Mémoire de D.E.A.
Laboratoire d'Informatique
de l'École Normale Supérieure

Types récursifs, combinaisons booléennes et fonctions surchargées : application au typage de XML

Alain Frisch

sous la direction de :

Giuseppe Castagna
Véronique Benzaken

soutenu le 17 septembre 2001

Remerciements

Je tiens à remercier Mme Véronique Benzaken et Mr Giuseppe Castagna pour leur disponibilité, la confiance qu'ils m'ont accordée dans l'encadrement du stage et la patience dont ils ont fait preuve lors des multiples relectures du présent rapport.
Je voudrais également remercier toute l'équipe du LIENS dans laquelle j'ai travaillé pour son accueil chaleureux. J'ai pu découvrir la recherche en informatique dans un cadre agréable et propice au travail.

Résumé

Nous étudions un λ-calcul typé avec une opération de filtrage qui permet d'exprimer des fonctions surchargées. L'algèbre de types a des types de base, les types produit et flèche, les types récursifs, les combinaisons booléennes finies arbitraires. Nous considérons une notion de sous-typage sémantique, issue de l'interprétation des types comme ensembles de valeurs du langage.

Les caractéristiques présentes dans le calcul et l'algèbre de types sont motivées par l'utilisation du calcul comme un noyau pour un langage adapté à la manipulation de documents XML.

Nous utilisons une approche sémantique pour étudier l'algèbre de types, tout en conservant une preuve syntaxique de préservation du typage par réduction.

Table des matières

Chapitre 1

Introduction

1.1 Motivations, objectifs, et travaux existants

1.1.1 Objectifs

L'objectif initial du stage était de proposer un langage adapté à la manipulation de documents XML [XML] , et d'en étudier les propriétés formelles, essentiellement celles de typage. Il est bien entendu possible de manipuler des documents XML avec des langages classiques, mais cela peut s'avérer assez lourd. Par exemple, on peut vouloir disposer d'un mécanisme pour extraire des informations au milieu d'un document complexe ; c'est ce que propose par exemple XPath [XPA].

Plutôt que de proposer un langage dédié à XML, nous cherchons à analyser les besoins spécifiques à la manipulation de documents XML et à les intégrer à un langage généraliste : d'une part un langage qui manipule XML peut vouloir aussi faire autre chose ; d'autre part ce qui est intéressant pour XML l'est aussi pour d'autres applications. C'est en ce sens que nous parlons de langage adapté à XML.

1.1.2 Des types statiques pour XML

XML XML est un langage standard de description de documents semi-structurés. Il s'agit essentiellement d'une syntaxe concrète pour stocker sous forme textuelle des données organisées en arbre ; contrairement à HTML, la structure n'est pas imposée, et chaque application peut définir ses propres contraintes sur les documents XML manipulés.

Un fragment de document XML ressemble à :

```
<personne>
  <nom>Frisch</nom>
  <prenom>Alain</prenom>
</personne>
```

Dans cet exemple, `personne, nom, prenom` sont des *tags* ; ils sont utilisés pour délimiter des éléments. Une des conditions de bonne formation des documents XML est celle de bon parenthésage ; elle permet de voir les documents XML comme des arbres ; aux nœuds, on trouve des *tags* ou des données atomiques, comme des chaînes de caractères. Nous laisserons de coté tous les détails syntaxiques de XML (éléments vides, entités, ...), les définitions de DTD, et, dans un premier temps, également les attributs (nous y reviendrons en 5.1.1).

Des contraintes vues comme des types Une application qui manipule des documents XML impose en général des contraintes sur les documents qu'elle s'attend à recevoir : quels sont les *tags* autorisés et leurs possibles agencements, ...

Il existe plusieurs systèmes standards pour exprimer des contraintes sur les documents XML : DTD, XML-Schema [SCH], ... Un document XML est validé par une DTD ou un Schéma lorsqu'il en satisfait les contraintes. Il est naturel de voir ce mécanisme de validation comme du typage ; un type est une contrainte exprimée dans un certain système et on peut donc aussi l'interpréter de manière ensembliste comme l'ensemble des documents qui satisfont cette contrainte. Avec ce point de vue, DTD et XML-Schema sont des systèmes de types pour les documents.

Remarque 1. *Les types ont plusieurs interprétations classiques, qui correspondent à des points de vue assez différents :*

- *Avec les isomorphismes de Curry-Howard, on voit les types comme des propositions dans une logique, et on étudie un λ-calcul dont les termes sont des preuves des propositions ; un terme a le type t si c'est une preuve de t. Le calcul correspond à la mise en forme normale des preuves (élimination des coupures).*
- *Dans les langages du style ML, on voit les types comme des abstractions du comportement ou de la forme des objets. On utilise le typage comme analyse statique pour assurer des résultats de sûreté des programmes. [Cou97] montre comment interpréter ce genre de systèmes de types dans le formalisme de l'interprétation abstraite. Avec ce point de vue, les types ne peuvent pas intervenir dans les calculs (la sémantique est non typée).*
- *Dans les langages du style C, les types sont des informations données par le programmeur pour définir la représentation concrète des données ; on peut en général utiliser une donnée d'un certain type comme un objet d'un autre type de même taille physique (ou même d'une taille différente).*
- *Dans le cadre de la validation de documents XML, on voit les types comme des contraintes sur la structure et le contenu des documents. Un type définit donc un ensemble de documents, et il faut le voir dans ce sens. La question «Quel est le type d'un document ?» n'a pas beaucoup de sens. Ce qui compte, c'est la question : «Ce document a-t-il oui ou non ce type là ?». Ce point de vue sur les types est proche de celui des bases de données : le schéma d'une base décrit ses états admissibles ; il contraint les données localement et globalement, et permet d'optimiser les requêtes.*

Essentiellement, les contraintes exprimées caractérisent des ensembles réguliers d'arbres. Dans une DTD, à chaque *tag*, on associe un modèle de contenu, décrit par une expression régulière. Avec XML-Schema, le modèle du contenu ne dépend pas que du *tag*, mais aussi de la position de l'élément dans l'arbre ; cela donne des contraintes plus fines.

1.1.3 Programmer avec XML

Les applications travaillent donc avec des documents d'un certain type, par exemple pour en extraire de l'information (requêtes dans des bases de données), ou pour les transformer en des documents d'un autre type (par exemple pour les présenter en XHTML). Il y a plusieurs manières d'écrire des applications qui manipulent des documents XML.

