha} \mapsto \overline{\tau}]\tau_i}{\Gamma_1, \ldots, \Gamma_n \vdash D\,\overline{\tau}\ (p_1, \ldots, p_n) : \varepsilon\ \overline{\tau}}$$

FIG. 13.6: Le langage calculatoire (fragment du système de preuve concernant les motifs).

syntaxique : quand il réussit, il n'est pas nécessaire d'engendrer une obligation de preuve.

13.7 Lecture algorithmique

Le jugement $\Gamma \vdash t : \tau\ \{P\}$ définit un algorithme pour la génération des obligations de preuve. Les quatres paramètres du jugement, Γ, t, τ et P sont des entrées de l'algorithme qui s'occupe uniquement de construire une dérivation du jugement en opérant un parcours en profondeur des termes. Tout au long de la descente dans t, l'algorithme augmente l'environnement Γ lorsqu'il rencontre les constructions fun, let et match. Simultanément, la postcondition P est propagée vers le bas par ce même processus très simple. Sur les constructions let, la propagation utilise l'annotation de l'utilisateur de manière à déterminer quelle postcondition doit être poussée dans le membre gauche. La sortie de l'algorithme consiste en des obligations de preuve, de la forme $\Gamma \models F$, attachées aux feuilles de la dérivation (voir les règles VALUE, APP, and CASE-NIL).

En partageant les préfixes de l'environnement et les sous-formules, il est possible de construire une représentation en mémoire de l'ensemble des obligations de preuve qui est linéaire en la taille du programme. Néanmoins, comme chaque obligation de preuve est présentée indépendamment des autres au prouveur automatique, cette information partagée est perdue ce qui dégrade la taille d'ensemble des obligations de preuve à une taille, au pire, quadratique en la taille du programme.

13.8 Correction

La correction de notre système de type et de système de preuve est établie d'une manière standard par la voie syntaxique (Wright & Felleisen, 1994).

On peut oublier une hypothèse lorsqu'elle est redondante :

Lemme 13.8.1 (Affaiblissement de l'environnement)
$\Gamma_1, F, \Gamma_2 \vdash t : \tau\ \{P\}$ *et* $\Gamma_1 \models F$ *implique* $\Gamma_1, \Gamma_2 \vdash t : \tau\ \{P\}$. ◇

$$\textsc{Value}\ \frac{\Gamma \vdash v : \tau \qquad \Gamma \models P\ (\lceil v \rceil)}{\Gamma \vdash v : \tau\ \{P\}} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : \tau_1 \dot{\longrightarrow} \tau_2 \qquad \Gamma \vdash v_2 : \tau_1 \qquad \Gamma \models \mathsf{pre}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil) \qquad \Gamma \models \mathsf{post}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil) \Rightarrow P}{\Gamma \vdash v_1\ (v_2) : \tau_2\ \{P\}}$$

$$\textsc{Let}\ \frac{x \# P \qquad \lceil \Gamma, \overline{\alpha}, (x : \tau_1) \rceil \vdash F : \mathsf{prop} \qquad \Gamma, \overline{\alpha} \vdash t_1 : \tau_1\ \{\lambda(x : \lceil \tau_1 \rceil).F\} \qquad \Gamma, (x : \forall \overline{\alpha}.\, \tau_1), \forall \overline{\alpha}.[x \mapsto x\,\overline{\alpha}]F \vdash t_2 : \tau_2\ \{P\}}{\Gamma \vdash \mathsf{let}\,(x\,\overline{\alpha} : \tau_1 / F) = t_1\ \mathsf{in}\ t_2 : \tau_2\ \{P\}}$$

$$\textsc{Case-Nil}\ \frac{\Gamma \vdash v : \tau \qquad \Gamma \models \mathsf{false}}{\Gamma \vdash \mathsf{match}\ v\ \mathsf{with}\ \emptyset : \tau'\ \{P\}} \qquad \textsc{Case-Cons}\ \frac{\Gamma \vdash v : \tau \qquad \Gamma' \vdash p : \tau \qquad p \# v, P \qquad \Gamma, \Gamma', \lceil v \rceil = \lceil p \rceil \vdash t : \tau'\ \{P\} \qquad \Gamma, (\forall \Gamma'. \lceil v \rceil \neq \lceil p \rceil) \vdash \mathsf{match}\ v\ \mathsf{with}\ c : \tau'\ \{P\}}{\Gamma \vdash \mathsf{match}\ v\ \mathsf{with}\ p.t\ [\!]\ c : \tau'\ \{P\}}$$

FIG. 13.7: Le langage calculatoire (fragment du système de preuve concernant les expressions).

Preuve. Par induction sur la dérivation de typage, chaque feuille de la forme $\Gamma_1, F \models F'$ peut être remplacée par $\Gamma_1 \models F'$. □

De même, une postcondition peut être affaiblie puisqu'elle apparaît toujours à droite d'une implication d'une formule à prouver ou elle est elle-même cette formule :

Lemme 13.8.2 (Affaiblissement de la postcondition)
$\Gamma \vdash t : \tau\ \{P_1\}$ *et* $\Gamma \models P_1 \Rightarrow P_2$ *implique* $\Gamma \vdash t : \tau\ \{P_2\}$. ◇

Preuve. Par induction sur la dérivation de typage. Les cas des règles LET, CASE-NIL et CASE-CONS sont des applications immédiates de l'hypothèse d'induction. Le cas de la règle VALUE est clair compte tenu que l'hypothèse $\Gamma \models P_1\ (\lceil v \rceil)$ vient par inversion du jugement de typage et que $\Gamma \models P_1 \Rightarrow P_2$. Le cas de la règle APP est aussi immédiat puisque par inversion, on a $\Gamma \models \mathsf{post}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil) \Rightarrow P_1$ qui implique clairement $\Gamma \models \mathsf{post}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil) \Rightarrow P_2$ car $\Gamma \models P_1 \Rightarrow P_2$. □

Les lemmes de substitution habituels sont nécessaires à la preuve de préservation du typage par l'évaluation.

Lemme 13.8.3 (Substitution de type dans les valeurs)
Soit ϕ *valant* $[\overline{\alpha} \mapsto \overline{\tau}]$. *Alors,* $\Gamma_1, \overline{\alpha}, \Gamma_2 \vdash v : \tau$ *et* $\overline{\alpha} \# \mathcal{D}(\Gamma_2)$ *implique* $\Gamma_1, \phi(\Gamma_2) \vdash \phi(v) : \phi(\tau_2)$. ◇

Preuve. La preuve se fait par induction sur les dérivations de typage. □

Lemme 13.8.4 (Substitution de type dans les expressions)
Soit ϕ *valant* $[\overline{\alpha} \mapsto \overline{\tau}]$.
Alors, $\Gamma_1, \overline{\alpha}, \Gamma_2 \vdash t : \tau\ \{P\}$ *et* $\overline{\alpha} \# \mathcal{D}(\Gamma_2)$ *implique* $\Gamma_1, \phi(\Gamma_2) \vdash \phi(t) : \phi(\tau_2)\ \{\phi(P)\}$. ◇

Preuve. La preuve se fait par induction sur les dérivations de typage. □

Lemme 13.8.5 (Substitution de valeur dans les valeurs)
Posons ρ *pour* $[x \mapsto \Lambda\overline{\alpha}.v']$. *Alors,* $\Gamma_1, (x : \forall \overline{\alpha}.\, \tau_1), \Gamma_2 \vdash v : \tau_2$ *et* $\Gamma_1, \overline{\alpha} \vdash v' : \tau_1$ *et* $x \notin \mathcal{D}(\Gamma_2)$ *implique* $\Gamma_1, \rho(\Gamma_2) \vdash \rho(v) : \tau_2$. ◇

Preuve. La preuve se fait par induction sur les dérivations de typage. Seul le cas de la règle VAR n'est pas standard. La conclusion de cette règle est

$$\Gamma_1, (x : \forall \overline{\alpha}.\, \tau_1), \Gamma_2 \vdash x' \overline{\tau}' : [\overline{\alpha}' \mapsto \overline{\tau}']\tau'$$

et son hypothèse est

$$(\Gamma_1, (x : \forall\overline{\alpha}.\,\tau_1), \Gamma_2)(x') = \forall\overline{\alpha}'.\tau' \qquad (\mathbf{1})$$

Si $x \neq x'$ alors $\rho(x'\overline{\tau}') = x'\overline{\tau}'$ et l'hypothèse (1) implique $(\Gamma_1, \rho(\Gamma_2))(x') = \forall\overline{\alpha}'.\tau'$. Par application de la règle VAR à ces deux jugements, on obtient le résultat.

Si $x = x'$ alors $\rho(x'\overline{\tau}') = [\overline{\alpha} \mapsto \overline{\tau}']v'$. L'hypothèse (1) s'écrit maintenant :

$$(\Gamma_1, (x : \forall\overline{\alpha}.\,\tau_1), \Gamma_2)(x) = \forall\overline{\alpha}.\tau_1$$

Donc $\overline{\alpha} = \overline{\alpha}'$ et $\tau_1 = \tau'$ (**2**). Par le lemme de substitution de type dans les valeurs, l'hypothèse de bon typage de de v' donne

$$\Gamma_1 \vdash [\overline{\alpha} \mapsto \overline{\tau}']v' : [\overline{\alpha} \mapsto \overline{\tau}']\tau_1$$

d'où le résultat découle grâce à (2). □

Lemme 13.8.6 (Substitution de valeur dans les expressions)
Posons ρ pour $[x \mapsto \Lambda\overline{\alpha}.v]$. Alors, $\Gamma_1, (x : \forall\overline{\alpha}.\,\tau_1), \Gamma_2 \vdash t : \tau_2\ \{P\}$ et $\Gamma_1, \overline{\alpha} \vdash v : \tau_1$ et $x \notin \mathcal{D}(\Gamma_2)$ implique $\Gamma_1, \rho(\Gamma_2) \vdash \rho(t) : \tau_2\ \{\rho(P)\}$. ◇

Preuve. Par induction sur les dérivations de typage. □

Lemme 13.8.7 (Analyse par cas)
Supposons $\emptyset \vdash v : \tau$ et $\Gamma' \vdash p : \tau$ et $p \,\#\, v$. Alors, $[p \mapsto v]$ est définie si et seulement si la formule $\exists\Gamma'.\lceil v\rceil = \lceil p\rceil$ est valide. ◇

Preuve. La preuve se fait par induction sur le motif p. Ce sont des conséquences directes des propriétés exigées pour le modèle et de la définition de la substitution généralisée.

Si $[p \mapsto v]$ est définie alors elle est de la forme

$$[D\,\overline{\tau}\ (p_1, \ldots, p_n) \mapsto D\,\overline{\tau}\ (v_1, \ldots, v_n)]$$

L'induction nous affirme la validité des formules $\exists\Gamma'_i.\lceil v_i\rceil = \lceil p_i\rceil$. Comme les motifs sont linéaires les $(\Gamma'_i)_{i\in 1\ldots n}$ sont disjoints. On peut construire sans précaution le préfixe $\Gamma' \equiv \Gamma'_1 \ldots \Gamma'_n$. L'interprétation standard de la quantification existentielle dans le modèle ainsi que la propriété 3 implique que la formule $\exists\Gamma'.\lceil D\,\overline{\tau}\ (p_1, \ldots, p_n)\rceil = \lceil D\,\overline{\tau}\ (v_1, \ldots, v_n)\rceil$ est valide dans le modèle.

Si $[p \mapsto v]$ n'est pas définie alors on sait tout de même que le bon typage du motif p implique qu'il est de la forme $D'\,\overline{\tau}\ (p_1, \ldots, p_m)$ et que le bon typage de la valeur v implique qu'elle est de la forme $D\,\overline{\tau}\ (v_1, \ldots, v_n)$ avec $D \neq D'$. D'après la propriété 5 du modèle, la formule $\exists\Gamma'.\lceil D'\,\overline{\tau}\ (p_1, \ldots, p_m)\rceil = \lceil D\,\overline{\tau}\ (v_1, \ldots, v_n)\rceil$ est invalide quelque soit Γ'. □

Théorème 13.8.8 (Préservation des types)
$\Gamma \vdash t : \tau\ \{P\}$ et $t \to t'$ implique $\Gamma \vdash t' : \tau\ \{P\}$. ◇

Preuve. Par induction sur les dérivations d'évaluation.

▷ *Cas* « APP ». Notons v_f, la fonction

$$\mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t$$

Par inversion de la règle APP sur l'hypothèse $\Gamma \vdash v_f\ (v) : \tau_2\ \{P\}$, nous avons :

$$\Gamma \vdash v_f : \tau_1 \dashrightarrow \tau_2 \qquad (\mathbf{1})$$

$$\Gamma \vdash v : \tau_1 \qquad (\mathbf{2})$$

$$\Gamma \models \mathsf{pre}(\lceil v_f\rceil)\ (\lceil v\rceil) \qquad (\mathbf{3})$$

$$\Gamma \models \mathsf{post}(\lceil v_f \rceil)\ (\lceil v \rceil) \Rightarrow P \tag{4}$$

Par inversion de la règle Fun :

$$\Gamma, (f : \tau_1 \dot{\longrightarrow} \tau_2), f = \lceil \mathsf{fun}\ f \ldots \rceil, (x_1 : \tau_1), F_1 \vdash t : \tau_2\ \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\} \tag{5}$$

Deux applications du lemme 13.8.6 sur (5) avec

$$\rho \equiv [x_1 \mapsto v][f \mapsto v_f]$$

donnent :

$$\Gamma, \lceil v_f \rceil = \lceil \mathsf{fun}\ f \ldots \rceil, \rho(F_1) \vdash \rho(t) : \tau_2\ \{\lambda(x_2 : \lceil \tau_2 \rceil).\rho(F_2)\} \tag{6}$$

Par la propriété 1 du modèle, $\lceil v_f \rceil = \lceil \mathsf{fun}\ f \ldots \rceil$ est équivalent à true.
Nous avons aussi que $\rho(F_1)$ est β-equivalent à $\mathsf{pre}(\lceil v_f \rceil)\ (\lceil v \rceil)$.
Ainsi, par la propriété 2 du modèle :

$$\Gamma \models \rho\ (F_1) \tag{7}$$

En appliquant deux fois le lemme 13.8.1 sur (6), on obtient :

$$\Gamma \vdash \rho(t) : \tau_2\ \{\lambda(x_2 : \lceil \tau_2 \rceil).\rho(F_2)\} \tag{8}$$

Comme $\mathsf{post}(\lceil v_f \rceil)\ (\lceil v \rceil)$ est $\beta\eta$-equivalent à $\lambda(x_2 : \lceil \tau_2 \rceil).\rho(F_2)$, la propriété 2 du modèle implique :

$$\Gamma \vdash \rho(t) : \tau_2\ \{\mathsf{post}(\lceil v_f \rceil)\ (\lceil v \rceil)\} \tag{9}$$

Les lemmes 13.8.2 et (4) donne le résultat :

$$\Gamma \vdash [x \mapsto v][f \mapsto v_f]t : \tau_2\ \{P\} \tag{10}$$

▷ *Cas* « Let-L ». Immédiat par hypothèse d'induction.

▷ *Cas* « Let-R ». Mêmes arguments que pour la règle App.

▷ *Cas* « Case ».

Par inversion de la règle Case en hypothèse, nous avons :

$$\Gamma \vdash v : \tau \tag{1}$$

$$\Gamma' \vdash p : \tau \tag{2}$$

$$\Gamma, \Gamma', v = p \vdash t : \tau'\ \{P\} \tag{3}$$

$$\Gamma, (\forall \Gamma'.v \neq p) \vdash \mathsf{match}\ v\ \mathsf{with}\ c : \tau'\ \{P\} \tag{4}$$

Sous-cas « $[p \mapsto v]$ est définie. »
Le lemme 13.8.7 $\mathcal{D}([p \mapsto v]) = \mathcal{D}(\Gamma')$.
Par application du lemme 13.8.6 à (3), nous avons :

$$\Gamma, \lceil v \rceil = \lceil v \rceil \vdash [p \mapsto v]t : \tau'\ \{[p \mapsto v]P\} \tag{5}$$

$[p \mapsto v]P \equiv P$ parce que Γ' n'est pas présent dans la conclusion de la règle et la règle est supposée bien formée.