La plus directe est de travailler avec un langage classique, et de considérer les documents comme de simples fichiers textes. Il faut alors se charger d'analyser les documents, et éventuellement d'en générer ; bien entendu, il n'y a aucune interaction avec le système de type XML, et rien ne garantit que les documents produits sont même seulement bien formés.

À un niveau supérieur, on utilise un *parser* XML, qui se charge de lire un document XML et de le représenter en mémoire sous forme structurée (un arbre d'éléments). On peut travailler directement sur cet arbre, le modifier en place, l'écrire de nouveau dans un fichier XML. Il existe même des standards (DOM [DOM]) pour spécifier les interfaces d'accès et de modification des documents en mémoire. On parle de *parser* validant lorsqu'il vérifie que le document analysé a le bon type ; par contre, une fois le document analysé, on ne peut plus utiliser l'information sur le type du document pour assurer certaines propriétés, ni prouver que le document, après transformation, est du type attendu. La seule garantie que l'on a est que le document produit est bien formé.

Il y a des langages spécialisés dans la transformation de documents XML, comme XSLT [XSL], qui n'ont aucune interaction avec les systèmes de types. Ils garantissent juste que les documents produits sont bien formés, mais il semble envisageable d'ajouter des systèmes de types à ce genre de langage pour assurer statiquement que les documents produits sont du bon type.

Il serait agréable d'avoir un langage qui tire parti du typage statique des documents XML. On voudrait par exemple garantir qu'un programme de transformation a bien le type $t \rightarrow s$, où t et s sont deux types XML ; cela signifie que si on lui donne un document validé par t, le résultat produit sera nécessairement validé par s. Pour cela, il faut que le langage, en particulier son système de types et son modèle de calcul, ait été conçu en gardant en tête la spécificité des types XML.

Quand on veut typer des fonctions dans un langage assez expressif, il se pose rapidement des problèmes de non-décidabilité. Avec des modèles de calcul assez peu expressifs, comme [MSV00], on peut effectivement espérer un typage précis des transformations. Sinon, il faut se contenter d'approximations, en demandant par exemple au programmeur de donner un type, et de regarder si l'on peut prouver dans une certaine logique (le système de types) que la fonction a bien ce type.

1.1.4 XDuce

Le langage XDuce [Hos01, HP00, HVP00] propose :

- Un codage interne [HVP00] (voir la section suivante) des documents XML qui permet d'attaquer les problèmes de typage avec des méthodes issues de l'algorithmique des langages d'arbre réguliers.
- Un système de types du genre DTD, mais dans lequel le contenu d'un élément n'est pas imposé uniquement par son *tag* mais aussi par sa position,

et sans la restriction de déterminisme des DTD.
- Une interprétation des types comme des ensembles de documents, qui mène naturellement à définir le sous-typage $\leq$ comme l'inclusion des ensembles dénotés (on parle de sous-typage sémantique).
- Une opération de filtrage (*pattern-matching*) [HP01], avec une algèbre de motifs adaptée à l'extraction d'information de documents XML.

Un programme XDuce est essentiellement un ensemble de définitions de fonctions. Les fonctions sont définies par cas, en utilisant le filtrage. Tous les objets manipulés sont des documents XML (ou, plus précisément, des séquences d'éléments). Il se pose naturellement le problème de l'intégration du système de types dans un langage de programmation plus complet, qui puisse manipuler des objets de différentes natures, et en particulier des fonctions de premier ordre. Les langages fonctionnels semblent très adaptés à la manipulation de documents XML ; souvent, les transformations XML se font sans effet de bord. En général, les langages fonctionnels permettent d'aborder plus facilement les problèmes de typage que les langages impératifs (par exemple, on pourrait essayer de typer un langage qui s'appuie sur DOM pour transformer l'arbre XML, mais le problème semble difficile). La présence de l'opération de filtrage comme brique pour définir les fonctions dans XDuce montre une ressemblance avec les langages de type ML.

Hosoya [Hos01] signale la difficulté d'ajouter le constructeur $\rightarrow$ à l'algèbre de types de XDuce. La difficulté vient de la relation de sous-typage pour les types fonctionnels. Pour garder l'interprétation des types comme contraintes qui définissent des ensembles d'objets, on veut voir le type $t \rightarrow s$ comme l'ensemble des fonctions qui, appliquées à un objet de type t, ne peuvent pas renvoyer autre chose qu'un objet de type s (il peut y avoir plusieurs résultat possibles en présence de non déterminisme ou d'effets de bords). Le problème est que cette contrainte n'est pas décidable pour un langage Turing-complet. La première motivation de notre travail a été de voir comment résoudre ce problème.

Donnons un exemple simple où l'utilisation de fonctions du premier ordre peut éviter de dupliquer inutilement du code. Supposons que le type `Addrbook` représente les documents constitués d'une suite d'éléments de type `Person` et que `Html` représente le type des fragments de document XHTML. On peut alors écrire une fonction `showAddrBook`, de type `AddrBook` $\rightarrow$ (`Person` $\rightarrow$ `Html`) $\rightarrow$ `Html` qui formate un carnet d'adresse avec une présentation des entrées donnée en argument. On voit que ce genre de transformation est en fait assez générique, et on peut chercher à introduire un polymorphisme paramétrique à la ML. Nous nous orientons plutôt dans ce travail vers un polymorphisme *ad hoc* à la λ&-calcul ([CGL95]). Nous montrerons par la suite (section 5.2) comment ajouter une forme rudimentaire, mais probablement suffisante en pratique, de polymorphisme paramétrique.

1.1.5 Codage interne

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

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

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

Par exemple, le document :

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

est codé en forme interne par :

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

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

$$\begin{array}{lcll} \texttt{T} & ::= & & \\ & & () & \text{séquence vide} \\ & \mid & \texttt{X} & \text{type déclaré} \\ & \mid & \texttt{l[T]} & \textit{tag} \\ & \mid & \texttt{T,T} & \text{concaténation} \\ & \mid & \texttt{T|T} & \text{réunion} \end{array}$$

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

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

L'interprétation de ces types comme ensembles de séquences est claire.

Pour la forme interne, on utilise des automates d'arbres binaires. Notons $X, X_0, X_1, \ldots$ les états, et supposons donnée pour chacun la description $M(X)$ de ses transitions, qui est dans la syntaxe :

$$\begin{array}{lcll} T & ::= & & \\ & & \varnothing & \text{type vide} \\ & \mid & \epsilon & \text{type singleton de la séquence vide} \\ & \mid & l(X, X) & \textit{tag} \\ & \mid & T|T & \text{réunion} \end{array}$$

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

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

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

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

(dans une DTD, le *tag* prescrit le contenu de l'élément ; avec les types de XDuce, le même *tag* peut apparaître à plusieurs endroits d'un même type avec des contenus différents)

La forme interne des documents et des types de XDuce est assez spécifique à XML, où les *tags* jouent un rôle particulier. En fait, la forme interne n'utilise que des caractéristiques bien connues en théorie des types : types de base singletons (pour les *tags* et la séquence vide), types produit (pour la structure d'arbre), types récursifs et types réunion (pour coder les expressions régulières). Pour représenter un élément de *tag* l, et de contenu v_1, on utilise par exemple la notation : (l, v_1). Pour représenter une séquence qui commence par un tel élément, suivi de la séquence représentée par v_2, on prendra : $((l, v_1), v_2)$. Le type donné en exemple ci-dessus sera écrit :

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

où $\mathtt{person}, \mathtt{name}, \mathtt{email}, \epsilon$ désignent des types singletons. Pour tenir compte du *subtagging* de XDuce, on utilise simplement des types qui contiennent plusieurs *tags*. Nous n'allons pas plus formaliser la traduction des documents XML dans la représentation de XDuce ou dans la nôtre.

Remarque 2. *On peut évidemment prendre* $(l, (v_1, v_2))$ *au lieu de* $((l, v_1), v_2)$*, pour être plus proche de XDuce. L'avantage de prendre* $((l, v_1), v_2)$*, c'est que l'on peut extraire l'élément* (l, v_1) *de la séquence sans reconstruire le couple.*

1.1.6 Les fonctions surchargées ; le $\lambda\&$ calcul

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

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

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

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

Remarque 3. *Avec le filtrage ou avec des fonctions surchargées, les calculs sont dirigés par les types. Il faut donc que la structure des objets lors de l'exécution permette encore de répondre à la question : « cette valeur a-t-elle oui ou non le type t ? ». Pour les documents XML, c'est toujours le cas, puisqu'en regardant le document, on peut dire s'il a un type donné. Autrement dit, et contrairement au type* dynamic *[ACPP91, ACP+92, LM91] que l'on peut rajouter à ML, on ne garde pas pendant l'exécution à coté des objets leur type ; c'est leur représentation même qui doit permettre de dire s'ils ont un type donné. Il faut faire attention pour les objets fonctions : en général, même dans les langages où on précise le type des fonctions dans les programmes (C par exemple), il n'y en a plus trace à l'exécution. Nous aurons besoin de garder cette information.*

Nous nous proposons d'ajouter à XDuce directement des fonctions surchargées. Il n'y a pas de nouveau constructeur de type ; on introduit juste les types intersection. Le type $(t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)$ dénote les fonctions qui ont simultanément tous les types $t_i \to s_i$, autrement dit ce sont des fonctions surchargées. Le type intersection s'interprète effectivement comme une conjonction de contraintes (donc comme l'intersection des ensembles dénotés).

Donnons un exemple très simple d'utilisation des fonctions surchargées. Supposons qu'une entreprise reçoive de ses clients des documents XML qui sont des suites de commandes, et qu'il y a deux types de commandes t_1, t_2 (par exemple, pour des types de produits différents). Elle renvoie un accusé de réception qui reprend chaque commande, et, y ajoute des informations suivant le type. À une commande de type t_i, elle répond par un fragment XML de type s_i. Sans fonction surchargée, la fonction de traitement a le type $(t_1 \vee t_2)* \to (s_1 \vee s_2)*$ (on utilise ici librement l'étoile de Kleene des expressions régulières). Elle procède en itérant simplement une fonction de réponse pour les commandes individuelles, de type $(t_1 \vee t_2) \to (s_1 \vee s_2)$. En fait, c'est une fonction surchargée, qui a le type : $(t_1 \to s_1) \wedge (t_2 \to s_2)$, et on peut donc donner à la transformation globale (sans récrire son code), le type plus précis : $((t_1 \vee t_2)* \to (s_1 \vee s_2)*) \wedge (t_1* \to s_1*) \wedge (t_2* \to s_2*)$. Ainsi, si un client n'envoie que des commandes de type t_1, donc un bon de commande de type t_1*, il peut supposer que ce qu'il reçoit est de type s_1* ; un client qui envoie des commandes des deux types doit pouvoir recevoir des réponses $(s_1 \vee s_2)*$. Si l'on ajoute un nouveau type de commandes, il suffit de modifier le code la fonction de traitement des commandes pour qu'elle puisse traiter un nouveau type (on change également l'interface des fonctions pour préciser les nouveaux types gérés).

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

Pour implémenter le corps d'une fonction surchargée, on utilise l'opération de filtrage, avec le même mécanisme de sélection que dans XDuce (on essaie successivement chaque branche).

1.1.7 Combinaisons booléennes

Pour exprimer les types XML, qui sont des langages d'arbre réguliers, nous avons vu qu'il suffit d'avoir des types de base, des types produit, réunion et récursifs. Pour les fonctions surchargées, on a besoin du constructeur $\rightarrow$ et des types intersection. Il faut donc pouvoir exprimer dans l'algèbre de types les combinaisons booléennes. Motivons encore ce besoin par le choix de l'opération de filtrage comme primitive (et donc devant être typée précisément) et l'utilisation de XML dans le monde des bases de données.

Combinaison booléennes et filtrage Lorsqu'un filtrage est défini par plusieurs branches, pour trouver le type précis des variables capturées par les différentes branches, il faut être capable d'exprimer que si un motif p échoue, l'objet filtré n'est pas dans le type accepté $\wr p \int$ par le motif de cette branche. Si on sait par le contexte que l'objet filtré est de type t, on peut supposer pour typer la branche suivante qu'il a en fait le type $t \wedge \neg \wr p \int$ où $\neg$ désigne le complémentaire (c'est-à-dire $t \backslash \wr p \int$ en notant $\backslash$ la différence). Lorsque l'on connaît le type du résultat de chaque branche, il faut attribuer un type au résultat du filtrage globalement : c'est naturellement la réunion des types de chaque branche.