Par application du lemme 13.8.1 et grâce à la propriété 1, nous obtenons la conclusion :

$$\Gamma \vdash [p \mapsto v]t : \tau' \ \{P\} \tag{6}$$

Sous-cas « $[p \mapsto v]$ n'est pas définie ».
Par application du lemme 13.8.7, nous avons que $\not\models \exists \Gamma'.\lceil p \rceil = \lceil v \rceil$ ce qui implique $\models \forall \Gamma'.\lceil p \rceil \neq \lceil v \rceil$.
Le lemme 13.8.1 sur (4) donne la conclusion :

$$\Gamma \vdash \mathsf{match}\, v \,\mathsf{with}\, c : \tau' \ \{P\}$$

□

Théorème 13.8.9 (Progression de l'évaluation sous environnement statique cohérent)
Soit un environnement de typage Γ *tel que* Γ *ne contient que des variables de types et des formules et tel que* $\Gamma \not\models \mathsf{false}$. $\Gamma \vdash t : \tau \ \{P\}$ *implique que* t *est soit réductible, soit une valeur* v *telle que* $\Gamma \models P\ (\lceil v \rceil)$ *est valide.* ◇

Preuve. Cette preuve se fait par induction sur le terme t. Comme le système est dirigée par la syntaxe, on a un cas par règle.

▷ *Cas* « Value ». La conclusion du typage est $\Gamma \vdash v : \tau \ \{P\}$ et ses prémisses sont $\Gamma \vdash v : \tau$ et $\Gamma \models P\ (\lceil v \rceil)$. Nous sommes dans le second cas de l'alternative. Il n'y a plus d'étapes d'évaluation à effectuer et la valeur obtenue vérifie P.

▷ *Cas* « App ». La conclusion du typage est $\Gamma \vdash v_1\ (v_2) : \tau_2 \ \{P\}$ et ses prémisses sont $\Gamma \vdash v_1 : \tau_1 \xrightarrow{\cdot} \tau_2$, $\Gamma \vdash v_2 : \tau_1$, $\Gamma \models \mathsf{pre}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil)$ et $\Gamma \models \mathsf{post}(\lceil v_1 \rceil)\ (\lceil v_2 \rceil) \Rightarrow P$. Une valeur close de type $\tau_1 \xrightarrow{\cdot} \tau_2$ est nécessairement une valeur de la forme $\mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t$. On a donc :

Fun
$$\frac{\begin{array}{c} f \# F_1, F_2 \\ \lceil (x_1 : \tau_1) \rceil \vdash F_1 : \mathsf{prop} \qquad \lceil (x_1 : \tau_1), (x_2 : \tau_2) \rceil \vdash F_2 : \mathsf{prop} \\ (f : \tau_1 \xrightarrow{\cdot} \tau_2), f = \lceil \mathsf{fun}\, f \ldots \rceil, (x_1 : \tau_1), F_1 \vdash t : \tau_2 \ \{\lambda(x_2 : \lceil \tau_2 \rceil).F_2\} \end{array}}{\Gamma \vdash \mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t : \tau_1 \xrightarrow{\cdot} \tau_2}$$
□

Par la règle β-reduction, le terme $v_1(v_2)$ se réduit en $[x \mapsto v_2][f \mapsto v_1]t$ car $v_1 \equiv \mathsf{fun}\, f(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t$.

▷ *Cas* « Let ». La conclusion du typage est $\Gamma \vdash \mathsf{let}\,(x\,\overline{\beta} : \tau_1/F) = t_1\ \mathsf{in}\ t_2 : \tau_2 \ \{P\}$ et ses prémisses sont $x \# P$, $\Gamma\overline{\beta}\lceil (x : \tau_1) \rceil \vdash F : \mathsf{prop}$, $\Gamma\overline{\beta} \vdash t_1 : \tau_1 \ \{\lambda(x : \lceil \tau_1 \rceil).F\}$ (**1**) et $\Gamma(x : \forall\overline{\beta}.\tau_1), \forall\overline{\beta}.[x \mapsto x\,\overline{\beta}]F \vdash t_2 : \tau_2 \ \{P\}$ (**2**). On peut appliquer l'hypothèse d'induction sur le terme t_1 grâce à (1). Donc, soit t_1 se réduit et dans ce cas grâce à l'application de la règle Context, le terme $\mathsf{let}\,(x\,\overline{\beta} : \tau_1/F) = t_1\ \mathsf{in}\ t_2$ se réduit. Soit t_1 est une valeur v_1 telle que $\Gamma \models P\ (\lceil v_1 \rceil)$. On peut alors réduire $\mathsf{let}\,(x\,\overline{\beta} : \tau_1/F) = v_1\ \mathsf{in}\ t_2$ en $[x \mapsto \Lambda\overline{\beta}.v_1]t_2$.

▷ *Cas* « Case-Nil ». Ce cas est exclu de l'évaluation puisqu'on a supposé l'invariant $\Gamma \not\models \mathsf{false}$ et que $\Gamma \models \mathsf{false}$ est une prémisse de la règle de typage de Case-Nil.

▷ *Cas* « Case-Cons ». Le terme $\mathsf{match}\, v\ \mathsf{with}\ p.t \,[]\, c$ n'est pas une valeur. Soit $[p \mapsto v]$ est définie et on applique la règle E-Case-Cons-Match au terme t. Soit $[p \mapsto v]$ n'est pas définie et on passe aux cas suivants à l'aide de la règle E-Case-Cons-Fail.

Comme d'habitude un programme dans un état d'erreur est un terme clos qui ne peut pas s'évaluer et qui n'est pas une valeur. Un programme bien typé s'évalue sans erreur et, quand il termine, la valeur calculée vérifie la spécification.

Théorème 13.8.10 (Correction du système de preuve)
Si $\emptyset \vdash t : \sigma \ \{P\}$ *et si il existe un terme* t' *non réductible tel que* $t \to^\star t'$ *alors* t' *est une valeur* v *et* $P\ (\lceil v \rceil)$ *est vraie.* ◇

Preuve. Par induction sur le nombre d'étapes d'évaluation. On peut appliquer les deux théorèmes précédents car $\emptyset \not\models \mathsf{false}$. □

```
type option (a) = None | Some of a

let unSome (o) where o ≠ None returns (x) where (o = Some (x)) =
match o with
  | Some (x) → x
end
```

FIG. 13.8: Utilisation du type de données « option ».

13.9 Exemples

Dans cette section, nous détaillons deux exemples très simples pour illustrer le fonctionnement du système de preuves.

13.9.1 Manipulation du type *option*

Parfois, une fonction renvoie un résultat de type τ sur un domaine restreint du type d'entrée τ'. Lorsque l'on sort de ce domaine, aucune valeur du type τ n'est *a priori* correcte. On peut choisir une valeur par défaut pour dénoter le fait que la fonction n'a pas pu calculer de résultat, on peut aussi choisir d'utiliser le type de données « option τ » défini par la figure 13.8.

Le constructeur « None » dénote l'absence de valeur, le constructeur « Some » représente au contraire la valeur calculée de type τ. Il arrive qu'un argument externe permette de déduire que la valeur optionnelle est en fait différente de « None ». Dans ce cas, il est correct d'utiliser la fonction « unSome » définie dans la figure 13.8. Elle rejettera les applications mal formées en passant « None » en argument. En effet, comme le montre l'instanciation incorrecte de la règle APP suivante. Il serait nécessaire de montrer que $\Gamma \models \mathsf{pre}(\text{unSome})\ (\text{None})$, ce qui reviendrait à montrer que $\Gamma \models \text{None} \neq \text{None}$.

$$\textsc{App}\ \frac{\Gamma \vdash \text{unSome} : \text{option(a)} \dot{\longrightarrow} \text{a} \qquad \Gamma \vdash \text{None} : \text{option(a)} \qquad \Gamma \models \mathsf{pre}(\text{unSome})\ (\text{None}) \qquad \Gamma \models \mathsf{post}(\text{unSome})\ (\text{None}) \Rightarrow P}{\Gamma \vdash \text{unSome}\ (\text{None}) : \text{option}\ (\text{a})\ \{\mathsf{false}\}}$$

Si on se focalise maintenant sur la définition de la fonction « unSome », on voit que le corps de la fonction est traité sous l'hypothèse que « o ≠ None » et qu'on doit montrer la postcondition « o = Some a (x) » pour le résultat « x » du calcul.

$$\textsc{Lam}\ \frac{\begin{array}{c}\Gamma, (\text{unSome} : \text{option(a)} \dot{\longrightarrow} \text{a}), \\ \text{unSome} = (\lambda(\text{o} : \text{option(a)}).\text{o} \neq \text{None}, \lambda(\text{o} : \text{option(a)})(\text{x} : \text{a}).\text{o} = \text{Some a (x)}) \\ (\text{o} : \text{option(a)}), \text{o} \neq \text{None} \\ \vdash t : \text{a}\ \{\lambda(x : \text{a}).\text{o} = \text{Some a (x)}\}\end{array}}{\Gamma \vdash \mathsf{fun}\ \text{unSome(o} : \text{option(a)/o} \neq \text{None}) : (\text{x} : \text{a/o} = \text{Some a (x)}) = \ldots : \text{option(a)} \dot{\longrightarrow} \text{a}}$$

Pour cela, deux obligations de preuves sont générées. La première correspond à la branche du `match`. On la distingue en observant la règle de traitement de l'analyse par cas.

$$\textsc{Case-Cons}\ \frac{\begin{array}{c}\Gamma \vdash \text{o} : \text{option(a)} \qquad (\text{x} : \text{a}) \vdash \text{Some a (x)} : \text{option(a)} \\ \Gamma, (\text{x} : \text{a}), \text{o} = \text{Some a (x)} \vdash \text{x} : \text{a}\ \{\lambda x.\text{o} = \text{Some a (x)}\} \\ \Gamma, (\forall(\text{x} : \text{a}).\text{o} \neq \text{Some a (x)}) \vdash \mathsf{match}\ v\ \mathsf{with}\ \emptyset : \text{option(a)}\ \{\lambda x.\text{o} = \text{Some a (x)}\}\end{array}}{\Gamma \vdash \mathsf{match}\ \text{o}\ \mathsf{with}\ \text{Some a (x).x} \parallel c : \text{a}\ \{\lambda x.\text{o} = \text{Some a (x)}\}}$$

Par application de la règle sur les valeurs, l'obligation de preuve est (*modulo* une η-expansion) :

$$\Gamma, (\mathrm{x} : \mathrm{a}), \mathrm{o} = \mathrm{Some\ a\ (x)} \models \mathrm{o} = \mathrm{Some\ a\ (x)}$$

et elle est triviale.

Reste ensuite une application de la règle Case-Nil, qui s'instancie ainsi :

$$\frac{\Gamma, (\forall(\mathrm{x} : \mathrm{a}).\mathrm{o} \neq \mathrm{Some\ a\ (x)}) \models \mathsf{false}}{\Gamma, (\forall(\mathrm{x} : \mathrm{a}).\mathrm{o} \neq \mathrm{Some\ a\ (x)}) \vdash \mathsf{match}\ v\ \mathsf{with}\ \emptyset : \mathrm{option(a)}\ \{\lambda x.\mathrm{o} = \mathrm{Some\ a\ (x)}\}}$$

Comme $\Gamma \equiv \ldots, (\mathrm{o} : \mathrm{option(a)}), (\mathrm{o} \neq \mathrm{None})$, la seconde obligation de preuve est donc :

$$(\mathrm{o} : \mathrm{option(a)}), \mathrm{o} \neq \mathrm{None}, (\forall(\mathrm{x} : \mathrm{a}).\mathrm{o} \neq \mathrm{Some\ a\ (x)}) \models \mathsf{false}$$

ce qui est prouvable grâce à la propriété de complétude (propriété 4) imposée au modèle.

13.9.2 Recherche du minimum dans une liste

Nous passons maintenant à un exemple un peu moins trivial mais tout à fait classique : la recherche du plus petit élément d'une liste d'entiers. On suppose l'existence d'une relation d'ordre, notée $<$ et de type $int \to int \to \mathsf{prop}$, sur les entiers et d'une fonction, notée $<?$ et de type $int \to int \to bool$, décidant cette relation (c'est-à-dire telle que $\forall xy.\mathsf{post}(<?, (x, y), true) \leftrightarrow x < y$).

On se donne un type de données pour les listes d'entiers.

```
type list :
| Nil : list
| Cons : (int × list) → list
```

Un prédicat inductif est nécessaire pour définir la notion d'appartenance d'un entier à une liste.

```
inductive predicate in_list : (int × list) → prop =
| ∀ x l. in_list (x, Cons (x, l))
| ∀ x y l. in_list (x, l) ⇒ in_list (x, Cons (y, l))
```

Cette formulation inductive n'est pas très pratique pour le prouveur du premier ordre. On prouve donc (en Coq), le lemme suivant :

```
lemma : ∀ z x l. in_list (z, Cons (x, l)) ⇒ x = z ∨ in_list (z, l)
```

Nous sommes prêts pour écrire la fonction de recherche du minimum d'une liste non vide[1] :

```
let rec min (l) where l ≠ Nil returns (x) where ∀ y. in_list (y, l) ⇒ x ≤ y =
  match l with
  | Cons (x, Nil) → x
  | Cons (x, xs) →
    let y = min (xs) in
      if x <? y then x else y
```

Cinq obligations de preuve sont générées. Nous allons détailler chacune d'elles.

1. Dans la première branche, on doit montrer que « x » vérifie bien la postcondition. L'obligation de preuve obtenue est

   $$\mathrm{l} = \mathrm{Cons\ (x,\ Nil)} \models \forall \mathrm{y.in\_list\ (y,\ l)} \Rightarrow \mathrm{x} \leq \mathrm{y}$$

   Cette obligation est automatiquement traitée par le prouveur automatique grâce au lemme montré plus haut.

[1] La fonction n'est pas écrite dans un style à récursion terminale pour simplifier la présentation.

2. Dans la seconde branche, on doit montrer que la précondition de la fonction « min » est bien vérifiée par la liste « l ». L'obligation de preuve obtenue est

$$\forall y.l \neq \text{Cons } (y, \text{Nil}), l = \text{Cons } (x, xs) \models xs \neq \text{Nil}$$

   et elle est immédiate pour le prouveur du premier ordre.

3. Dans la seconde branche du match, dans la première branche du if, on doit montrer que la variable « x » est le nouveau minimum. L'obligation de preuve s'écrit :

$$\begin{array}{l} \forall y.l \neq \text{Cons } (y, \text{Nil}), l = \text{Cons } (x, xs), \\ \forall z.\text{in\_list } (z, xs) \Rightarrow y \leq z, \\ x < y \\ \models \forall z.\text{in\_list } (z, \text{Cons } (x, xs)) \Rightarrow x \leq z \end{array}$$

   Grâce au lemme et aux axiomes concernant l'ordre $<$, le prouveur automatique décharge cette preuve sans problème.

4. Dans la seconde branche du match, dans la seconde branche du if, on doit montrer que la variable « y » reste le minimum. L'obligation de preuve est maintenant :

$$\begin{array}{l} \forall y.l \neq \text{Cons } (y, \text{Nil}), l = \text{Cons } (x, xs), \\ \forall z.\text{in\_list } (z, xs) \Rightarrow y \leq z, \\ \neg(x < y) \\ \models \forall z.\text{in\_list } (z, \text{Cons } (x, xs)) \Rightarrow y \leq z \end{array}$$

   qui est aussi facilement prouvée automatiquement.

5. Il reste enfin la preuve que l'analyse est complète. Ceci est prouvé automatiquement car la précondition nous dispense de l'analyse du cas « l = Nil ».

Plusieurs remarques peuvent être faites suite à l'explication de cet exemple.

D'abord, on peut remarquer qu'une seule preuve est manuelle : celle du lemme d'ordre général qui témoigne d'une propriété essentielle du prédicat « in_list ». Toutes les obligations de preuve à l'intérieur du programme ont été déchargées automatiquement. La méthodologie qui consiste à prouver manuellement des résultats généraux sur les prédicats et les objets logiques en les formulant au premier ordre pour qu'ensuite le prouveur du premier ordre puisse s'occuper de les instancier correctement semble être assez efficace.

Par ailleurs, on remarquera que la preuve de correction de la fonction n'a pas nécessité d'induction malgré son caractère récursif. D'une manière générale, la récursion produit elle-même le raisonnement par récurrence (*modulo* la terminaison de la fonction). Le programme est une sorte de squelette de preuve dont les obligations de preuve sont les feuilles.

CHAPITRE QUATORZE

Extensions

Pour le moment, nous avons seulement formalisé un langage de programmation noyau car nous voulions mettre l'accent sur la simplicité de sa logique de Hoare. Pour cette raison, nous avons omis certaines fonctionalités qui seraient utiles à une implémentation réaliste. Nous discutons maintenant quelques-unes de ces fonctionnalités en distinguant celles dont l'ajout n'est pas problématique de celles qui soulèvent des problèmes non triviaux.

14.1 Assertions supplémentaires

La construction suivante permet d'insérer une assertion logique en n'importe quel point du code :

$$\mathsf{assert}\ F\ \mathsf{in}\ t$$

Cette construction vérifie que la formule F est valide : une obligation de preuve est émise. Elle n'a aucun rôle calculatoire. Dynamiquement, elle se comporte comme le terme t. On peut aussi la voir comme un sucre syntaxique pour :

$$\mathsf{let}\ x\ \mathsf{where}\ F = ()\ \mathsf{in}\ t$$

Elle est particulièrement utile quand notre outil est utilisé conjointement à un prouveur automatique : si le prouveur échoue dans sa recherche de la validité d'une obligation de preuve, l'utilisateur peut affirmer des propriétés intermédiaires de manière à couper la preuve en étapes plus simples et plus petites (si l'obligation de preuve est bien valide) ou de manière à trouver ce qui est faux dans la spécification (si l'obligation de preuve est en fait invalide).