Pour savoir si un filtrage est exhaustif, ce qui est nécessaire pour empêcher les erreurs à l'exécution, on doit décider si la réunion des types acceptés par chaque branche recouvre bien le type de l'objet filtré : $t \leq \wr p_1 \int \vee \ldots \vee \wr p_n \int$

Les combinaisons booléennes arbitraires (finies) sont nécessaires pour typer précisément l'opération de filtrage.

Combinaison booléennes et intégration de bases de données Si un même programme veut pouvoir traiter des documents provenant de sources différentes, qui peuvent avoir des types différents, il doit savoir traiter la réunion de ces types.

Réciproquement, si on veut envoyer un document à plusieurs programmes qui ont des contraintes un peu différentes sur leur type d'entrée, il faut s'assurer qu'il a le type intersection des différents types acceptés.

Les opérations booléennes sont très utiles lorsqu'il s'agit de mélanger des types de documents hétérogènes.

Les combinaisons booléennes dans XDuce Dans XDuce, il n'y a que la réunion. Dans la mesure où les types décrivent des langages réguliers d'arbres et que cette classe est stable par les opérations de complémentaire et d'intersection, il est possible de calculer toutes les combinaisons booléennes, mais cela peut s'avérer assez lourd. Pour typer le filtrage, la solution utilisée dans XDuce [HP01] est de représenter l'intersection et le complémentaire de manière paresseuse, sans effectivement les calculer.

Il semblerait assez naturel, puisque les types intersection et complémentaire sont utiles en eux-mêmes et dans la mesure où l'on a déjà la réunion, d'introduire explicitement ces types dans l'algèbre. C'est ce que nous faisons dans ce travail, et cela simplifie la formalisation des algorithmes pour typer les filtrages.

1.1.8 Les systèmes de types avec intersection et réunion, types récursifs et sous-typage

De nombreux systèmes de types font intervenir une ou plusieurs des caractéristiques que l'on considère : types récursifs, relation de sous-typage, types intersection et réunion.

Les deux premières questions à étudier dans ces systèmes sont celles de la définition et de la décidabilité de la relation de sous-typage ; c'est une propriété de l'algèbre de types elle-même. Pour une algèbre simple, disons celle d'un λ-calcul simplement typé avec une relation de sous-typage sur les types de base, la définition est syntaxique et l'algorithme opère simplement en décomposant les types (il est dirigé par la syntaxe).

Un des travaux fondamentaux est [AC93] où la question est traitée dans le cas d'une algèbre avec types flèche et types récursifs. L'algorithme proposé est une variante de celui sans type récursifs ; il opère en gardant la trace (*memoization*) des paires de types rencontrées dans la descente récursive. Cette méthode est utilisée dans la plupart des travaux sur les types récursifs pour assurer la terminaison des algorithmes.

Les types intersection, éventuellement accompagnés des types réunion ont été beaucoup étudiés (par exemple dans [BDCd95]), soit comme outils pour l'analyse de λ-calculs, soit dans la perspective d'une utilisation dans des langages de programmation.

[Pie90] étudie un λ-calcul simplement typé avec des types réunion et intersection, et montre la puissance expressive obtenue avec ces constructions. La relation de sous-typage est axiomatisée et les règles utilisées sont validées dans un modèle PER. Pour le typage, il y a une règle d'introduction pour l'intersection et une règle d'élimination pour la réunion, qui correspondent à des interprétations intuitionnistes de ces connecteurs booléens (l'élimination de l'intersection et l'introduction de la réunion se font par la règle de subsomption). Il y a également des fonctions surchargées, mais sans possibilité de diriger le calcul suivant les types.

Dans XDuce, la relation de sous-typage n'est pas axiomatisée, mais déduite de l'interprétation des types comme ensembles de valeurs. Nous voulons garder cette interprétation et étudier un système avec toutes les caractéristiques évoquées. Pour résumer, du point de vue de la théorie des types, nous proposons un système avec :

- Constructeurs : types de base, types produit, types flèche.
- Combinaisons booléennes finies arbitraires.
- Types récursifs.
- Interprétation des types comme des contraintes, ce qui donne les fonctions surchargées (avec les types intersection).
- Une relation de sous-typage définie sémantiquement (inclusion des ensembles de valeurs), et non syntaxiquement (axiomatisée par des règles).

Le système de types probablement le plus proche est celui étudié dans [Dam94a, Dam94b, Dam94c]. L'approche utilisée par ces travaux est de coder tous les types comme des expressions régulières d'arbres (et les valeurs comme des arbres). Le problème du sous-typage se ramène donc à celui de l'inclusion de langages dénotés par de telles expressions et ce problème est décidable. Le codage utilisé semble parfois peu naturel : les fonctions sont codées par leur graphe, mais dans les langages réguliers d'arbres on n'a que des arbres avec arité finie et séquences de fils ordonnées. De même, l'interprétation des types dans un modèle à idéaux soulève plusieurs difficultés techniques. En particulier, l'ajout des types complémentaire pose problème, le complémentaire d'un idéal n'est pas un idéal. Essentiellement, l'étude montre des résultats de décidabilité, mais le formalisme strict des langages réguliers d'arbres semble un peu contraignant pour les types flèche. Le simple fait de passer par un codage peut poser des problèmes à l'utilisateur du langage pour comprendre la relation de sous-typage (pour avoir un langage utilisable, il faut que le programmeur puisse comprendre le système de types et, en particulier, la relation de sous-typage).

L'approche de [HVP00], plus directe, permet de dériver des algorithmes efficaces (on ne peut pas améliorer la complexité, mais suivant la présentation des algorithmes, on peut obtenir des programmes plus ou moins rapides) et le formalisme semble plus simple. Concernant la complexité NEXPTIME, qui pourrait sembler décourageante, cette approche répond à la question finale de [Dam94b] : oui, il est possible d'avoir un algorithme raisonnablement efficace dans la plupart des cas.

Nous adopterons cette approche, en montrant comment y intégrer les types flèche. Les types sont interprétés de manière purement ensembliste, et non pas sur des structures ordonnées. De plus, nous obtenons un résultat important de complétude : une instance de sous-typage est vraie si et seulement si toute valeur du premier type est valeur du second type. Nous verrons même qu'il n'est pas nécessaire de travailler avec les valeurs du langage, et que l'on peut considérer, pour décider une instance sous-typage, que les fonctions sont des relations quelconques (fonctions non déterministes).

1.2 Aperçu

1.2.1 Présentation du langage

Présentons brièvement le langage que l'on étudie. Il s'agit d'un λ-calcul, qui pourrait servir de base à un langage spécialisé à la manipulation de documents XML ; on le voit comme un noyau qui se prête bien à l'étude théorique.

Expressions et motifs La syntaxe du langage est la suivante :

$$
\begin{array}{lcll}
\mathtt{e} & ::= & & \text{expressions} \\
 & & \mathtt{c} & \text{constante} \\
 & | & \mathtt{op}(\mathtt{e}) & \text{opérateur} \\
 & | & x & \text{variable} \\
 & | & \mu f : (t_1 \rightarrow s_1 ; \ldots ; t_n \rightarrow s_n).\lambda x.\mathtt{e} & \text{abstraction} \\
 & | & (\mathtt{e}_1, \mathtt{e}_2) & \text{couple} \\
 & | & \text{match } \mathtt{e} \text{ with } p_1 \rightarrow \mathtt{e}_1 \mid p_2 \rightarrow \mathtt{e}_2 & \text{filtrage}
\end{array}
$$

Les ensembles des variables, des constantes et des opérateurs sont fixés. L'application usuelle des λ-calculs sera considérée ici comme un opérateur particulier `apply`.

Les abstractions sont récursives, et on donne dans leur interface leurs différents types déclarés (un seul type pour les fonctions non surchargées) ; dans le corps $\mathtt{e}$ de l'abstraction, on peut utiliser x, qui désigne l'argument de la fonction, et f qui désigne la fonction elle-même. Le choix d'introduire tout de suite des abstractions récursives, au lieu par exemple de considérer d'une part des abstractions non récursives et d'autre part un opérateur de point fixe a plusieurs motivations :

- avec la sémantique stricte que nous allons étudier, le dépliage qui donne la récursivité est introduit au moment de l'application des abstractions ; les autres objets ne peuvent pas être récursifs ;
- le système de types impose de toute manière de donner l'interface des fonctions, et les abstractions non récursives n'auraient donc même pas vraiment une syntaxe plus légère.

Nous utilisons un filtrage à deux branches, le cas général étant exprimable à partir de celui-ci (section 5.4).

Les motifs (p_1 et p_2 dans les productions ci-dessus), sont définis par la syntaxe suivante, avec des règles de bonne formation qui seront précisées plus loin (section 2.4.1) :

p	$::=$		motifs
		x	variable de capture
	$\mid$	t	contrainte de type
	$\mid$	$p_1 \wedge p_2$	conjonction
	$\mid$	$p_1 \vert p_2$	alternative
	$\mid$	$(p, \_)$	couple 1
	$\mid$	$(\_, p)$	couple 2

Les motifs sont en fait des termes infinis avec une hypothèse de régularité (nombre fini de sous-termes), de même que les types récursifs. On pourrait donner une syntaxe pour les exprimer par des expressions finies.

Sémantique Décrivons rapidement le comportement des motifs. L'opération de filtrage prend un objet et lui applique un motif, qui peut soit échouer, s'il refuse de reconnaître l'objet, soit l'accepter en en extrayant certaines composantes qu'il lie à des variables (qui peuvent être utilisées par la suite, dans le corps de la branche).

Le motif variable x capture simplement l'objet filtré et le lie à x. Le motif t échoue si l'objet filtré n'est pas de type t, et sinon il réussit sans lier de variables. Le motif $p_1 \wedge p_2$ réussit lorsque p_1 et p_2 réussissent simultanément ; ils ne doivent pas lier tous les deux la même variable. Le motif $p_1 | p_2$ réussit lorsque l'un des deux motifs p_1 et p_2 réussit ; plus précisément, il essaie d'abord le premier, puis en cas d'échec le deuxième. Les deux doivent donc lier les mêmes variables. Les motifs $(p, \_)$ et $(\_, p)$ sont des destructeurs : si l'objet filtré n'est pas un couple, ils échouent. Sinon, ils utilisent le motif p pour filtrer une des composantes.

Donnons un exemple. On code ici les listes comme en Lisp, avec des couples (tête, queue) et un terminateur distingué. Les motifs infinis permettent d'extraire le premier élément d'une liste qui vérifie un certain critère (lisible sur le type) dans une séquence : le motif p tel que $p = (x \wedge t, \_)|(\_, p)$ extrait ainsi le premier élément de type t. Ce motif accepte n'importe quelle séquence qui a au moins un tel élément. Il faut noter que dans les motifs alternative l'ordre est significatif. Ainsi le motif $p' = (\_, p')|(x \wedge t, \_)$ extrait le *dernier* élément qui est de type t.

Comme les motifs sont infinis, la terminaison du processus de filtrage décrit ci-dessus n'est pas assurée. Pour empêcher un motif de boucler, il faut garantir que l'on croise forcément un nœud couple, ce qui aura pour effet de continuer le filtrage sur des valeurs plus petites (une des deux composantes d'un couple), et donc d'assurer la terminaison.

Le comportement des motifs étant ainsi défini, la sémantique du langage est assez classique ; elle est donnée par une sémantique opérationnelle à petits pas. Les valeurs $v \in \mathcal{V}$ sont intuitivement les résultats des calculs bien typés. Les réductions à la racine sont données par :

$$
\begin{array}{lcl|l}
\mathtt{op}(v) & \longrightarrow & \mathtt{e} & \vdash v : t_{\mathtt{op}}, v \stackrel{\mathtt{op}}{\rightarrow} \mathtt{e} \\
\text{match } v \text{ with } p_1 \rightarrow \mathtt{e}_1 \mid p_2 \rightarrow \mathtt{e}_2 & \longrightarrow & \mathtt{e}_1[v/p_1] & v/p_1 \neq \Omega \\
\text{match } v \text{ with } p_1 \rightarrow \mathtt{e}_1 \mid p_2 \rightarrow \mathtt{e}_2 & \longrightarrow & \mathtt{e}_2[v/p_2] & v/p_1 = \Omega, v/p_2 \neq \Omega \\
\hline
\mathtt{op}(v) & \longrightarrow & \mathbf{wrong} & \not\vdash v : t_{\mathtt{op}} \\
\text{match } v \text{ with } p_1 \rightarrow \mathtt{e}_1 \mid p_2 \rightarrow \mathtt{e}_2 & \longrightarrow & \mathbf{wrong} & v/p_1 = \Omega, v/p_2 = \Omega
\end{array}
$$

Donnons le sens des notations utilisées : v/p représente le résultat du filtrage de la valeur v par le motif p. C'est soit Ω si le filtrage échoue, soit une substitution qui associe à chaque variable dans p la valeur capturée par cette variable. L'erreur de type **wrong** est provoquée soit par l'application d'un opérateur à une valeur qu'il ne sait pas traiter (pour chaque opérateur $\mathtt{op}$, on a un type $t_{\mathtt{op}}$ qui dénote l'ensemble des valeurs qu'il accepte et une relation binaire $\stackrel{\mathtt{op}}{\rightarrow}$ qui donne sa sémantique), soit par un filtrage dont les deux branches échouent.