La construction absurd oblige false à être prouvable dans le contexte et marque un morceau de code comme étant inaccessible. Cette construction est un sucre syntaxique pour :

$$\mathsf{match}\ ()\ \mathsf{with}\ \emptyset$$

14.2 Variables et paramètres fantômes

Il est quelque fois tentant d'introduire explicitement une variable fantôme c'est-à-dire un nom pour le témoin d'une hypothèse existentiellement quantifiée. Pour cela, nous proposons d'écrire :

$$\mathsf{let\ logic}\ x : \theta / F\ \mathsf{in}\ t$$

Cette construction lie la variable x dans la formule F et dans le terme t. Cela nécessite la preuve de l'assertion $\exists(x : \theta).F$ et il faut aussi introduire la formule F comme nouvelle hypothèse dans le

contexte. Les assertions logiques du terme t peuvent faire référence à la variable x et leurs preuves peuvent exploiter l'hypothèse F. Néanmoins, les occurrences calculatoires de x à l'intérieur de t sont interdites car le « let logic » n'a pas de contenu calculatoire – il est effacé avant l'exécution.

De même, il peut parfois être désirable d'abstraire une fonction par rapport à un paramètre fantôme x, de cette façon :

$$\mathsf{fun}\, f(\mathsf{logic}\ x : \theta)(x_1 : \tau_1/F_1) : (x_2 : \tau_2/F_2) = t$$

Cette construction lie x dans F_1, F_2 et t. Notons que θ peut être un type logique arbitraire donc cette extension permet l'abstraction vis-à-vis d'une proposition ou d'un prédicat si nécessaire.

Les variables et paramètres fantômes peuvent être vus comme un sucre syntaxique qu'on peut traduire vers notre langage noyau (Nanevski *et al*, 2007). En effet, on peut prouver pour let logic que

$$\mathsf{let\ logic}\ x : \theta/F\ \mathsf{in}\ t \equiv \mathsf{assert}\ \exists(x : \theta).F\ \mathsf{in}\ [F' \mapsto \forall(x : \theta).F \Rightarrow F']t$$

La notation $[F' \mapsto \forall(x : \theta).F \Rightarrow F']t$ signifie qu'on rajoute un préfixe à toutes les formules du terme t pour y introduire la variable logique x.

De même, on peut prouver pour les paramètres fantômes que

$$\begin{gathered}\mathsf{fun}\ f\ (\mathsf{logic}\ x : \theta)(x_1 : \theta_1/F_1) : (x_2 : \theta_2/F_2) = t \\ \equiv \\ \mathsf{fun}\ f\ (x_1 : \theta_1/\exists(x : \theta).F_1) : (x_2 : \theta_2/\forall(x : \theta).F_1 \Rightarrow F_2) = [F' \mapsto \forall(x : \theta).F_1 \Rightarrow F']t\end{gathered}$$

Dans une implémentation réaliste, cependant, ces constructions doivent être primitives pour éviter une explosion de la taille des obligations de preuves.

14.3 Exceptions

En supposant les constructeurs d'exception déclarés statiquement, les exceptions sont une forme de programmation fonctionnelle pure : un programme qui utilise des exceptions peut aisément être traduit vers un langage fonctionnel pur (Spivey, 1990).

Si chaque fonction déclare les exceptions qu'elle peut lancer et fournit des postcondtions pour ses sorties exceptionnelles, étendre le langage de programmation avec des exceptions n'est pas un problème. Why (Filliâtre, 2003) offre une logique de Hoare qui supporte les exceptions par exemple.

14.4 Modules

La langage noyau présenté ici n'a pas de système de modules. Cependant, la modularité est une fonctionnalité clé dans le développement de programmes certifiés. L'intérêt d'un programme certifié est moindre par rapport à celui d'un composant certifié réutilisable. Dans un langage de programmation où les programmes sont annotés par des assertions, un système de module doit fournir non seulement des types abstraits dans le monde calculatoire mais aussi des types abstraits et des prédicats abstraits dans le monde logique (Leino & Nelson, 2002, Nanevski *et al*, 2007). Nous pensons que la conception d'un système de module qui offrirait des composants flexibles, légers et fortement typés est encore un problème ouvert.

14.5 Égalité entre types

Dans cette présentation, les types logiques θ et les formules logiques t sont des catégories syntaxiques distinctes (voir la figure 11.1). Par conséquent, la logique ne peut pas exprimer des équations entre types. Nous avons noté (comme d'autres auteurs (Nanevski *et al*, 2007)) que l'introduction d'égalité de type, de la forme $\tau = \tau$ et $\theta = \theta$ dans la syntaxe des formules permettrait d'encoder les types

algébriques généralisés ainsi qu'une certaine forme de module de première classe. Les égalités entre types permettent aussi de raisonner sur des structures de données dont le typage est non uniforme comme les files purement fonctionnelles de Haim Kaplan et Robert Tarjan (Kaplan & Tarjan, 1999).

CHAPITRE QUINZE

Application : des ensembles finis implémentés par des arbres AVL

En guise d'illustration de notre outil, nous avons traduit la bibliothèque d'ensemble finis de O'Caml dans notre langage de programmation. Elle utilise une représentation à base d'arbres de recherche équilibrés et ses opérations sont purement fonctionnelles ce qui en fait un bon candidat pour l'expérimentation.

15.1 Paramètres

Dans la suite, nous fixons un type "elt" pour les éléments de nos ensembles. Nous supposons l'existence d'un type algébrique "*bool*", dont les constructeurs de données sont "true" et "false". Nous supposons l'existence d'un test d'égalité sur les éléments, noté "=". C'est une fonction de type calculatoire elt × elt $\dot{\longrightarrow}$ *bool* et dont la spécification peut être écrite comme suit :

$$\mathsf{post}(=, x_1, x_2, b) \Leftrightarrow (b = \text{true} \Leftrightarrow x_1 = x_2)$$

De même, nous supposons donnée une relation d'ordre, notée "<", de type logique elt → elt → prop ainsi qu'une fonction de décision pour cet ordre de type calculatoire elt × elt $\dot{\longrightarrow}$ *bool*.

Dans un langage de programmation réaliste, notre implémentation des arbres binaires de recherche équilibrés serait un foncteur, paramétré par le type "elt", par la fonction "=", par la relation "<", par les axiomes de "<" et par la fonction "<".

Nous supposons l'existence d'un type abstrait, noté "set elt", au niveau logique muni des opérations standards de la théorie des ensembles (l'ensemble vide, l'ensemble singleton, l'union, la relation d'appartenance, *etc.*) et des axiomes ou théorèmes qui décrivent les propriétés de ces opérations. Dans un langage de programmation réel, ces objets seraient fournis par une bibliothèque standard (au niveau logique).

15.2 Définitions

La figure 15.1 contient la définition du type algébrique "tree", d'une fonction logique définie de manière inductive "elements" et d'un prédicat inductif "bst". Un arbre binaire est soit vide, soit formé d'un nœud binaire contenant un élément à sa racine, un sous-arbre gauche, un sous-arbre droit et un entier correspondant à la hauteur de l'arbre. Nos arbres binaires de recherche sont conçus pour implémenter une abstraction d'ensemble fini. La fonction logique "elements" associe à chaque arbre

```
type tree =
| Empty : tree
| Node : (elt × tree × tree × int) → tree

logic fun elements : tree → set elt =
| Empty → ∅
| Node (x, l, r, _) → {x} ∪ elements (l) ∪ elements (r)

inductive predicate bst : tree → prop =
| bst (Empty)
| ∀ x, l, r. h.
  (∀y. y ∈ elements (l) ⇒ y < x) ∧ bst (l) ∧
  (∀y. y ∈ elements (r) ⇒ x < y) ∧ bst (r) ∧
  ⇒ bst (Node (x, l, r, h))
```

FIG. 15.1: Définitions pour les arbres binaires de recherche.

```
let rec member (x, t) where bst (t)
returns u where b = true ⇔ x ∈ elements (u)
= match t with
  | Empty → false
  | Node (y, l, r, _) →
    if x = y then
      true
    else if x < y then
      member (x, l)
    else
      member (x, r)
  end
```

FIG. 15.2: Décision d'appartenance à un arbre binaire de recherche.

binaire l'ensemble fini qu'il représente. Elle est définie par induction sur les valeurs du type algébrique "tree". La propriété d'être un arbre binaire de recherche est définie par le prédicat inductif "bst".

Dans la définition de "bst", les types des variables universellement quantifiées "x", "l", "r", "h" et "y" sont inférés. Les types de la fonction "elements" et du prédicat "bst" peuvent aussi être inférés si nécessaire. En pratique ces annotations de types peuvent toujours être omises à moins que la récursion polymorphe soit utilisée (Henglein, 1993).

La définition de "bst" ne contraint ni la forme de l'arbre ni l'entier correspondant à sa hauteur. Ceci est obtenu par un autre prédicat inductif, appelé "avl". Contrairement à l'approche à base de « types dépendants » (Xi, 1998, Altenkirch *et al*, 2005, Sozeau, 2006) et de « types algébriques généralisés » (Xi *et al*, 2003), nous favorisons un style de programmation dans lequel les invariants ne sont pas nécessairement vissés à l'intérieur des structures de données par le biais de leurs déclarations de type. En clair, on préfère écrire t : tree where $bst(t)$ plutôt que t : bst_tree.

15.3 Test d'appartenance à un arbre binaire de recherche

La figure 15.2 montre une fonction, "member", qui vérifie si un élément "x" appartient à un arbre "t". La précondition "bst(t)" demande à ce que "t" soit un arbre binaire de recherche, mais n'exige pas qu'il soit équilibré comme ce n'est pas nécessaire pour obtenir la correction. Si on souhaite s'assurer

```
type iterator = list tree

logic fun remaining : iterator → set elt =
        | [] → ∅
        | t : : ts → elements (t) ∩ remaining (ts)

inductive predicate ok : iterator → prop =
| ok ([])
| ∀ t, ts. elements (t) ∩ remaining (ts) = ∅ ∧
          bst (t) ∧ ok (ts) ⇒ ok (t : : ts)

let iterator (t) returns i
where ok (i) ∧ elements (t) ≡ remaining (i)
= [t]

let next (i)
where ok (i)
returns oix : option (iterator × elt)
where oix = None ⇒ remaining (i) = ∅ ∧
        ∀i', x. oix = Some (i', x) ⇒
           remaining (i) ≡ {x} ∪ remaining (i')
           x ∉ remaining (i') ∧ ok (i')
= match i with
   | [] → None
   | Empty : : ts → next (ts)
   | Node (x, l, r, _) : : ts → Some (l : : r : : ts, x)
   end
```

FIG. 15.3: Itérateur du premier ordre sur des arbres.

(informellement) de la borne de complexité logarithmique, on peut renforcer cette précondition en rajoutant la propriété “avl(t)” dans la précondition. Ceci illustre comment une structure de données peut être équipée de multiples invariants, tout en permettant à certains de ne pas être maintenus par moments. La postcondition affirme que le booléen calculé indique si “x” est un membre de l’ensemble implémenté par l’arbre “t”. Aucune annotation de types n’est nécessaire dans cette définition. Tous les types sont inférés.

15.4 Itération du premier ordre

Nous définissons et spécifions maintenant des itérateurs persistants (Filliâtre, 2006), du premier ordre, sur les arbres binaires de recherche. Leur pouvoir d’expression surpasse celui de “fold” (voir section 15.5) et pourtant leur spécification est plus simple.

L’implémentation apparaît dans la figure 15.3. Un itérateur est représenté par une liste d’arbres, ce qui peut être vu comme la pile d’un parcours en profondeur dans un arbre plus grand ou bien encore comme un zipper (Huet, 1997).

À chaque itérateur “i” correspond un ensemble d’éléments, que nous écrivons “remaining(i)”. La définition inductive est simplement l’union des éléments des arbres apparaissant dans la liste.

Un itérateur est bien formé seulement si les arbres qu’il contient ont des ensembles disjoints d’éléments. Ceci est exprimé par le prédicat inductif “ok”.

La fonction “iterator” créée un itérateur “i” à partir d’un arbre “t” et satisfait la postcondition “ok(i) ∧ elements(t) ≡ remaining(i)” où ≡ correspond à l’égalité extensionnelle sur les ensembles (qui

```
let  cardinal (t) returns n where n = ‖ elements (t) ‖ =
   let rec count (i, n)
   where ok(i) ∧ n + ‖ remaining (i) ‖ = ‖ elements (t) ‖
   returns n' where n' = ‖ elements (t) ‖
      = match next (i) with
      | None → n
      | Some (i', _) → count (i', n + 1)
      end
   in count (iterator (t), 0)
```

FIG. 15.4: Un client très simple de l'itérateur du premier ordre.

peut, éventuellement, coïncider avec l'égalité définitionnelle). Cet itérateur initial est simplement la liste singleton $[t]$.

La fonction "next", quand elle est appliquée à un itérateur "i", ne retourne rien ou retourne la paire d'un nouvel itérateur "i'" et d'un élément "x". La postcondition décrit comment ces valeurs sont liées.

La figure 15.4 montre comment ces itérateurs peuvent être utilisés. Ici, le client est une fonction qui compte le nombre d'éléments dans un arbre. Il ne dépend pas de la manière dont sont implémentés les arbres en interne : il dépend uniquement de la spécification des itérateurs, qui sont exprimés en terme d'ensembles (logiques) abstraits. De cette manière le code du client peut être placé dans autre module sans accès à la définition des arbres.

La fonction "cardinal" effectue une boucle, exprimée par une fonction récursive interne, à l'aide d'un accumulateur entier n. La précondition de cette fonction interne représente l'invariant de boucle : le nombre d'éléments comptés jusqu'à maintenant auquel on additionne le nombre d'éléments restants à traiter est égal au nombre total d'éléments de l'ensemble. La postcondition est simplement la précondition spécialisée au cas où aucun élément ne reste à traiter.

La précondition de la fonction "count" doit aussi affirmer que "i" est un itérateur vérifiant la propriété "ok" bien qu'elle n'ait pas à connaître la définition de "ok". Ceci est quelque peu gênant. Dans le futur, nous voulons permettre la définition de type existentiel de la forme « i : iterator where ok(i) » de manière à pouvoir exporter un type abstrait vérifiant un invariant. Il n'y a pas de difficulté s'opposant à cela.

La définition de "cardinal" est syntaxiquement lourde, comme elle est exprimée dans notre langage noyau. Dans une implémentation réaliste, une syntaxe plus pratique pour les boucles pourrait être introduite et désucrée en une fonction récursive munie d'itérateurs. Une simple formule, l'invariant de boucle, pourrait alors être écrite plutôt que les deux formules quasi-identiques de cette version bas niveau du code.

15.5 Itération d'ordre supérieur

Nous présentons maintenant une spécification de la fonction d'ordre supérieur classique "fold" sur les ensembles implémentés par des arbres binaires de recherche. La spécification est légèrement plus complexe que celle des itérateurs du premier ordre pour au moins deux raisons. En premier lieu, la spécification doit mentionner l'état du client (l'accumulateur) et son invariant. Ensuite, comme le code n'est pas sous la forme d'une récursion terminale, certaines informations sont implicitement encodées dans la pile. Un paramètre fantôme sert à les expliciter dans la spécification (voir chapitre 14).

La définition "is_invariant" résume ce qu'on attend d'un invariant de parcours d'arbre. Ses paramètres sont "inv", l'invariant du client, qui est lui-même paramétré par rapport à un accumulateur et un ensemble d'éléments restants ; "s_1", l'ensemble initial contenant tous les éléments ; la fonction cliente "f" ; et un accumulateur "accu". En somme, cette définition affirme que :

```
predicate is_invariant (inv, s0, s1, f, accu) =
  s0 ⊆ s1 ∧ inv (accu, s0) ∧
  ∀ s x s' accu accu'.
    s ⊆ s0 ∧ s = s' ∪ {x} ∧ x ∉ s' ∧ inv (accu, s) ⇒
      pre (f) (accu, x) ∧
      post (f) (accu, x) (accu') ⇒ inv (accu', s')

let rec fold (logic s0, logic inv, accu, t, f)
where is_invariant (inv, s0, elements (t), f, accu) ∧ bst (t)
returns accu' where inv (accu', s0\ elements (t)) =
  match t with
  | Empty → accu
  | Node (x, l, r, _) →
    let accu = fold (s0, inv, accu, l, f) in
    let accu = f (accu, x) in
      fold (s0\ (elements (l) ∪ {x}), inv, accu, r, f)
  end

predicate cardinal_inv (t) =
  λ (accu, s). accu + ‖ s ‖ = ‖ elements (t) ‖

let cardinal (t) returns x where x = ‖ elements (t) ‖ =
  fold (elements (t), cardinal_inv (t), 0, t, λ (accu, x). accu + 1)
```

FIG. 15.5: Parcours d'un arbre binaire à l'aide d'une fonction d'ordre supérieur.