Pour l'opérateur `apply`, $t_{\mathtt{apply}}$ représente l'ensemble des couples formés d'une fonction et d'une valeur *quelconque*. La sémantique de l'application est donnée par :

$$
\begin{array}{ccl|l}
(\mathtt{f}, v) & \stackrel{\mathtt{apply}}{\rightarrow} & \mathtt{e}[v/x; \mathtt{f}/f] & \exists i.\ \vdash v : t_i \\
(\mathtt{f}, v) & \stackrel{\mathtt{apply}}{\rightarrow} & \mathbf{reject} & \forall i.\ \not\vdash v : t_i
\end{array}
$$

lorsque $\mathtt{f} = (\mu f : (t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.\mathtt{e})$.

Lorsque l'on applique une fonction à une valeur, et que l'interface ne précise pas que la fonction sait traiter cette valeur, la fonction renvoie une valeur particulière **reject**, qui n'est pas l'erreur de type (l'erreur de type n'est pas une valeur). Ce choix un peu particulier est motivé par deux raisons :

- il s'intègre bien dans à la philosophie du langage, où l'on peut faire à l'exécution des tests de type : si l'on ne sait rien sur un objet, sauf qu'il s'agit d'une fonction, on peut toujours lui appliquer un argument et vérifier qu'elle a su ou bien voulu le traiter ;
- les types flèche ne sont que des contraintes ; si une fonction n'est pas de type $t \rightarrow s$ et que l'on dispose d'un objet de type t, il est tout à fait possible que la fonction réussisse à le transformer en un objet de type s ; le typage ne doit donc pas interdire *a priori* une telle application, et il faut par conséquent que la sémantique ne provoque pas d'erreur.

Typage et sous-typage Dans la sémantique intervient le jugement de typage $\vdash$ du langage (en particulier pour définir v/p, à cause des motifs contrainte de type). Le système de règles de typage sera donné plus loin, mais comme on peut s'y attendre, on a besoin de la relation de sous-typage pour le définir. Il se pose un problème de circularité. Nous voulons poser :

$$t \leq s \Leftrightarrow [\![t]\!] \subseteq [\![s]\!]$$

où $[\![t]\!]$ est l'ensemble des valeurs de type t. Mais donc le sous-typage est défini en fonction du typage des valeurs, donc du système de types, qui lui-même utilise le sous-typage.

Remarque 4. *On se restreint aux valeurs, car si l'on interprétait un type comme un ensemble d'expressions, on ne pourrait pas avoir* $[\![t \vee s]\!] = [\![t]\!] \cup [\![s]\!]$ *(le symbole* $\vee$ *introduit les types réunion, que l'on veut justement interpréter comme des réunions ensemblistes). Considérons par exemple une expression qui tire au hasard un entier et suivant sa valeur renvoie des objets de types différents. On peut dire que l'expression a bien le type réunion, mais aucun des deux. Même sans non-déterminisme, le système de types peut être assez puissant pour prouver* $\vdash e : t \vee s$ *sans pouvoir prouver ni* $\vdash e : t$ *ni* $\vdash e : s$.

Remarque 5. *Pour pouvoir interpréter les types comme des ensembles de valeurs et avoir la propriété* $[\![\neg t]\!] = \mathcal{V} \backslash [\![t]\!]$ *($\mathcal{V}$ désigne l'ensemble des valeurs), il faut avoir :* $(\vdash v : \neg t) \Leftrightarrow (\not\vdash v : t)$. *Autrement dit, on doit toujours avoir* $\vdash v : t$ *ou* $\vdash v : \neg t$, *mais pas les deux en même temps. Effectivement, le système de types aura cette propriété. Donc dans la sémantique opérationnelle, on peut remplacer les tests* $\not\vdash v : t$ *par* $\vdash v : \neg t$.

1.2.2 Circularité : sous-typage, filtrage

Pour briser le cercle de dépendance entre le typage et le sous-typage, on pourrait utiliser l'approche classique, qui consiste à axiomatiser la relation de sous-typage, par exemple avec des règles du genre :

$$\frac{t_2 \leq t_1 \quad s_1 \leq s_2}{t_1 \to s_1 \leq t_2 \to s_2}$$

En présence de types récursifs, ces règles doivent être comprises de manière coinductive (l'algorithme mémorise les inégalités déjà vues). Avec en plus les combinaisons booléennes arbitraires, il n'est pas clair de savoir quelles règles adopter ; par exemple le problème $t \leq t_1 \wedge t_2$ se décompose naturellement en deux problèmes plus petits : $t \leq t_1$ et $t \leq t_2$. Mais que faire dans le cas d'un type réunion à droite (ou d'une intersection à gauche) ?

En fait, on verra que la règle ci-dessus pour la flèche est valide, mais que l'on a oublié les cas où $t_2 = 0$ ou $s_2 = 1$ (0 désigne le type vide et 1 son complémentaire, qui est le type « universel ») or dans ces cas, le type à droite n'exprime aucune contrainte.

Nous préférons donc continuer avec l'approche sémantique du sous-typage (inclusion des ensembles dénotés). Il y a une autre circularité dans notre système : le système de types n'utilise pas seulement la relation de sous-typage, mais aussi, pour typer les filtrages, deux fonctions :

- pour tout motif p, il faut disposer du type $\wr p \int$ qui représente les valeurs acceptées par p ;
- si $t \leq \wr p \int$ et si x est une variable qui apparaît dans p, $(t \Downarrow p)(x)$ est un type qui représente les valeurs possibles pour x lorsque l'on filtre une valeur de type t par p (ce filtrage réussit car $t \leq \wr p \int$).