- l'ensemble des éléments restants est un sous-ensemble de l'ensemble initial ;
- l'accumulateur courrant et l'ensemble courant des éléments restants satisfont l'invariant restant ;
- à chaque instant, si un élément "x" est pris dans l'ensemble des éléments restants, l'invariant garantit qu'on peut appliquer "f" à l'accumulateur courant et à "x" et que le nouvel accumulateur ainsi obtenu satisfait l'invariant.

La fonction "fold" est paramétrée par rapport à deux variables fantômes, l'invariant du client "inv" et l'ensemble "s_0" des éléments restants. Dans le cas des itérateurs du premier ordre, le premier n'était pas nécessaire parce que le client gardait le contrôle de l'invariant désiré, le second n'était pas non plus nécessaire parce que l'ensemble des éléments restants était directement exprimé par l'itérateur "remaining(i)". Ici, l'ensemble des éléments restants est implicite dans la pile, c'est pourquoi une variable fantôme doit être utilisée pour y faire référence.

Cette définition peut sembler assez complexe. Elle montre en même temps qu'il est possible de spécifier et d'exploiter des fonctions d'ordre supérieur dans notre système. Plus d'expériences seront nécessaires pour déterminer si on peut facilement définir et spécifier des fonctions d'ordre supérieur en pratique.

15.6 Quelques chiffres

Pour terminer, nous rendons compte de quelques données expérimentales. Dans la figure 15.6, on trouve le nombre d'obligations de preuves générées pour chaque fonction du module. Le prouveur utilisé est Ergo.

On remarquera tout d'abord que sur les 804 obligations de preuves, seul un lemme est prouvé manuellement en Coq. Il s'agit de la propriété de positivité de la fonction *height*. En effet, cet énoncé nécessite une induction. Compte tenu de la simplicité de cette preuve, on pourrait augmenter le langage

Fonction	**Preuves**	< 5s	< 30s	< 600s	manuelle
is_empty	20	20	0	0	0
height ?	5	5	0	0	0
mk_node	18	18	0	0	0
bal	179	143	23	13	0
add	39	33	6	0	0
join	82	64	6	12	0
min_elt	9	6	2	1	0
max_elt	11	7	3	1	0
remove_min_elt	22	16	4	3	0
merge	32	27	5	0	0
concat	27	25	2	0	0
split	75	55	9	11	0
remove	40	30	2	8	0
union	81	66	0	15	0
inter	44	34	1	9	0
diff	44	34	0	10	0
subset	48	42	0	6	0
fold	18	11	4	3	0
next	10	10	0	0	0
lemmes	50	44	5	0	1
Total	804	652	67	85	1

FIG. 15.6: Nombre d'obligations de preuves générées.

de programmation à l'aide d'une stratégie de preuve précisée par le programmeur. Ainsi, on écrirait :

$$\mathsf{lemma\ (by\ induction)} : \forall t.\ \mathit{height}\ (t) \geq 0$$

pour appliquer le principe d'induction des arbres sur l'énoncé. Les obligations de preuves obtenues seraient alors automatiquement prouvées.

On note ensuite que certaines preuves nécessitent un temps de calcul assez important (de l'ordre de 30 secondes jusqu'à atteindre 10 minutes). En général, ces preuves font intervenir des raisonnements sur des unions et des différences d'ensembles. Il est souvent nécessaire de faire commuter des opérateurs et de re-parenthéser les formules en utilisant les propriétés de commutation, associativité et distributivité. Or, Ergo n'implémente pas un algorithme de *congruence closure* travaillant *modulo* AC. Il est donc handicapé par la combinatoire induite.

Pour finir, le fait de prouver automatiquement les spécifications d'ordre supérieur grâce à notre prouveur du premier ordre semble montrer que notre codage et nos choix de design sont adaptés aux idiomes de la programmation fonctionnelle.

CHAPITRE SEIZE

Travaux existants

Les origines de notre travail proviennent de la logique de Hoare (Floyd, 1967, Hoare, 1969). Les extensions de la logique de Hoare pour intégrer les fonctions récursives et d'ordre supérieur ont été intensivement étudiées à la fin des années 70 et au début des années 80 (Clarke, 1979, Apt, 1981, Damm & Josko, 1983, German *et al*, 1983, Goerdt, 1985).

En particulier, le problème de la complétude a reçu beaucoup d'attention avec un article d'Edmund Clarke (1979) qui a proposé une preuve montrant qu'il n'existe pas de logique de Hoare correcte, complète et « naturelle » pour un langage de programme équipé de procédures récursives, d'ordre supérieur et des variables globales. Le résultat de Clarke s'appuie sur l'hypothèse que les formules et les obligations de preuve sont écrites dans une logique du premier ordre et, comme écrit dans l'article, « de manière à refléter la syntaxe du langage de programmation ». Werner Damm et Bernhard Josko (Damm & Josko, 1983, German *et al*, 1983) ont fait remarquer qu'il suffit d'utiliser une logique d'ordre supérieur pour passer outre le résultat négatif de Clarke. Nous suivons ce dernier point de vue. La justification intuitive de cette approche est que, si les fonctions sont abstraites par rapport à des fonctions alors les spécifications doivent pouvoir être abstraites par rapport à des spécifications.

Notre travail a été fortement inspiré de différents travaux existants concernant des outils conçus pour vérifier des programmes impératifs (Detlefs *et al*, 1998b, Flanagan *et al*, 2002, Filliâtre, 2003, Filliâtre & Marché, 2004, Marché *et al*, 2004, Barnett *et al*, 2004a).

Nous cherchons à exploiter les forces de ces travaux tout en laissant de côté, au moins pour un moment, la programmation impérative et en mettant l'accent sur le polymorphisme et la modularité.

Notre méthode pour générer les obligations de preuve est particulièrement directe : elle est totalement spécifiée par les figures 13.3 et 13.7 du chapitre 13. En comparaison avec la méthode utilisée par ESC/Java (Flanagan & Saxe, 2001), nous évitons la traduction en « forme passive » parce que nous n'avons pas d'affectation. Nous évitons l'explosion exponentielle que peut engendrer l'interaction entre les séquences et les alternatives en exigeant des annotations systématiques (section 12.3) sur les séquences (c'est-à-dire les constructions `let` de notre langage).

La logique de Scott pour les fonctions calculables interprète les λ-termes dans un modèle dénotationnel où l'égalité implique, ou coïncide avec, l'équivalence observationnelle. Cette logique est munie un ensemble de règles de déduction correctes et permet de raisonner explicitement sur la divergence et l'égalité des calculs. Elle admet une variante en appel par valeur et en appel par nom. Cette logique a été implémenté par Robin Milner en 1972 (Milner, 1972). Des implémentations plus récentes (Agerholm, 1994, Bartels *et al*, 1996, Müller *et al*, 1999) embarque LCF à l'intérieur d'une certaine forme de logique d'ordre supérieur. Dans une approche similaire, John Longley et Andy Pollack (Longley & Pollack, 2004) injecte le noyau fonctionnel de Standard ML, *via* une sémantique dénotationnelle totalement abstraite, à l'intérieur de la logique d'ordre supérieur.

Notre approche est bien moins élaborée : en nous concentrant sur la correction partielle, en adoptant une stratégie d'évaluation par valeur, et en n'injectant que les valeurs dans la logique (et non les expressions), nous pouvons ignorer le problème de la non-terminaison et travailler dans un espace des valeurs qui n'a pas d'élément *bottom* ou d'ordre définitionnel.

Les outils et les approches qui se focalisent sur les langages de programmation paresseux, tels que Programmatica (Kieburtz, 2002, Hallgren *et al*, 2004), le traducteur Cover (Abel *et al*, 2005) ou la logique de Honda et Yoshida pour les fonctions d'ordre supérieur (Honda & Yoshida, 2004) nécessitent un raisonnement sur la non-terminaison ce qui engendre des obligations de preuves contenant des conditions au bord pour s'assurer de la bonne définition des objets. La simplicité de notre approche a un prix : notre système ne peut ni établir la terminaison d'une expression, ni raisonner sur l'égalité observationnelle des expressions.

Le système Programatica (Hallgren *et al*, 2004) permet d'annoter des programmes Haskell avec des assertions et d'en extraire des obligations de preuve. Les formules sont exprimées dans une logique *ad hoc*, un μ-calcul, appelée P-logique (Kieburtz, 2002) dont les termes sont des programmes Haskell. Par ce biais, la P-logique est capable d'exprimer des propriétés des programmes Haskell de manière très concise. Cependant, adopter une logique non standard signifie qu'il est impossible d'utiliser un prouveur automatique existant dans un premier temps.

ESC/Haskell (Xu, 2006) permet aussi d'annoter des programmes Haskell par des préconditions et des postconditions qui sont exprimées en Haskell. Bien que cette idée semble attirante, elle soulève des problèmes complexes. Par exemple, il est impossible d'écrire des propriétés quantifiées universellement (puisque ces propriétés ne sont pas exécutables). De plus, si un programme utilisé dans le monde logique diverge, il est difficile de lui donner une interprétation. Enfin, on doit encore développer un prouveur *ad hoc*, s'appuyant sur une évaluation symbolique de Haskell. Notre système n'est pas correct vis-à-vis d'une sémantique dynamique en appel par nom. Ceci découle de deux raisons principales.

D'abord, les expressions divergentes admettent false comme postcondition valide. Si une telle expression t_1 est le premier composant d'une séquence, comme dans "$\mathsf{let}\, x/\mathsf{false} = t_1\ \mathsf{in}\ t_2$", alors la seconde composante t_2 est traitée sous l'hypothèse false. En conséquence, toutes les obligations de preuve trouvées dans le terme t_2 sont trivialement satisfaites. Ceci est correct seulement si le terme t_2 n'est jamais exécuté ce qui est le cas avec une stratégie d'évaluation par valeur mais pas avec une stratégie en appel par nom. La seconde raison, c'est que dans une sémantique d'appel par nom, tout type est habité par la valeur *bottom* et certains types sont habités par des valeurs infinies. Ceci n'est pas reflété par notre injection des valeurs et des types calculatoires dans le monde logique.

Honda & Yoshida (2004) ont développé une logique de Hoare pour les langages de programmation avec des fonctions d'ordre supérieur. Bien que nos approches respectives puissent paraître proches, elles sont en vérité très différentes. Alors que nous n'injectons que les valeurs dans le monde logique, Honda et Yoshida permettent à des expressions éventuellement impures d'intervenir dans les formules. L'égalité de la logique est interprétée comme une équivalence observationnelle. Encore une fois, ceci rend non-standard la logique considérée ce qui implique la nécessité d'un prouveur *ad hoc* pour décharger les obligations de preuve automatiquement.

L'assistant de preuve Coq (The Coq development team, 2006) peut être utilisé comme un langage de programmation dans lequel les programmes sont à la fois développés et prouvés corrects. Le compilateur certifié Compcert (Leroy, 2006) illustre cette approche sous la forme d'un programme de taille importante qui a été développé suivant cette méthode. Cependant, Coq n'est pas (encore) tout à fait pratique pour programmer car, par exemple, il ne permet d'écrire que des fonctions qui terminent et qui sont pures.

Le langage de programmation Russell (Sozeau, 2006) étend Coq pour permettre la définition de programmes annotés par des assertions logiques dans le style de Hoare. Il y a beaucoup de similarités entre le travail de Matthieu Sozeau et le nôtre. Une différence technique importante est que nous

séparons le processus de vérification des types, qui est appliqué en premier lieu et reste classique, et le processus d'extraction des obligations de preuve, qu'on exécute lors d'une seconde phase, alors qu'en Russell, tout comme en Coq, la vérification des types et la preuve est la même activité. En particulier, Russell encourage l'utilisation de types indexés, comme $\mathit{list}\, n$ ce qui implique que lors du passage d'un argument effectif de type $\mathit{list}\, m$ à un paramètre formel de type $\mathit{list}\, n$, l'obligation de preuve $n = m$ est générée. Une autre différence, c'est que les termes de Russell sont élaborés en termes Coq alors que nous adoptons une approche moins fondationnelle en acceptant de faire confiance à un prouveur externe.

Le système HTT (théorie des types de Hoare) (Nanevski *et al*, 2006, 2007) est assez similaire à notre système dans la mesure où il possède un algorithme de vérification des types simple et un algorithme de génération des obligations de preuve. Il partage aussi notre utilisation d'une logique d'ordre supérieur ainsi que notre prédilection pour les types abstraits et la modularité. Néanmoins, il essaie aussi de répondre au problème des états modifiables et alloués dynamiquement.

Quelques auteurs (Barnett *et al*, 2004b, Gronski *et al*, 2006, Xu, 2006, Nanevski *et al*, 2007) autorisent du code à apparaître dans leurs spécifications. Ceci est motivé en partie par un désir de rendre les formules exécutables de manière à ne vérifier certaines assertions que durant l'exécution du programme, et aussi par peur que certaines fonctionnalités doivent être implémentées en deux exemplaires (un dans le monde calculatoire et un dans le monde logique). Nos choix techniques et de conception sont différents : nous considérons que tout programme est potentiellement impur et nous n'autorisons pas le code à apparaître dans les formules. Si le programmeur désire insérer une vérification dynamique, il doit le faire explicitement. De plus, nous pensons qu'en pratique, les opportunités de partager du code entre le monde logique et le monde calculatoire nous semblent négligeables : les listes et les séquences semblent être une des rares exceptions où l'implémentation et la spécification coïncident.

Les types indexés (Zenger, 1997, Xi, 1998, Sheard, 2005a) et les types raffinés (Davies, 2005) s'appuient sur ce qu'on appelle des indices. Les indices sont des éléments d'un domaine mathématique donné, tel que les ensembles finis arbitraires ou l'ensemble des entiers naturels. Les types sont enrichis par des contraintes sur ces indices ce qui permet d'exprimer des invariants, des préconditions et postconditions. La syntaxe des contraintes est restreinte de manière très subtile de manière à obtenir la décidabilité des implications entre contraintes (c'est-à-dire des obligations de preuve dans notre système). Les types algébriques généralisés (Xi *et al*, 2003), le sujet de la première partie de cette thèse, sont aussi une instance de cette idée où les indices sont des types, c'est-à-dire des termes du premier ordre. L'intérêt de cette approche réside dans le haut degré d'automatisation obtenu : le vérificateur de type joue le rôle d'un prouveur automatique complet. Dans un autre côté, cela vient au prix d'une restriction à une logique décidable. Le cas d'Omega (Sheard, 2005a) est quelque peu différent puisque le programmeur peut écrire des fonctions des types dans les types et raisonner ainsi de manière plus souple sur les indices. Néanmoins, il est très difficile de déterminer si la vérification des types d'Omega est décidable car aucune étude théorique n'est accessible à ce jour. En fait, notre décision d'utiliser une logique très expressive, et donc indécidable, est à notre avis la continuation logique de nos travaux sur les types algébriques généralisés présentés dans la première partie de cette thèse (Pottier & Régis-Gianas, 2006b,a).

Avec Concoqtion (Fogarty *et al*, 2007), Walid Taha *et al* offrent une implémentation effective des types indexés s'appuyant sur O'Caml (Leroy *et al*, 2004), pour la partie calculatoire, et Coq (The Coq development team, 2006), pour la partie logique. Ils enrichissent d'abord les paramètres de types d'O'Caml pour y injecter des expressions Coq arbitraires. Les déclarations de types algébriques sont généralisées pour autoriser la spécialisation des indices en fonction des constructeurs de données (suivant ainsi la philosophie des GADT mais en utilisant des expressions Coq dont l'expressivité est bien plus grande que les types ML). Les expressions sont enrichies pour intégrer un *pattern matching*

effectuant une inversion semblable à celle de Coq dans les indices. Pour dénoter une valeur calculatoire, il faut la refléter manuellement sous la forme d'un indice (un type singleton). Deux objets sont donc nécessaires pour dénoter une valeur calculatoire : son nom en tant qu'identifiant calculatoire et son nom en tant qu'identifiant de type (d'indice). Cette duplication a des conséquences néfastes sur la lisibilité des programmes puisque de nombreuses quantifications sont dupliquées et des annotations de type sont nécessaires pour lier les indices aux valeurs calculaires. Enfin, le *design* du langage oblige de nombreuses preuves à s'immiscer dans les programmes pour convertir par exemple explicitement une liste de type $list\ (n + m)$ en une liste de type de $list\ (n + m)$.

Plusieurs langages de programmation, inspiré de la correspondance de Curry-Howard, fournissent des types dépendants qui permettent au code et au preuve d'être exprimés et combinés dans un même programme (Altenkirch *et al*, 2005, Chen & Xi, 2005, Sheard, 2005b, Westbrook *et al*, 2005). Cela donne aux programmes un aspect plus auto-contenu mais demande en pratique à l'implémentation du langage de contenir un prouveur ce qui n'est pas une mince affaire. Comme ces systèmes ne font aucune distinction entre la vérification des types et la vérification des preuves, l'inférence de type y est souvent problématique. De plus, certains de ces systèmes ne distinguent pas nettement les valeurs calculatoires des valeurs logiques ce qui pose des problèmes de compilation (quelles sont les entités qui n'ont pas d'existence lors de l'évaluation et que la compilation peut donc effacer ?)

TROISIÈME PARTIE

Conclusion

CHAPITRE DIX-SEPT

Synthèse des travaux de cette thèse

Pour finir, nous résumons nos travaux dans ce chapitre et, en particulier, nous décrivons rapidement les implémentations qui n'ont pas été abordées pour le moment. Le rôle des implémentations est pourtant essentiel puisque très souvent, les choix théoriques suivent les conclusions tirées de l'expérimentation sur machine et réciproquement, l'approfondissement des propriétés théoriques conduit à l'amélioration de l'implémentation. Dans le chapitre suivant, nous donnons quelques directions possibles pour la recherche à venir.

17.1 Inférence de types pour les GADT

Dans la première partie de cette thèse, nous avons formalisé une *inférence de type stratifiée pour ML en présence de GADT*. Cet algorithme d'inférence de types est stratifié dans le sens où il sépare la partie complexe du système de types, pour laquelle un algorithme d'inférence « totale » de types n'est pas envisageable (dans notre cas, le traitement des GADT), de la partie plus simple pour laquelle un algorithme d'inférence correct et complet est connu (le noyau ML).

D'une manière générale, la méthode de conception d'une inférence stratifiée consiste à :

(i) expliciter syntaxiquement les constructions problématiques du point de vue de l'inférence de types (dans notre cas, la détermination des équations entre types et leur utilisation par la règle de conversion) ;

(ii) définir un algorithme d'inférence correct et complet pour ce langage ;

(iii) définir un algorithme d'inférence locale, *a priori* incomplet mais dont le comportement est facile à prédire, reconstruisant la plupart des annotations de type. Pour ce dernier point, les formes décrites dans le chapitre 7 constitue un outil général garantissant la localité de l'inférence.

Ce système a été implémenté en étendant un prototype existant auquel j'ai contribué et qui est dédié à l'inférence de ML telle qu'elle est décrite par François Pottier et Didier Rémy dans le chapitre 10 du livre « Advanced Topics in Types and Programming Languages » (Pottier & Rémy, 2005). Ce prototype traite non seulement ML mais aussi les annotations de type, la récursion polymorphe, l'introduction de variables de type quantifiées universellement et les rangées.

Cette extension a été relativement simple. Voici les différents étapes suivies :

1. Dans la syntaxe (et l'analyseur syntaxique) :
   - rajout des coercions $(t : \exists\overline{\gamma}.(\tau_1 \triangleright \tau_2))$ dans la syntaxe des termes de ML ;
   - extension des déclarations de types algébriques pour permet des schémas plus généraux pour les constructeurs de données ;
2. Dans la génération des contraintes de types :
   - récolte des équations entre types lors du traitement des clauses ;

– traitement des coercions de la forme $\exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)$ comme des applications d'une fonction dont le schéma de type est : $\forall\overline{\gamma}.\tau_1 \rightarrow \tau_2$; et vérification de leur validité à l'aide du système d'équations courant.

3. Rajout d'un module implémentant les formes décrites dans le chapitre 7 et leurs opérations d'unification, de normalisation et d'élagage.
4. Rajout d'un module implémentant l'inférence locale bidirectionnelle du système *Wob*.
5. Rajout d'un module implémentant l'inférence locale itérée du système *Ibis*.

Le fait de générer une contrainte de typage dans le même langage de contraintes que pour l'inférence de type de ML a permis la réutilisation du solveur de contraintes sans aucun changement. Au final, le nombre de lignes de code nécessaire à l'extension des GADT est très faible : les changement à effectuer dans le code existant est de l'ordre de la centaine de lignes de code et les nouveaux modules sont longs de quelques centaines. La modularité de l'inférence stratifiée a donc pu se vérifier concrètement lors de l'implémentation.

Le lecteur désireux de tester cette implémentation peut la télécharger ou bien utiliser l'interface WEB en ligne (Régis-Gianas, 2005).

La production d'analyseurs syntaxiques bien typés présentée dans le chapitre 9 a aussi été implémentée dans un prototype (Régis-Gianas, 2004). Ce prototype intègre les différentes variantes et optimisations que nous avons suggérées. Il n'avait pas d'autres objectifs que d'illustrer notre article et il n'est donc pas utilisable en pratique pour le développement de compilateur par exemple. Par contre, nous avons développé dans la foulée, avec François Pottier, un générateur d'analyseurs syntaxiques LR(1) (et non LALR(1)) qui intègre la plupart de nos idées. Cet outil, appelé Menhir (Pottier & Régis-Gianas, 2005), est disponible et il est utilisé par la communauté des programmeurs O'Caml.

17.2 Système de preuves pour les langages fonctionnels purs

Dans la seconde partie de cette thèse, nous avons formalisé un système de preuves pour un langage fonctionnel pur. Nous avons adapté la logique de Hoare aux langages de programmation fonctionnels purs en appel par valeur. Cette adaptation s'appuie très largement sur des outils connus : la logique d'ordre supérieur et la propagation descendante des assertions logiques. Notre contribution principale se trouve dans les choix techniques que nous avons pris.

Tout d'abord, notre solution au problème posé par le reflet des valeurs fonctionnelles consiste à n'utiliser dans le monde logique que leur spécification sous la forme d'un couple précondition/post-condition. Ce choix permet d'une part de stratifier le monde logique et le monde calculatoire, ce qui simplifie la vérification et la synthèse des types, l'effacement et la génération des obligations de preuve par rapport à une approche où le code peut apparaître dans les spécifications. D'autre part, il permet d'exprimer certaines spécifications de fonctions d'ordre supérieur dans une logique du premier ordre.

Ensuite, nous avons recours à une logique d'ordre supérieur car elle est plus naturelle pour exprimer les spécifications des fonctionnelles. Nous utilisons un encodage standard vers le premier ordre pour maximiser l'automatisation. Nos logiques sont polymorphiquement typées dans le style de Hindley Milner pour obtenir une inférence de type correcte et complète.

Enfin, le langage de programmation fonctionnel n'intègre pas de types dépendants car nous favorisons un style où seules les assertions logiques dénotent des propriétés arbitraires. Les types ML ne dénotent que des propriétés de sûreté qu'on sait inférer et prouver automatiquement.

Ces choix de conception ont été très largement influencés par le résultat d'une expérimentation entreprise au début de ces travaux. Après avoir étudié la capacité des GADT à encoder certaines classes d'invariants, nous avons voulu les étendre pour traiter des propriétés arbitrairement complexes. Beaucoup d'*aficionados* des types se sont dirigés vers l'enrichissement de l'algèbre de types par des indices (Fogarty *et al*, 2007), des fonctions calculant des types (Sheard, 2005a) ou des types

dépendants (Altenkirch *et al*, 2005). Nous étions aussi interéssés par cette approche mais la complexité reconnue des types dépendants nous inquiétait. Par ailleurs, les assertions logiques de la logique de Hoare nous semblaient une alternative intéressante car elles peuvent caractériser plusieurs objets à la fois tandis que les types sont liés aux termes individuellement.

Avec la conviction que seuls les choix pratiques devaient nous influencer, nous avons décidé d'implémenter un premier prototype pour expérimenter ces différents styles. La première hypothèse qui nous est venue à l'esprit, c'est que le langage de programmation devait permettre la définition de termes impurs, dans le sens où ils peuvent diverger ou effectuer des effets de bords. Dans un premier temps, nous avons choisi de ne pas séparer les termes logiques des termes du langage de programmation étant entendu que les termes impurs ne devaient pas apparaître dans le monde logique. Un simple argument de typage, une analyse de pureté, nous a permis de garantir cette dernière propriété. Ce premier prototype est composé de deux strates :

- un langage noyau implémentant une variante du calcul des constructions inductives avec des termes impurs (principalement la récursion arbitraire) mais qui ne peuvent pas apparaître dans les types ;
- un langage de surface dans lequel on a implémenté un algorithme d'inférence locale permettant de supprimer la plupart des applications explicites de types ainsi que le passage de certains arguments qu'on peut déterminer grâce à la dépendance entre valeurs et types.

Dans ce langage, lorsqu'un terme de type P est attendu et que P est de type Prop, on génère une obligation de preuve. Cette obligation de preuve est codée dans la logique du premier ordre (Ayache & Filliâtre, 2006) pour être traitée par un prouveur automatique ou bien elle est prouvée manuellement en Coq.

L'intérêt de ce prototype est de permettre la programmation avec des types dépendants et aussi la programmation avec des assertions logiques (en utilisant des arguments purement logiques de type Prop). Nous avons écrit plusieurs exemples non triviaux dans ce prototype dont une bibliothèque d'arbres équilibrés et une bibliothèque de files purement fonctionnelles s'appuyant sur un article assez complexe de Haim Kaplan et Robert E. Tarjan (1999).

Grâce à cette expérimentation, nous avons pu constater que les types dépendants n'apportent pas plus d'expressivité qu'une logique de Hoare. Au contraire, si les types dépendants favorisent la concision (il est plus court d'écrire qu'une liste a le type « $list\ n$ » plutôt que « $l : list$ where length $l = n$ »), ils ne favorisent pas la modularité des programmes et introduisent de conversion de types désagréables (un terme de type $list\ (n + m)$ n'a en général pas le type $list\ (m + n)$). Enfin, la stratification entre le langage de programmation et le langage des assertions logiques s'est imposée car la définition de l'extraction est immédiate, ce qui n'est pas le cas avec un système comme Coq (Letouzey, 2004) et l'analyse de pureté des termes n'est pas nécessaire puisque les termes du langage de programmation n'apparaissent plus dans les assertions logiques.

Une fois fixé sur nos choix, nous avons formalisé le système de preuves présenté dans la seconde partie de cette thèse. Nous avons enfin implémenté un prototype pour ce système. Le développement de ce système a été plus complexe que les précédents car il nécessite de nombreuses transformations et la stratification entre le langage calculatoire et le langage logique multiplie les catégories syntaxiques à traiter. Nous terminons ce chapitre par une description de l'architecture de ce prototype.

► **Syntaxe** De nombreuses transformations syntaxiques rentrent en jeu dans ce prototype. Si l'analyse syntaxique est aujourd'hui bien maîtrisée, la difficulté liée à la manipulation d'arbres de syntaxe avec lieur est encore sujet à réflexion (Aydemir *et al*, 2005). En effet, il est souvent nécessaire d'obtenir des noms frais dans le contexte lorsque l'on transforme ou complète les termes du langage de programmation. On peut utiliser une représentation de De Bruijn (1972) mais le traitement manuel des indices augmente les risques d'erreur. Nous avons préféré utiliser le système AlphaCaml (Pottier, 2005)

qui engendre automatiquement une bibliothèque d'opérations pertinentes pour une syntaxe d'ordre supérieur donnée. AlphaCaml s'appuie sur une représentation des variables comme un couple d'un identifiant et d'un entier. Cette implémentation permet de réécrire des programmes tout en maintenant le mieux possible les identifiants du source et en garantissant tout même la fraîcheur des variables.

Par ailleurs, contrairement à notre présentation théorique, nous avons choisi d'utiliser des fonctions n-aires (et donc des applications n-aires) dans la syntaxe. Ce choix est anodin du point de vue de la conception du système de preuve. Il a pour but d'expliciter la construction des fermetures qui est à notre avis important pour que le programmeur soit plus au fait du coût d'évaluation de son programme.

▶ **Mise sous forme A-normale** Le noyau théorique formalisé dans cette thèse est sous forme A-normale : tous les résultats intermédiaires des calculs sont nommés. Dans notre prototype, le programmeur peut suivre un style standard où il compose directement les calculs. La première passe effectue une transformation nommant les résultats intermédiaires en utilisant des noms frais mais préfixé par une information sur le contexte. Par exemple, le résultat de l'application d'une fonction f sera nommé f_result. En pratique, ce nommage intelligent est important pour faciliter les preuves effectuées manuellement en Coq.

▶ **Inférence des post-conditions** Une fois la mise sous forme A-normale effectuée, une seconde passe complète les annotations logiques manquantes sur les lets liant des valeurs ou des applications de fonctions. Cette transformation est purement syntaxique et maintient le bon typage du programme source.

▶ **Inférence des types** Nous avons implémenté une version originale de l'inférence de type de ML à base de contraintes de type. Habituellement, dans cette approche, on génère une contrainte de type à partir d'une succession de définitions *toplevel* et à l'aide de cette contrainte, on répond à deux questions :

1. Est-ce que le programme est bien typé ?
2. Quel est le schéma de type de la valeur définie ?

La satisfiabilité de la contrainte répond à la première question. Il faut alors rajouter dans l'implémentation une construction *ad hoc* dump dans la syntaxe des contraintes pour que le solveur puisse aussi répondre à la seconde. Par exemple, la résolution de la contrainte « let $x : \forall\alpha.\alpha \to \alpha$ in dump x » renvoie le schéma de x.

Nous avons besoin d'une information plus précise qu'un simple schéma de type pour notre application. En effet, pour générer des obligations de preuve dans une logique explicitement typée, il faut que la dérivation de typage soit codée à l'intérieur du terme (sous la forme d'annotations de type). La résolution des contraintes de typage doit donc conduire à la construction d'un terme explicitement typé correspondant au terme source annoté. Les présentations théoriques des contraintes de typage n'ont pas pris en compte de telles considérations pratiques. Nous avons donc étendu le langage des contraintes par des primitives de constructions de termes. Le solveur manipule ces termes comme des boîtes noires ce qui permet de garantir une *totale indépendance des contraintes et du solveur vis-à-vis de la syntaxe du langage explicitement typé.*

Plus précisément, la syntaxe des contraintes est maintenant :

Contraintes	$C ::=$	
Vérité	$\mid$	true
Égalité	$\mid$	$\tau = \tau$
Conjonction	$\mid$	$C \wedge C$
Quantification existentielle	$\mid$	$\exists \overline{\gamma}.C$
Quantification universelle	$\mid$	$\forall \overline{\alpha}.C$
Instanciation	$\mid$	$x \preceq \tau$
Définition	$\mid$	$\mathsf{def}\ x : \forall \overline{\alpha}[C].\tau\ \mathsf{in}\ C$
Annotation de type	$\mid$	τ
Constructeur de terme	$\mid$	$F(C, \ldots, C)$
Terme nommé	$\mid$	x

Trois nouvelles constructions ont été rajoutées.

L'annotation de type sert à produire un type syntaxique à partir d'un type τ pouvant contenir des variables de type liées existentiellement dans la contrainte. À l'aide de cette construction, on peut donc produire un type qui est la solution de la contrainte. Par exemple, la contrainte $C_1 \equiv \exists\alpha.\alpha = int \wedge \alpha$ se réduit vers le type syntaxique int.

Le constructeur de terme permet d'utiliser une fonction non interprétée pour construire un nouveau terme à partir des termes issus d'une liste de contraintes prises en argument. Par exemple, si $F(X1, X2)$ est le terme $(x : X1) + (y : X2)$, la résolution de la contrainte $F(C_1, C_1)$ produit le terme $F(int, int)$, qui, une fois interprété par le client du solveur, se lit $(x : int) + (y : int)$.

La construction x permet de produire le terme syntaxique associé à la variable x dans la contrainte. Par exemple, la contrainte $\mathsf{def}\ x : \forall\alpha.[F(C_1, C_1)]\alpha\ \mathsf{in}\ x$ se réduit vers le terme $(x : int) + (y : int)$.

► **Génération des obligations de preuve** Contrairement à la théorie, la vérification des types n'est pas faite en même temps que la production des obligations de preuve. Actuellement, l'implémentation fait confiance à l'inférence de type pour produire des termes bien typés. Un module supplémentaire pourrait vérifier ce fait à l'avenir. La collecte des hypothèses et la vérification par le prouveur automatique est faite au fur et à mesure du parcours du programme pour obtenir une interaction plus efficace avec l'utilisateur. Si une preuve n'est pas traitée par le prouveur automatique, un fichier Coq est produit et l'utilisateur en est informé. Par ailleurs, le codage des hypothèses de la logique d'ordre supérieur vers la logique du premier ordre est effectué incrémentalement pour plus d'efficacité. Enfin, un système de cache permet de mémoriser lorsqu'une preuve a déjà été effectuée lors d'une session précédente pour augmenter la réactivité du système. En effet, lorsque le nombre de preuves dépasse la dizaine, la recherche de preuves devient coûteuse en temps.