On pourrait donner un algorithme *ad hoc* pour calculer $(t \Downarrow p)(x)$, et prouver ensuite que les types calculés sont effectivement ce que l'on attend. Cela correspond à axiomatiser $(t \Downarrow p)(x)$. Nous préférons, comme pour le sous-typage, déduire l'algorithme d'une définition plus sémantique :

$$[\![(t \Downarrow p)(x)]\!] = \{(v/p)(x) \mid v \in [\![t]\!]\}$$

où $(v/p)(x)$ désigne la valeur extraite de v par la variable x lors du filtrage par le motif p. Il faut donc disposer de la sémantique du filtrage, qui elle même dépend du typage des valeurs.

Résumons dans un diagramme les dépendances dans le langage :

$$\begin{tikzcd}
\text{Système de types} \arrow[rr] \arrow[d] & & \text{Filtrage des types} \arrow[d] \\
\text{Sous-typage} \arrow[r] & \text{Typage des valeurs} \arrow[ul] & \text{Sémantique du filtrage} \arrow[l]
\end{tikzcd}$$

Une flèche désigne une dépendance ; par exemple, pour étudier le filtrage au niveau des types, on a besoin de la sémantique du filtrage.

1.2.3 Modèles

Notre approche pour briser ces circularités est la suivante :

- Nous utilisons l'intuition des fonctions vues comme des relations (fonctions non déterministes, qui ne provoquent pas d'erreur) pour définir une notion abstraite de modèle : un modèle est une interprétation des types comme des parties d'un domaine (d'où une relation de sous-typage), de sorte que les combinaisons booléennes se comportent comme prévu, et que la relation de sous-typage pour les types flèche corresponde à l'intuition.
- Dans le cadre général d'un modèle, nous définissons l'opération de filtrage d'un élément du modèle par un motif.
- En supposant l'existence d'un modèle, nous pouvons mener l'étude du langage. La preuve de l'existence sera faite après cette étude (sans l'utiliser), en exhibant un modèle (le modèle réduit), qui est même universel (dans le sens : si $s \leq t$ est vrai dans un modèle quelconque, c'est vrai dans le modèle réduit), et permet de dériver un algorithme pour calculer la relation de sous-typage.
- L'étude du langage et de son système de types donne une deuxième interprétation des types, cette fois comme ensemble de valeurs du langage, et donc une nouvelle notion de sous-typage et de filtrage.
- Il reste à vérifier que les deux relations de sous-typage coïncident.
- Pour avoir ce résultat, il faut que la définition des modèles soit assez proche de ce qu'il se passe dans le langage. Si dans le langage on renonçait aux fonctions surchargées, on obtiendrait une relation de sous-typage différente et il faudrait considérer une autre définition de modèle. Nous montrons qu'avec notre définition de modèle, tout marche bien.

Le point sensible est la définition de la notion de modèle. Elle est motivée par trois choses :

- Des raisonnements qui s'appuient sur l'intuition des fonctions vues comme des relations (fonctions non déterministes, qui peuvent ne pas terminer, mais qui ne donnent jamais d'erreur).
- Le fait que ce que l'on retrouve dans le langage est bien un modèle.

– La construction du modèle réduit (c'est une formalisation de l'intuition : on prend pour modéliser les fonctions des relations binaires finies).

L'étude théorique du système de types et la preuve du théorème de préservation du typage par la sémantique utilisent l'existence d'un modèle pour obtenir des informations sur la relation de sous-typage et son interaction avec l'algèbre de types. Par exemple, le fait que $t \wedge s \leq t$ est trivial (car dans tout modèle, $[\![t \wedge s]\!] = [\![t]\!] \cap [\![s]\!] \subseteq [\![t]\!]$). Si l'on avait axiomatisé la relation de sous-typage, ou directement donné l'algorithme, il aurait fallu prouver beaucoup de petites propriétés de la sorte et certaines auraient été plus pénibles, par exemple la transitivité de la relation calculée. De plus, il y a beaucoup de petites variantes, de détails d'implémentations et d'optimisations possibles pour l'algorithme de sous-typage ; il serait désagréable de bâtir la théorie dessus.

Dans un modèle, on a le diagramme de dépendance :

$$
\begin{tikzcd}
 & & \text{Filtrage des types} \arrow[dll] \arrow[d] \\
\text{Sous-typage} \arrow[r] & \text{Interprétation des types} & \text{Sémantique du filtrage} \arrow[l]
\end{tikzcd}
$$

Il n'y a plus de circularité !

1.2.4 Le comportement des fonctions ; le traitement de l'erreur

Nous allons préciser l'idée intuitive que l'on a du comportement des fonctions et de l'interprétation des types flèche ; cela va guider le développement.

D'un point de vue intentionnel, nous voyons les fonctions comme des objets dynamiques qui prennent en entrée un argument, effectuent un calcul, et renvoient éventuellement un résultat. Le calcul peut ne pas terminer, et certaines étapes pouvant être non-déterministes, il peut y avoir plusieurs résultats possibles. Du point de vue extensionnel, nous voyons donc les fonctions comme des graphes relationnels (pas forcément fonctionnels, ce qui correspond à autoriser le non-déterminisme).

Une fonction n'échoue jamais ; si on lui donne un argument qu'elle ne sait pas traiter, elle renvoie une objet particulier **reject** pour signaler cela. Il n'y a pas moyen de savoir, de l'extérieur, quand on récupère **reject**, si la fonction n'a pas su traiter l'argument, ou si elle l'a traité et a renvoyé **reject**.

Concernant les types, nous interprétons la flèche $t \rightarrow s$ comme une contrainte : une fonction est dans ce type si tous les résultats possibles qu'elle peut renvoyer quand on lui donne un objet de type t sont de type s. En particulier, si l'objet **reject** n'est pas de type s, alors toute fonction de type $t \rightarrow s$ sait traiter n'importe quel objet de type t. Lorsque l'on applique une fonction de type $t \rightarrow s$ à un objet de type t' avec $t' \not\leq t$, et si l'on n'a pas plus d'information, on ne peut rien dire du résultat (en particulier, on ne peut *pas* dire que le résultat sera **reject**).