► **Codage de PHOL vers PFOL** Le codage de la logique d'ordre supérieure vers la logique du premier ordre est dirigée par les types ce qui justifie la production de termes explicitement typés pour éviter de rajouter des passes d'inférence de types supplémentaires.

► **Codage de PFOL vers FOL** Le codage de la logique du premier ordre polymorphiquement typée vers la logique du premier ordre non typée est aussi grandement facilité par la présence d'annotations de type sur tous les nœuds du programme.

► **Traduction de PHOL vers Coq** La traduction de la logique d'ordre supérieur polymorphiquement typée vers Coq est très simple. La seule difficulté a été d'introduire une déconstruction des n-uplets due à la présence de fonctions n-aires dans notre syntaxe de surface.

CHAPITRE DIX-HUIT

Perspectives

Ce dernier chapitre donne des éléments d'approfondissement et des pistes qui n'ont pas pu être traités durant ces trois années de thèse.

18.1 Inférence de types pour les GADT

18.1.1 Motifs plus riches

Le passage à des motifs profonds dans MLGX nécessite l'augmentation de la syntaxe des motifs par des coercions. Pour le comprendre, donnons-nous le type algébrique généralisé suivant :

$$
\begin{array}{l}
\mathsf{type}\ \mathrm{ty}\ \alpha = \\
\quad \mid \mathrm{Int} : \mathrm{ty}\ \mathit{int} \\
\quad \mid \mathrm{Bool} : \mathrm{ty}\ \mathit{bool} \\
\quad \mid \mathrm{Pair} : \forall \beta_1, \beta_2.\ \mathrm{ty}\ \beta_1 \times \mathrm{ty}\ \beta_2 \to \mathrm{ty}\ (\beta_1 \times \beta_2)
\end{array}
$$

Ce type algébrique généralisé permet de représenter un type statique α de manière dynamique avec la propriété que la déconstruction de cette représentation permet l'obtention d'information statique sur le type α. C'est une utilisation classique des GADT inspiré principalement du polymorphisme intensionnel (Crary & Weirich, 1999).

On veut maintenant encapsuler une telle représentation du type α et une valeur de type α derrière une quantification existentielle.

$$\mathsf{type}\ \mathrm{obj} = \mathrm{O} : \forall\ \alpha.\ \mathrm{ty}\ \alpha \times \alpha \to \mathrm{obj}$$

On peut ensuite vouloir écrire une fonction d'affichage fonctionnant sur le type « obj » :

```
let rec print = λ (o : obj).
  match o with
  | O (Int, x) → print_int x
  | O (Bool, x) → print_bool x
  | O (Pair (ty1, ty2), (x1, x2)) → print (O (ty1, x1)) ; print (O (ty2, x2))
```

Cette fonction est malheureusement rejetée par notre inférence locale. Tout d'abord, elle est incorrecte syntaxiquement car nous n'acceptons que les motifs plats de la forme $K\ \overline{\beta}\ x_1\ \ldots x_n$. Ensuite, on voit que le problème n'est pas seulement syntaxique. On aimerait que le type β de la valeur accompagnant la représentation dynamique de type soit raffiné au fur à mesure de l'analyse *à l'intérieur même du motif*. Cependant, le raffinement dans MLGX est explicite : il faut donc se donner une syntaxe pour les coercions à l'intérieur des motifs. On augmente donc la syntaxe des motifs ainsi :

$$\begin{array}{rcl} p & ::= & x \\ & | & K\,\overline{\beta}\,\overline{p} \\ & | & (p : \exists\overline{\gamma}.(\tau_1 \triangleright \tau_2)) \end{array}$$

Avec cette syntaxe, notre exemple s'écrit dans MLGX :

```
let rec print = λ (o : obj).
  match o with
  | O β (Int, (x : int ▷ β)) → print_int x
  | O β (Bool, (x : bool ▷ β)) → print_bool x
  | O β (Pair β1 β2 (ty1, ty2), ((x1, x2) : β1 × β2 ▷ β))) → print (O (ty1, x1)); print (O (ty2, x2))
```

Ces constructions supplémentaires dans la syntaxe des motifs peuvent être vu comme des sucres syntaxiques. En effet, on peut toujours décomposer une analyse sur un motif profond à l'aide de plusieurs analyses imbriquées utilisant uniquement des motifs plats. En pratique, cette transformation si elle est appliquée naïvement peut dupliquer du code (il ne faut factoriser le code des branches) mais elle ne pose aucun problème de typage supplémentaire.

On peut remarquer que cette utilisation des coercions à l'intérieur des motifs ne fonctionne pas dans un langage de programmation ne fixant pas l'ordre d'évaluation du filtrage comme O'Caml par exemple. Dans un tel cas, le programme précédent est rejeté car les égalités entre types obtenues par le filtrage de la première composante dont le type est $ty\ \alpha$ ne sont pas disponibles pour vérifier la coercion de la seconde composante.

Les règles de typage des motifs profonds dans MLGX peuvent donc s'exprimer de deux façons : l'une rendant accessibles les équations dans les composantes à droite du motif (mais alors on fixe un ordre d'évaluation gauche-droite du filtrage), l'autre ne les rendant pas accessibles (et alors l'ordre d'évaluation du filtrage est libre).

18.1.2 Inférence locale plus fine à la sortie des branches

Une classe de *pattern matching* pourrait être traitée plus finement par l'inférence locale. Il s'agit des *pattern matchings* dont le type n'est pas fourni sous la forme d'une annotation mais qui peut être déterminé en comparant les équations obtenues pour chaque branche. Si on reprend l'exemple de la fonction *eval* mais en supprimant l'annotation spécifiant son type de retour, on obtient le programme suivant qui est mal typé dans MLGX puisque les branches n'ont pas le même type :

$$\begin{array}{l} \mathsf{let\ rec}\ \mathit{eval} : \underline{\exists\gamma.\forall\alpha.\mathit{term}\ \alpha \to \gamma} = \lambda t. \\ \quad \mathsf{match}\ (t : \mathit{term}\ \alpha)\ \mathsf{with} \\ \quad\quad | \ \mathit{Lit}\ i \to i \\ \quad\quad | \ \mathit{Inc}\ t \to \mathit{eval}\ t + 1 \\ \quad\quad | \ \mathit{IsZ}\ t \to \mathit{eval}\ t = 0 \\ \quad\quad | \ \mathit{If}\ b\ t\ e \to \mathsf{if}\ \mathit{eval}\ b\ \mathsf{then}\ \mathit{eval}\ t\ \mathsf{else}\ \mathit{eval}\ e \\ \quad\quad | \ \mathit{Pair}\ \beta_1\ \beta_2\ a\ b \to (\mathit{eval}\ a, \mathit{eval}\ b) \\ \quad\quad | \ \mathit{Fst}\ \beta_1\ \beta_2\ t \to \mathit{fst}\ (\mathit{eval}\ t) \\ \quad\quad | \ \mathit{Snd}\ \beta_1\ \beta_2\ t \to \mathit{snd}\ (\mathit{eval}\ t) \end{array}$$

Les types de retour des branches ont été élagués ce qui est justifié dans le cas général. Cependant, dans ce cas particulier, les différents systèmes d'équations réécrivent tous la variable de type α. De plus, dans le cas de la branche traitant le motif *Pair*, pour éviter l'échappement des variables β_1 et β_2, le seul représentant valide à l'extérieur de la branche est le type α. Il serait donc pertinent de détecter un tel cas particulier – mais qui est aussi très courant – et d'inférer le type α pour le *pattern matching*.

18.1.3 Extension de l'algèbre de types

On pourrait étendre l'inférence stratifiée pour qu'elle traite les types indicés et ainsi améliorer l'inférence de type du langage DML (Xi, 2001) ou du langage OMEGA (Sheard, 2005a). Il serait aussi intéressant d'étendre l'algèbre des types pour traiter des objets plus complexes comme des arbres réguliers tels qu'on les trouve dans *C*duce (Benzaken *et al*, 2003) ou OCAMLDUCE. En effet, dans l'implémentation d'intégration des types XML dans O'CAML (Frisch, 2006), Alain Frisch a du faire cohabiter comme nous l'inférence de types de ML avec la vérification d'annotations de types liées au filtrage XML.

18.1.4 Inférence correcte et complète pour une restriction des GADT

Nous avons évoqué dans la section 5.5 la possibilité d'utiliser le fragment décidable de l'unification d'ordre supérieur, les *patterns* (Dowek *et al*, 1996) pour résoudre de manière correcte et complète les contraintes de type pour un langage forçant les *pattern matchings* à être annotés par un type rigide. Obtenir un tel résultat permettrait de fournir un algorithme d'inférence de type pour les GADT qui aurait de meilleures propriétés que notre inférence stratifiée. Néanmoins, il ne répondrait pas aussi bien que cette dernière au problème de l'extension d'un système ML existant car il serait beaucoup plus intrusif.

18.1.5 Applications de l'inférence locale à d'autres systèmes de type

L'inférence itérée et les formes peuvent être appliquées à d'autres systèmes de type. En particulier, dans le premier prototype lié à la preuve de programmes fonctionnels, nous avons utilisé ces outils pour traiter l'inférence dans le calcul des constructions inductives.

Pour cela, nous avons tout d'abord étendu la syntaxe des formes et leurs opérations pour y intégrer des lieurs et nous avons généralisé l'unification entre formes pour qu'elle travaille *modulo* α-conversion. Cette généralisation doit encore être formalisée mais elle ne semble pas présenter de difficulté. Des travaux de Christian Urban, Andrew Pitts et Murdoch Gabbay portent sur une unification très proche de la nôtre appelée unification nominale (Urban *et al*, 2004).

Ensuite, nous avons généralisé la forme des jugements de l'inférence locale itérée en permettant l'inférence des environnements de typage. Dans ce système, les jugements prennent la forme :

$$\Gamma \vdash t \Downarrow s \rightsquigarrow t' \Uparrow (s', \Gamma')$$

L'environnement de typage Γ, le terme t et la forme attendue s sont les entrées du système. L'environnement de typage inféré Γ', la forme inférée s' et le terme élaboré t' sont les sorties.

Dans ce système, on peut inféré le type des variables à l'aide de leur utilisation. Par exemple, le terme $\lambda(x : \bot).x + 1$ est élaboré en $\lambda(x : int).x + 1$ car

$$(x : \bot) \vdash x + 1 \Downarrow \bot \rightsquigarrow x + 1 \Uparrow (int, (x : int))$$

Néanmoins, ce système ne devine pas le polymorphisme et les dépendances. Ainsi, $\lambda x.x$ est rejeté et doit être écrit $\lambda\alpha.\lambda(x : \alpha).x$.

Ne pas chercher à inférer le polymorphisme est un choix raisonnable dans un système de type polymorphe d'ordre supérieur (Le Botlan & Rémy, 2003). De plus, l'algorithme d'inférence locale obtenu est très simple à implémenter lorsque l'on est muni d'un module traitant les formes avec lieurs.

Il serait intéressant d'étudier la pertinence de notre inférence locale sur des systèmes intégrant une relation de sous-typage comme dans les articles fondateurs de Benjamin Pierce et David Turner (2000) ou de Martin Odersky et Matthias Zenger (2001).

18.1.6 Optimisation du code manipulant des GADT

Un des intérêts des GADT est la possibilité de restreindre le nombre de cas à traiter dans un filtrage à l'aide d'un argument de typage statique (comme dans notre application du chapitre 9). Malheureusement, le compilateur de *pattern matching* d'O'Caml n'est pas au fait de cette possibilité. Ainsi, lorsqu'il manque des cas dans une analyse, il rajoute automatiquement une branche traitant les cas restant en levant une exception. Or, lui offrir la garantie que les cas écrits par le programmeur sont suffisants ouvre la voie à des optimisations. L'optimisation ultime étant la suppression totale de test si le filtrage n'a qu'un seul cas.

18.1.7 Implémentation dans O'Caml

Nous avons commencé l'implémentation de l'inférence stratifiée dans la distribution d'O'Caml. Cependant, la tache s'avère plus ardue que prévu. En effet, le système de type d'O'Caml est beaucoup plus riche que celui de MLGI : on y trouve des types abstraits issus d'un système de modules complexes, des variables de rangées pour traiter les objets ainsi que des variantes polymorphes qui ont atteint un niveau de sophistication et de complexité important. Il n'y a pas à l'heure actuelle de formalisation du système dans sa totalité si bien que rajouter les GADT sans précaution pourrait introduire des bugs à cause d'une interaction malencontreuse entre les différents propriétés du système. Par exemple, le typage des variantes polymorphes peuvent s'exprimer en rajoutant des inéquations entre variables de rangées dans le langage de contraintes. On devrait donc généraliser la totalité de notre système pour tenir compte non seulement d'égalités entre types mais aussi d'inégalités. Restreindre les indices des GADT à des types « simples » est difficilement envisageable à cause du polymorphisme et du système de modules.

18.2 Système de preuves pour les langages fonctionnels purs

18.2.1 Extensions

Parmi les extensions que nous avons proposé pour le système de preuves dans le chapitre 14, le développement d'un système de modules et l'intégration des égalités entre types dans la logique sont les plus ambitieuses.

Un système de modules devrait fournir des propriétés dans ses interfaces et la liaison des composants devrait générer des obligations de preuve pour prouver leur compatibilité. De nombreux travaux sont dédiés à ce problème (Coquand *et al*, 2003, Courant, 1997) et seront donc une base de départ à notre travail.

Les égalités entre types sont déjà présentes en Coq par exemple mais elles ne sont pas intégrées dans la règle de conversion (car cela pourrait rendre la vérification du typage indécidable). Lorsque l'on peut prouver une égalité entre deux types τ_1 et τ_2, on peut par contre convertir explicitement un terme de type τ_1 en un terme de type τ_2 à l'aide de l'égalité dépendante (ou de John Major). Nous avons utilisé cette méthode pour prouver une structure de données dont le typage nécessite un type algébrique généralisé (Kaplan & Tarjan, 1999). Les programmes obtenus sont assez lourds (comme le sont ceux de MLGX) et on aimerait obtenir un système plus souple en pratique.

18.2.2 Recherche de contre-exemples

L'ergonomie du système pourrait gagner de l'intégration d'une recherche de contre-exemples. En effet, lorsque le prouveur automatique ne parvient pas à prouver une propriété, on ne sait pas si cela est du à l'incomplétude inhérente à ce type d'outils ou bien si la propriété est tout simplement fausse. Une première possibilité est de tester la négation de la propriété mais on peut encore obtenir une réponse négative à cause de l'incomplétude du prouveur. Une alternative intéressante est d'utiliser un

model checker en lui demandant de parcourir (intelligemment) le domaine des données pour trouver un contre-exemple. Par exemple, une interface vers Alloy (Jackson, 2006), un *model checker* permettant la recherche de contre-exemples (Jackson & Damon, 1996), serait donc utile mais elle nécessite un codage pertinent des prédicats inductifs dans la logique du premier ordre avec fermeture transitive qui est utilisée par Alloy.

18.2.3 Prouveurs automatiques dédiés aux langages fonctionnels

L'efficacité d'Ergo pourrait encore être améliorée pour s'adapter aux spécifications naturelles des langages fonctionnels. En effet, dans un langage fonctionnel, on favorise la manipulation de type abstrait implémentant des objets logiques abstraits comme des ensembles, des fonctions partielles, des termes, etc. Dans l'implémentation de notre bibliothèque d'ensembles finis, nous avons été confrontés à une impossibilité de prouver automatiquement des énoncés mettant en jeu la commutation et l'associativité des opérateurs ensemblistes. L'intégration des symboles associatifs et commutatifs augmenterait l'efficacité d'Ergo. D'autres propriétés et heuristiques sont encore à découvrir pour augmenter l'automatisation de la preuve de programmes fonctionnels.

18.2.4 Raisonnement *modulo* α-conversion

Les langages fonctionnels sont très utilisés pour écrire des compilateurs, des analyseurs et plus généralement des calculs symboliques. Dès lors, pour raisonner sur les objets syntaxiques avec lieurs, il serait intéressant d'intégrer les ensembles nomimaux (Pitts, 2003) dans la logique. On pourrait s'inspirer du travail de François Pottier dans Pure FreshML (Pottier, 2007) pour intégrer des raisonnements sur la fraîcheur des variables (dépendant éventuellement de propriétés arbitrairement complexes). Le système de preuves de Pure FreshML s'appuie aussi sur la logique de Hoare mais ses formules sont restreintes à des considérations d'échappements, de liaison et de fraîcheur des variables (ce qui permet une réduction de leur satisfiabilité vers SAT). De plus, les fonctions de première classe sont interdites dans Pure FreshML.

18.2.5 Utilisation encadrée des traits impératifs

Notre langage ne permet pas le raisonnement sur les programmes avec un état modifiable. Une direction de recherche est bien sûr de combler ce manque, elle est suivie par Arthur Charguéraud et François Pottier actuellement. Ils s'attaquent à la conception d'un langage impératif, muni d'annotations très précises sur la manipulation des états modifiables à l'aide de régions (Tofte & Talpin, 1994), qui est traduit, d'une manière la plus canonique possible, vers un langage fonctionnel pur similaire au nôtre.

Bibliographie

Abel, A., M. Benke, A. Bove, J. Hughes & U. Norell. 2005, *Verifying Haskell programs using constructive type theory*, *in Haskell workshop*, p. 62–73. URL http://www.tcs.informatik.uni-muenchen.de/~abel/haskell05.pdf.

Agerholm, S. 1994, *A HOL basis for reasoning about functional programs*, rapport technique RS-94-44, BRICS. URL http://www.brics.dk/RS/94/44/BRICS-RS-94-44.ps.gz.

Aho, A. V., R. Sethi & J. D. Ullman. 1986, *Compilers : Principles, Techniques, and Tools*, Addison-Wesley.

Altenkirch, T., C. McBride & J. McKinna. 2005, *Why dependent types matter*, URL http://www.e-pig.org/downloads/ydtm.pdf, unpublished.

Andrews, P. B. 1986, *An introduction to mathematical logic and type theory : to truth through proof*, Academic Press.

Apt, K. R. 1981, *Ten years of Hoare's logic : A survey—part I*, *ACM Transactions on Programming Languages and Systems*, vol. 3, n° 4, p. 431–483. URL http://doi.acm.org/10.1145/357146.357150.

Ayache, N. & J.-C. Filliâtre. 2006, *Combining the Coq Proof Assistant with First-Order Decision Procedures*, URL http://www.lri.fr/~filliatr/publis/coq-dp.ps.gz, unpublished.

Aydemir, B. E., A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich & S. Zdancewic. 2005, *Mechanized metatheory for the masses : The* PoplMark *challenge*, *in International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science, Springer Verlag. URL http://www.cis.upenn.edu/proj/plclub/mmm/poplmark/poplmark-conf.pdf.

Baader, F. & T. Nipkow. 1998, *Term rewriting and all that*, Cambridge University Press, New York, NY, USA, ISBN 0-521-45520-0.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden & M. Woodger. 1963, *Revised report on the algorithm language algol 60*, *Commun. ACM*, vol. 6, n° 1, doi :http://doi.acm.org/10.1145/366193.366201, p. 1–17, ISSN 0001-0782.

Barnett, M., K. R. M. Leino & W. Schulte. 2004a, *The Spec# programming system : An overview*, *in International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, *Lecture Notes in Computer Science*, vol. 3362, Springer Verlag. URL http://research.microsoft.com/specsharp/papers/krml136.pdf.

Barnett, M., D. A. Naumann, W. Schulte & Q. Sun. 2004b, *99.44% pure : Useful abstractions in specifications, in Formal Techniques for Java-like Programs*. URL http://www.cs.ru.nl/ftfjp/2004/Purity.pdf.

Bartels, F., F. von Henke, H. Pfeifer & H. Rueß. 1996, *Mechanizing domain theory*, Ulmer Informatik-Berichte 96-10, Universität Ulm, Fakultät für Informatik. URL http://www.csl.sri.com/users/ruess/papers/Fixpoints/fixpoints-domains3.ps.gz.

Beckert, B., R. Hähnle & P. H. Schmitt, . 2007, *Verification of Object-Oriented Software : The KeY Approach*, LNCS 4334, Springer-Verlag.

Benzaken, V., G. Castagna & A. Frisch. 2003, *Cduce : an xml-centric general-purpose language, in ICFP '03 : Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ACM Press, New York, NY, USA, ISBN 1-58113-756-7, p. 51–63, doi :http://doi.acm.org/10.1145/944705.944711.

de Bruijn, N. G. 1972, *Lambda-calculus notation with nameless dummies : a tool for automatic formula manipulation with application to the Church-Rosser theorem*, *Indag. Math.*, vol. 34, n° 5, p. 381–392.

Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino & E. Poll. 2005, *An overview of JML tools and applications*, *International Journal on Software Tools for Technology Transfer*, vol. 7, n° 3, p. 212–232. URL ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf.

Chen, C. & H. Xi. 2005, *Combining programming with theorem proving, in ACM International Conference on Functional Programming (ICFP)*. URL http://www.cs.bu.edu/~hwxi/academic/papers/icfp05.pdf.

Cheney, J. & R. Hinze. 2003, *First-class phantom types*, rapport technique 1901, Cornell University. URL http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901.

Clarke, D. G., J. M. Potter & J. Noble. 1998, *Ownership types for flexible alias protection, in ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, p. 48–64. URL http://doi.acm.org/10.1145/286936.286947.

Clarke, E. 1979, *Programming language constructs for which it is impossible to obtain good Hoare axiom systems*, *Journal of the ACM*, vol. 26, n° 1, p. 129–147. URL http://doi.acm.org/10.1145/322108.322121.

Conchon, S. & E. Contejean. 2006, *The Ergo automatic theorem prover*, URL http://ergo.lri.fr/, http://ergo.lri.fr/.

Conchon, S. & J.-C. Filliâtre. 2006, *Type-Safe Modular Hash-Consing, in ACM SIGPLAN Workshop on ML*, Portland, Oregon. URL http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.ps.gz.

Conchon, S. & J.-C. Filliâtre. 2007, *A Persistent Union-Find Data Structure, in ACM SIGPLAN Workshop on ML*, Freiburg, Germany. URL http://www.lri.fr/~filliatr/ftp/publis/puf-wml07.ps.

Coquand, T., R. Pollack & M. Takeyama. 2003, *A logical framework with dependently typed records, in Typed Lambda Calculus and Applications, TLCA'03, LNCS*, vol. 2701, Springer-Verlag.

Cormen, T. H., C. E. Leiserson, R. L. Rivest & C. Stein. 2001, *Introduction to Algorithms, Second Edition*, The MIT Press, ISBN 0262032937. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{%\&}path=ASIN/0262032937.

Couchot, J.-F. & S. Lescuyer. 2007, *Handling polymorphism in automated deduction*, *in CADE*, *Lecture Notes in Computer Science*, vol. 4603, F. Pfenning, Springer, ISBN 978-3-540-73594-6, p. 263–278.

Courant, J. 1997, *A Module Calculus for Pure Type Systems*, *in Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, Springer-Verlag, p. 112 – 128.

Cousot, P. 1990, *Methods and logics for proving programs*, *in Formal Models and Semantics*, *Handbook of Theoretical Computer Science*, vol. B, chap. 15, Elsevier Science, p. 841–993. URL http://www.di.ens.fr/~cousot/publications.www/Cousot-HTCS-vB-FMS-c15-p843--993-1990.pdf.gz.

Crary, K. & S. Weirich. 1999, *Flexible type analysis*, *in ACM International Conference on Functional Programming (ICFP)*, p. 233–248. URL http://www-2.cs.cmu.edu/~crary/papers/1999/lx/lx.ps.gz.

Crary, K., S. Weirich & G. Morrisett. 2002, *Intensional polymorphism in type erasure semantics*, *Journal of Functional Programming*, vol. 12, n° 6, p. 567–600. URL http://www-2.cs.cmu.edu/~crary/papers/2002/typepass/typepass.ps.

Damas, L. & R. Milner. 1982, *Principal type-schemes for functional programs*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 207–212. URL http://doi.acm.org/10.1145/582153.582176.

Damm, W. & B. Josko. 1983, *A sound and relatively* complete axiomatization of Clarke's language L4*, *in Logic of Programs*, *Lecture Notes in Computer Science*, vol. 164, Springer Verlag, p. 161–175.

Davies, R. 2005, *Practical refinement-type checking*, rapport technique CMU-CS-05-110, School of Computer Science, Carnegie Mellon University. URL http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-110.pdf.

Detlefs, D., G. Nelson & J. B. Saxe. 2005, *Simplify : a theorem prover for program checking*, *Journal of the ACM*, vol. 52, n° 3, p. 365–473. URL http://doi.acm.org/10.1145/1066100.1066102.

Detlefs, D. L., K. R. M. Leino & G. Nelson. 1998a, *Wrestling with rep exposure*, Research Report 156, SRC. URL http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-156.pdf.

Detlefs, D. L., K. R. M. Leino, G. Nelson & J. B. Saxe. 1998b, *Extended static checking*, Research Report 159, Compaq SRC. URL ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-159.pdf.

Donnelly, K. & H. Xi. 2005, *Combining higher-order abstract syntax with first-order abstract syntax in ATS*, *in ACM Workshop on Mechanized Reasoning about Languages with Variable Binding*, p. 58–63. URL http://www.cs.bu.edu/~hwxi/academic/papers/merlin05.pdf.

Dowek, G., T. Hardin & C. Kirchner. 2001, *HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic*, *Mathematical Structures in Computer Science*, vol. 11, n° 1, p. 1–25.

Dowek, G., T. Hardin, C. Kirchner & F. Pfenning. 1996, *Unification via explicit substitutions : The case of higher-order patterns*, *in Proceedings of the Joint International Conference and Symposium on Logic Programming*, M. Maher, MIT Press, Bonn, Germany, p. 259–273. URL citeseer.ist.psu.edu/article/dowek98unification.html.

Filliâtre, J.-C. 2006, *Backtracking iterators*, *in ACM Workshop on ML*. URL http://www.lri.fr/~filliatr/publis/enum2.ps.gz.

Filliâtre, J.-C. 2003, *Why : a multi-language multi-prover verification tool*, Research Report 1366, LRI, Université Paris Sud. URL http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz.

Filliâtre, J.-C. & C. Marché. 2004, *Multi-prover verification of C programs*, *in International Conference on Formal Engineering Methods (ICFEM)*, *Lecture Notes in Computer Science*, vol. 3308, Springer Verlag, p. 15–29. URL http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz.

Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe & R. Stata. 2002, *Extended static checking for Java*, *in ACM Conference on Programming Language Design and Implementation (PLDI)*, p. 234–245. URL http://www.soe.ucsc.edu/~cormac/papers/pldi02.ps.

Flanagan, C., A. Sabry, B. F. Duba & M. Felleisen. 1993, *The essence of compiling with continuations*, *in ACM Conference on Programming Language Design and Implementation (PLDI)*, p. 237–247. URL http://www.cs.rice.edu/CS/PLT/Publications/Scheme/pldi93-fsdf.ps.gz.

Flanagan, C. & J. B. Saxe. 2001, *Avoiding exponential explosion : generating compact verification conditions*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 193–205. URL http://www.soe.ucsc.edu/~cormac/papers/popl01.ps.

Floyd, R. W. 1967, *Assigning meanings to programs*, *in Mathematical Aspects of Computer Science*, *Proceedings of Symposia in Applied Mathematics*, vol. 19, American Mathematical Society, p. 19–32.

Fluet, M. & R. Pucella. 2002, *Phantom types and subtyping*, *in IFIP International Conference on Theoretical Computer Science (TCS)*, *IFIP Conference Proceedings*, vol. 223, Kluwer, p. 448–460. URL http://www.cs.cornell.edu/people/fluet/phantom-subtyping/TCS02/tcs02.ps.

Fogarty, S., E. Pasalic, J. Siek & W. Taha. 2007, *Concoqtion : indexed types now !*, *in PEPM '07 : Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ACM Press, New York, NY, USA, ISBN 978-1-59593-620-2, p. 112–121, doi :http://doi.acm.org/10.1145/1244381.1244400.

Frisch, A. 2006, *Ocamlduce.*, *in PLAN-X*, G. Castagna & M. Raghavachari, BRICS, Department of Computer Science, University of Aarhus, p. 89. URL http://dblp.uni-trier.de/db/conf/planx/planX2006.html#Frisch06a.

Fähndrich, M. & R. DeLine. 2002, *Adoption and focus : practical linear types for imperative programming*, *in ACM Conference on Programming Language Design and Implementation (PLDI)*, p. 13–24. URL http://research.microsoft.com/~maf/Papers/pldi02.pdf.

German, S., E. Clarke & J. Halpern. 1983, *Reasoning about procedures as parameters*, *in Logic of Programs*, *Lecture Notes in Computer Science*, vol. 164, Springer Verlag, p. 206–220.

Goerdt, A. 1985, *A Hoare calculus for functions defined by recursion on higher types*, *in Logic of Programs*, *Lecture Notes in Computer Science*, vol. 193, Springer Verlag, p. 106–117.

Gronski, J., K. Knowles, A. Tomb, S. N. Freund & C. Flanagan. 2006, *Sage : Hybrid checking for flexible specifications*, *in Scheme and Functional Programming*. URL http://www.cs.williams.edu/~freund/papers/06-sfp.pdf.

Hallgren, T., J. Hook, M. P. Jones & R. Kieburtz. 2004, *An overview of the Programatica toolset*, High Confidence Software and Systems Conference (HCSS). URL http://ogi.altocumulus.org/~hallgren/Programatica/HCSS04/hcss04-tools.pdf.

Henglein, F. 1993, *Type inference with polymorphic recursion*, *ACM Transactions on Programming Languages and Systems*, vol. 15, n° 2, p. 253–289. URL http://doi.acm.org/10.1145/169701.169692.

Hinze, R. 2003, *Fun with phantom types*, *in The Fun of Programming*, J. Gibbons & O. de Moor, Palgrave Macmillan, p. 245–262. URL http://www.informatik.uni-bonn.de/~ralf/publications/With.pdf.

Hoare, C. A. R. 1969, *An axiomatic basis for computer programming*, *Communications of the ACM*, vol. 12, n° 10, p. 576–580. URL http://doi.acm.org/10.1145/363235.363259.

Honda, K. & N. Yoshida. 2004, *A compositional logic for polymorphic higher-order functions*, *in International ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, p. 191–202. URL http://www.dcs.qmul.ac.uk/~kohei/logics/polyrec.pdf.gz.

Horspool, R. N. & M. Whitney. 1990, *Even faster LR parsing*, *Software – Practice & Experience*, vol. 20, n° 6, p. 515–535. URL http://www.cs.uvic.ca/~nigelh/Publications/fastparse.pdf.

Huet, G. 1976, *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω., thèse de doctorat, Université Paris 7.

Huet, G. 1997, *The zipper*, *J. Funct. Program.*, vol. 7, n° 5, doi :http://dx.doi.org/10.1017/S0956796897002864, p. 549–554, ISSN 0956-7968.

Hughes, J. 1989, *Why functional programming matters*, *Computer Journal*, vol. 32, n° 2, p. 98–107. URL http://www.math.chalmers.se/~rjmh/Papers/whyfp.pdf.

Jackson, D. 2006, *Software Abstractions : Logic, Language, and Analysis*, MIT Press, Cambridge, Mass.

Jackson, D. & C. A. Damon. 1996, *Elements of style : Analyzing a software design feature with a counterexample detector*, vol. 22, n° 7, p. 484–495.

Jia, L., F. Spalding, D. Walker & N. Glew. 2005, *Certifying compilation for a language with stack allocation*, *in IEEE Symposium on Logic in Computer Science (LICS)*, p. 407–416. URL http://www.cs.princeton.edu/~dpw/papers/stackcert-lics05.pdf.

Johnson, S. C. 1975, *Yacc : Yet another compiler-compiler*, Computing Science Technical Report 32, Bell Laboratories. URL http://dinosaur.compilertools.net/yacc/yacc.ps.

Kahn, G. 1987, *Natural semantics*, *in STACS '87 : Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag, London, UK, ISBN 3-540-17219-X, p. 22–39.

Kaplan, H. & R. E. Tarjan. 1999, *Purely functional, real-time deques with catenation*, *Journal of the ACM*, vol. 46, n° 5, p. 577–603. URL http://www.math.tau.ac.il/~haimk/bob.ps.

Kennedy, A. & C. V. Russo. 2005, *Generalized algebraic data types and object-oriented programming*, *in OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, ACM Press, New York, NY, USA, ISBN 1-59593-031-0, p. 21–40, doi :http://doi.acm.org/10.1145/1094811.1094814.

Kerber, M. 1991, *How to prove higher order theorems in first order logic*, *in International Joint Conferences on Artificial Intelligence*, p. 137–142. URL ftp://ftp.cs.bham.ac.uk/pub/authors/M.Kerber/91-IJCAI.pdf.

Kieburtz, R. B. 2002, *P-logic : Property verification for Haskell programs*, URL http://www.cse.ogi.edu/PacSoft/projects/programatica/plogic.pdf, draft.

Knuth, D. E. 1965, *On the translation of languages from left to right*, *Information & Control*, vol. 8, n° 6, p. 607–639.

Läufer, K. & M. Odersky. 1994, *Polymorphic type inference and abstract data types*, *ACM Transactions on Programming Languages and Systems*, vol. 16, n° 5, p. 1411–1430. URL http://www.cs.luc.edu/laufer/papers/toplas94.pdf.

Le Botlan, D. & D. Rémy. 2003, *MLF : Raising ML to the power of system F*, *in ACM International Conference on Functional Programming (ICFP)*, p. 27–38. URL http://cristal.inria.fr/~remy/work/mlf/icfp.pdf.

Leino, K. R. M. & G. Nelson. 2002, *Data abstraction and information hiding*, *ACM Transactions on Programming Languages and Systems*, vol. 24, n° 5, p. 491–553. URL http://research.microsoft.com/~leino/papers/krml71.pdf.

Leroy, X. 2006, *Formal certification of a compiler back-end or : programming a compiler with a proof assistant*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 42–54. URL http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf.

Leroy, X., D. Doligez, J. Garrigue, D. Rémy & J. Vouillon. 2004, *The Objective Caml system*. URL http://cristal.inria.fr/ocaml/htmlman/.

Lescuyer, S. 2006, *Codage de la logique du premier ordre polymorphe multi-sortée dans la logique sans sortes*, mémoire de master, Master Parisien de Recherche en Informatique. URL http://www.seas.upenn.edu/~lescuyer/pdf/RapportDEA.pdf.

Letouzey, P. 2004, *Programmation fonctionnelle certifiée – l'extraction de programmes dans l'assistant Coq*, thèse de doctorat, Université Paris 11. URL http://www.lri.fr/~letouzey/download/these_letouzey.ps.gz.

Longley, J. & R. Pollack. 2004, *Reasoning about CBV functional programs in Isabelle/HOL*, *in International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, *Lecture Notes in Computer Science*, vol. 3223, Springer Verlag, p. 201–216. URL http://homepages.inf.ed.ac.uk/rpollack/export/LongleyPollack04.pdf.

Marché, C., C. Paulin-Mohring & X. Urbain. 2004, *The Krakatoa tool for certification of Java/JavaCard programs annotated in JML*, *Journal of Logic and Algebraic Programming*, vol. 58, n° 1–2, p. 89–106. URL http://www3.ensiie.fr/~urbain/textes/jlap.ps.gz.

Marlow, S. & A. Gill. 2004, *Happy : the parser generator for Haskell*. URL http://www.haskell.org/happy/.

Martelli, A. & U. Montanari. 1982, *An efficient unification algorithm*, *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 2, p. 258–282. URL http://doi.acm.org/10.1145/357162.357169.

Mehta, F. & T. Nipkow. 2005, *Proving pointer programs in higher-order logic*, *Information and Computation*, vol. 199, n° 1–2, p. 200–227. URL http://www4.informatik.tu-muenchen.de/~nipkow/pubs/ic05.ps.gz.

Miller, D. 1992, *Unification under a mixed prefix*, vol. 14, n° 4, p. 321–358, ISSN 0747-7171.

Milner, R. 1972, *Implementation and applications of Scott's logic for computable functions*, *in Proceedings of the ACM conference on proving assertions about programs*, p. 1–6. URL http://doi.acm.org/10.1145/800235.807067.

Milner, R. 1978, *A theory of type polymorphism in programming*, *Journal of Computer and System Sciences*, vol. 17, n° 3, p. 348–375.

Milner, R., M. Tofte, R. Harper & D. MacQueen. 1997, *The Definition of Standard ML – Revised*, MIT Press.

Monnier, S. & L.-J. Guillemette. 2007, *A type preserving closure conversion in haskell*, *in ACM Workshop on Haskell*.

Müller, O., T. Nipkow, D. von Oheimb & O. Slotosch. 1999, *HOLCF = HOL + LCF*, *Journal of Functional Programming*, vol. 9, p. 191–223. URL http://www4.informatik.tu-muenchen.de/~nipkow/pubs/jfp99.ps.gz.

Nanevski, A., A. Ahmed, G. Morrisett & L. Birkedal. 2007, *Abstract predicates and mutable ADTs in Hoare type theory*, *in European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, Springer Verlag. URL http://www.eecs.harvard.edu/~aleks/papers/hoarelogic/esop07.pdf.

Nanevski, A., G. Morrisett & L. Birkedal. 2006, *Polymorphism and separation in Hoare type theory*, *in ACM International Conference on Functional Programming (ICFP)*, p. 62–73. URL http://www.eecs.harvard.edu/~aleks/papers/hoarelogic/icfp06.pdf.

Nelson, G. & D. C. Oppen. 1980, *Fast decision procedures based on congruence closure*, *J. ACM*, vol. 27, n° 2, doi :http://doi.acm.org/10.1145/322186.322198, p. 356–364, ISSN 0004-5411.

Nilsson, H. 2005, *Dynamic optimization for functional reactive programming using generalized algebraic data types*, *SIGPLAN Not.*, vol. 40, n° 9, doi :http://doi.acm.org/10.1145/1090189.1086374, p. 54–65, ISSN 0362-1340.

Odersky, M. & M. Zenger. 2005, *Scalable component abstractions*, *in ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, p. 41–57. URL http://lamp.epfl.ch/~odersky/papers/ScalableComponent.pdf.

Odersky, M., M. Zenger & C. Zenger. 2001, *Colored local type inference*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 41–53. URL http://lampwww.epfl.ch/papers/clti-colored.ps.gz.

Paulin-Mohring, C. 1992, *Inductive definitions in the system Coq : rules and properties*, Research Report RR1992-49, ENS Lyon. URL ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1992/RR1992-49.ps.Z.

Peyton Jones, S., . 2003, *Haskell 98 Language and Libraries : The Revised Report*, Cambridge University Press. URL http://www.haskell.org/onlinereport/.

Peyton Jones, S., G. Washburn & S. Weirich. 2004, *Wobbly types : type inference for generalised algebraic data types*, rapport technique MS-CIS-05-26, University of Pennsylvania. URL http://www.cis.upenn.edu/~geoffw/research/papers/MS-CIS-05-26.pdf.

Peyton Jones, S., G. Washburn & S. Weirich. 2004, *Wobbly types : type inference for generalised algebraic data types*, URL http://research.microsoft.com/Users/simonpj/papers/gadt/, manuscript.

Pfenning, F. & P. Lee. 1989, *LEAP : A language with eval and polymorphism*, *in International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, *Lecture Notes in Computer Science*, vol. 352, Springer Verlag, p. 345–359.

Pierce, B. C. 2002, *Types and Programming Languages*, MIT Press. URL http://www.cis.upenn.edu/~bcpierce/tapl/.

Pierce, B. C. & D. N. Turner. 2000, *Local type inference*, *ACM Transactions on Programming Languages and Systems*, vol. 22, n° 1, p. 1–44. URL http://doi.acm.org/10.1145/345099.345100.

Pitts, A. M. 2003, *Nominal logic, A first order theory of names and binding*, *Information and Computation*, vol. 186, p. 165–193. URL http://www.cl.cam.ac.uk/~amp12/papers/nomlfo/nomlfo-jv.pdf.

Pottier, F. 2005, *An overview of Cαml*, URL http://cristal.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf, submitted.

Pottier, F. 2007, *Static name control for FreshML*, *in Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, Wroclaw, Poland, p. 356–365. URL http://cristal.inria.fr/~fpottier/publis/fpottier-pure-freshml.ps.gz.

Pottier, F. & N. Gauthier. 2004, *Polymorphic typed defunctionalization*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 89–98. URL http://cristal.inria.fr/~fpottier/publis/fpottier-gauthier-popl04.pdf.

Pottier, F. & Y. Régis-Gianas. 2005, *Menhir*, URL http://cristal.inria.fr/~fpottier/menhir/.

Pottier, F. & Y. Régis-Gianas. 2006a, *Stratified type inference for generalized algebraic data types*, *in Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, p. 232–244. URL http://cristal.inria.fr/~fpottier/publis/pottier-regis-gianas-popl06.ps.gz.

Pottier, F. & Y. Régis-Gianas. 2006b, *Towards efficient, typed LR parsers*, *in ACM Workshop on ML*, *Electronic Notes in Theoretical Computer Science*, vol. 148, p. 155–180. URL http://cristal.inria.fr/~fpottier/publis/fpottier-regis-gianas-typed-lr.ps.gz.

Pottier, F. & D. Rémy. 2005, *The essence of ML type inference*, *in Advanced Topics in Types and Programming Languages*, B. C. Pierce, chap. 10, MIT Press, p. 389–489.

Pottier, F. & V. Simonet. 2003, *Information flow inference for ML*, *ACM Transactions on Programming Languages and Systems*, vol. 25, n° 1, p. 117–158. URL http://cristal.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz.

Rémy, D. 1994, *Type inference for records in a natural extension of ML*, *in Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, C. A. Gunter & J. C. Mitchell, MIT Press. URL ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz.

Reynolds, J. C. 1990, *An introduction to the polymorphic lambda calculus*, *in Logical Foundations of Functional Programming*, G. Huet, Addison-Wesley, p. 77–86. URL ftp://ftp.cs.cmu.edu/user/jcr/polyintro.ps.gz.

Reynolds, J. C. 2002, *Separation logic : A logic for shared mutable data structures*, *in IEEE Symposium on Logic in Computer Science (LICS)*, p. 55–74. URL ftp://ftp.cs.cmu.edu/user/jcr/seplogic.ps.gz.

Ringenburg, M. F. & D. Grossman. 2005, *Types for describing coordinated data structures*, *in Workshop on Types in Language Design and Implementation (TLDI)*, p. 25–36. URL http://www.cs.washington.edu/homes/miker/coord/coordinated_tldi05.pdf.

Régis-Gianas, Y. 2004, *A prototype parser generator for ML with generalized algebraic data types*, http://cristal.inria.fr/~regisgia/software/.

Régis-Gianas, Y. 2005, *A prototype typechecker for ML with generalized algebraic data types*, http://cristal.inria.fr/~regisgia/software/.

Sheard, T. 2004, *Playing with type systems*, .

Sheard, T. 2005a, Ω*mega*. URL http://www.cs.pdx.edu/~sheard/Omega/.

Sheard, T. 2005b, *Putting Curry-Howard to work*, *in Haskell workshop*. URL http://web.cecs.pdx.edu/~sheard/papers/PutCurryHoward2WorkFinalVersion.ps.

Simonet, V. & F. Pottier. 2005, *Constraint-based type inference for guarded algebraic data types*, Research Report 5462, INRIA. URL http://www.inria.fr/rrrt/rr-5462.html.

Simonet, V. & F. Pottier. 2007, *A constraint-based approach to guarded algebraic data types*, *ACM Transactions on Programming Languages and Systems*, vol. 29, n° 1. URL http://cristal.inria.fr/~fpottier/publis/simonet-pottier-hmg-toplas.ps.gz.

Sozeau, M. 2006, *Subset coercions in Coq*, *in TYPES*. URL http://www.lri.fr/~sozeau/research/russell/article.pdf.

Spivey, M. 1990, *A functional theory of exceptions*, *Science of Computer Programming*, vol. 14, p. 25–42.

Stuckey, P. J. & M. Sulzmann. 2005, *Type inference for guarded recursive data types*, URL http://www.comp.nus.edu.sg/~sulzmann/icalp05.ps.gz, manuscript.

Sulzmann, M., J. Wazny & P. J. Stuckey. 2006, *A framework for extended algebraic data types*, *in Proc. of FLOPS'06*, *LNCS*, vol. 3945, Springer-Verlag, p. 47–64.

Tarditi, D. R. & A. W. Appel. 2000, *ML-Yacc User's Manual*. URL http://www.smlnj.org/doc/ML-Yacc/.

The Coq development team. 2006, *The Coq Proof Assistant*. URL http://coq.inria.fr/.

Tofte, M. & J.-P. Talpin. 1994, *Implementation of the typed call-by-value λ-calculus using a stack of regions*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 188–201. URL http://www.diku.dk/users/tofte/publ/popl94.dvi.gz.

Urban, C., A. Pitts & M. Gabbay. 2004, *Nominal unification*, *Theoretical Computer Science*, vol. 323, p. 473–497. URL http://www.cl.cam.ac.uk/~cu200/Unification/nomu-tcs.ps.

Vytiniotis, D., S. Weirich & S. Peyton Jones. 2005, *Boxy types : type inference for higher-rank types and impredicativity*, URL http://research.microsoft.com/Users/simonpj/papers/boxy/, manuscript.

Westbrook, E., A. Stump & I. Wehrman. 2005, *A language-based approach to functionally correct imperative programming*, *in ACM International Conference on Functional Programming (ICFP)*, p. 268–279. URL http://cl.cse.wustl.edu/papers/rsp1-icfp05.pdf.

Wright, A. K. & M. Felleisen. 1994, *A syntactic approach to type soundness*, *Information and Computation*, vol. 115, n° 1, p. 38–94. URL http://www.cs.rice.edu/CS/PLT/Publications/Scheme/ic94-wf.ps.gz.

Xi, H. 1998, *Dependent Types in Practical Programming*, thèse de doctorat, Carnegie Mellon University. URL http://www.cs.bu.edu/~hwxi/academic/papers/thesis.ps.

Xi, H. 1999, *Dead code elimination through dependent types*, *in International Workshop on Practical Aspects of Declarative Languages (PADL)*, *Lecture Notes in Computer Science*, vol. 1551, Springer Verlag, p. 228–242. URL http://www.cs.bu.edu/~hwxi/academic/papers/padl99.ps.

Xi, H. 2001, *Dependent ML*, URL http://www.cs.bu.edu/~hwxi/DML/DML.html.

Xi, H., C. Chen & G. Chen. 2003, *Guarded recursive datatype constructors*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 224–235. URL http://www.cs.bu.edu/fac/hwxi/academic/papers/popl03.ps.

Xi, H. & F. Pfenning. 1999, *Dependent types in practical programming*, *in ACM Symposium on Principles of Programming Languages (POPL)*, p. 214–227. URL http://www.ececs.uc.edu/~hwxi/academic/papers/popl99.ps.

Xu, D. N. 2006, *Extended static checking for Haskell*, *in Haskell workshop*, ACM Press, p. 48–59. URL http://www.cl.cam.ac.uk/~nx200/research/escH-hw.ps.

Zenger, C. 1997, *Indexed types*, *Theoretical Computer Science*, vol. 187, p. 147–165.

Résumé

Cette thèse étudie deux approches pour augmenter la sûreté de fonctionnement des programmes informatiques par analyse statique. La première approche, le typage, permet de prouver automatiquement qu'un programme s'évalue sans échouer. Le langage fonctionnel ML possède un système de type très riche et un algorithme effectuant leur synthèse automatique. On s'intéresse à son adaptation aux types algébriques généralisés (GADT). Dans ce cadre, le calcul efficace d'un type plus général est impossible. On propose une stratification du langage qui maintient les caractéristiques habituelles sur le fragment ML et qui isole le traitement des GADT en explicitant leur utilisation. Le langage obtenu, MLGX, nécessite des annotations de type qui alourdissent les programmes. Une seconde strate, MLGI, offre au programmeur un mécanisme de synthèse locale, prédictible et efficace bien qu'incomplet, de la plupart de ces annotations. La première partie s'achève avec une démonstration de l'expressivité des GADT pour coder les invariants des automates à pile utilisés par l'analyse syntaxique LR. La seconde approche augmente le langage par des assertions logiques permettant d'exprimer des spécifications de complexité arbitraire. On vérifie la conformité de la sémantique du programme vis-à-vis de ces spécifications à l'aide d'une technique appelée logique de Hoare et d'outils semi-automatique de preuves mathématiques sur ordinateur. Les choix de conception du système permettent de traiter les fonctions de première classe. Ils sont dirigés par une implantation qui trouve une illustration dans la certification d'un module d'arbres dénotant des ensembles finis.

Abstract

This work studies two approaches to improve the safety of computer programs using static analysis. The first one is typing which guarantees that the evaluation of program cannot fail. The functional language ML has a very rich type system and also an algorithm that infers automatically the types. We focus on its adaptation to generalized algebraic data types (GADTs). In this setting, efficient computation of a most general type is impossible. We propose a stratification of the language that retain the usual characteristics of the ML fragment and make explicit the use of GADTs. The resulting language, MLGX, entails a burden on the programmer who must annotate its programs too much. A second stratum, MLGI, offers a mechanism to infer locally, in a predictable and efficient way, incomplete yet, most of the type annotations. The first part concludes on an illustration of the expressiveness of GADTs to encode the invariants of pushdown automata used in LR parsing. The second approach augments the language with logic assertions that enables arbitrarily complex specifications to be expressed. We check the compliance of the program semantics with respect to these specifications thanks to a method called Hoare logic and thanks to semi-automatic computer-based proofs. The design choices permit to handle first-class functions. They are directed by an implementation which is illustrated by the certification of a module of trees that denote finite sets.