Évidemment, dans le cas d'un langage assez expressif, il est impossible de décider si une fonction satisfait une telle contrainte $t \longrightarrow s$. Cependant, nous allons nous laisser guider par ces intuitions pour étudier la relation de sous-typage. Par exemple, nous les utiliserons pour montrer la propriété fondamentale de disjonction forte dans l'univers des types flèche. Cela peut s'écrire :

$$\bigwedge t_i \to s_i \leq \bigvee t'_j \to s'_j \Rightarrow \exists j.\ \bigwedge t_i \to s_i \leq t'_j \to s'_j$$

(Les symboles $\bigwedge$ et $\bigvee$ désignent les intersections et réunions finies de types)

Remarque 6. *Considérons l'inégalité :*

$$t \to (s_1 \vee s_2) \leq (t \to s_1) \vee (t \to s_2)$$

En général, elle est fausse bien entendu, mais regardons le cas dégénéré où t est un type singleton. Si on voit les fonctions comme des fonctions mathématiques (qui sont déterministes), l'inclusion est alors vraie, et l'on n'a pas de disjonction forte. On a bien disjonction forte lorsque l'on utilise des fonctions non déterministes.

Regardons ce qu'il se passe dans la catégorie des espaces topologiques par exemple. Si t est connexe et s_1, s_2 sont des ouverts disjoints (ou des fermés disjoints), et que l'on interprète la flèche $t \to s$ comme l'ensemble des applications continues f telles que $f(t) \subseteq s$, alors on a bien le sous-typage indiqué plus haut. On voit que des dégénérescences (finitude du domaine) ou des obstructions sur les fonctions (continuité) changent la relation de sous-typage. Les fonctions que nous avons en tête sont violemment discontinues par rapport aux types : elles peuvent en effet effectuer un test sur le type de l'argument et se comporter complètement différemment suivant le cas. Autre manière de le dire : pour tout type t, la fonction caractéristique de t est définissable.

Chapitre 2

Algèbre de types, modèles, sous-typage, filtrage

2.1 Notations

On utilisera dans les différentes parties techniques les notations suivantes pour désigner des constructions générales dans la catégorie des ensembles :

$$\begin{array}{lcl}
\mathcal{P}(X) & : & \text{ensemble des parties de } X \\
\mathcal{P}_f(X) & : & \text{ensemble des parties finies de } X \\
+ & : & \text{somme disjointe} \\
\complement_Y X & : & \text{complémentaire de } X \text{ dans } Y \\
X \to_f Y & : & \text{graphes fonctionnels finis} \subseteq X \times Y \\
X \rtimes Y & = & \{(x,y) \in X \times Y \mid x \cap y = \varnothing\}
\end{array}$$

2.2 L'algèbre de types

Commençons l'étude formelle en introduisant l'algèbre de types. La présentation adoptée facilite les développements techniques.

2.2.1 Discussion

L'algèbre de types $\mathbf{T}$ a trois *univers* : l'univers **basic** des types de base, l'univers **product** des types produit, l'univers **arrow** des types flèche. La métavariable u désignera toujours un élément de $\{\mathbf{product}, \mathbf{arrow}, \mathbf{basic}\}$.

Nous définirons plus bas ce que sont les types $t \in \mathbf{T}$. Dans chaque univers, il y a des types atomiques $\mathbf{T}_u^\circ$. Pour l'univers **basic**, nous fixons un ensemble $\mathbf{T}_{\mathbf{basic}}^\circ$, qui est un paramètre de notre étude (on donnera plus loin des exemples de choix intéressants pour $\mathbf{T}_{\mathbf{basic}}^\circ$). Pour **product** et **arrow**, les types atomiques sont des couples de types ; un couple $(t_1, t_2) \in \mathbf{T}^2$ vu comme élément de $\mathbf{T}_{\mathbf{product}}^\circ$ (resp. $\mathbf{T}_{\mathbf{arrow}}^\circ$) est noté $t_1 \times t_2$ (resp. $t_1 \to t_2$).

Dans chaque univers, nous voulons construire des combinaisons booléennes de types atomiques. Pour fixer les idées, dans le cas général, une combinaison booléenne d'éléments dans un ensemble A est un terme ϕ engendré par les productions :

$$\phi ::= a \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi$$

($a \in A$). Nous n'utiliserons pas cette syntaxe concrète.

Remarque 7. *Ce système de connecteurs est complet, mais ni exhaustif, ni minimal ; nous le choisissons simplement parce qu'il correspond à ce que nous voulons faire en pratique.*

Toute combinaison booléenne peut se mettre sous forme normale disjonctive, et nous allons travailler directement sur cette représentation. Une forme normale peut être représentée par un objet dans $\mathcal{P}_f(\mathcal{P}_f(A) \times \mathcal{P}_f(A))$. Chaque élément correspond à une intersection, donnée par un couple ; le premier élément du couple donne les éléments de A qui apparaissent positivement dans l'intersection, et le deuxième donne les éléments qui apparaissent négativement (sous forme de leur négation). Par exemple, $a \vee (b \wedge c \wedge \neg d) \vee \neg e$ est représentée par $\{(\{a\},\{\}),(\{b,c\},\{d\}),(\{\},\{e\})\}$.

Remarque 8. *Au lieu de prendre $\mathcal{P}_f(A) \times \mathcal{P}_f(A)$ pour représenter une intersection d'une combinaison booléenne en forme normale, on peut prendre $\mathcal{P}_f(A) \rtimes \mathcal{P}_f(A) = \{(x,y) \in \mathcal{P}_f(A) \times \mathcal{P}_f(A) \mid x \cap y = \varnothing\}$. En effet, si un même élément a apparaît positivement et négativement, l'intersection est vide, et on peut la supprimer de la forme normale. Cela ne change presque rien à l'exposé formel, mais pour l'efficacité de l'implémentation, il faut absolument faire cela.*

Par définition, un u-type est une combinaison booléenne de u-types atomiques en forme normale disjonctive et un type t est un triplet ($t_{\mathbf{basic}}$, $t_{\mathbf{product}}$, $t_{\mathbf{arrow}}$) où les $t_u \in \mathbf{T}_u$ sont des u-types. La notation t_u désignera suivant le contexte un u-type arbitraire ou la u-composante du type t. Il faut donc avoir :

$$\begin{array}{rcl} \mathbf{T} & = & \prod_u \mathbf{T}_u \\ \mathbf{T}_u & = & \mathcal{P}_f(\mathcal{P}_f(\mathbf{T}_u^\circ) \times \mathcal{P}_f(\mathbf{T}_u^\circ)) \end{array}$$

2.2.2 Définition : les types récursifs

Résumons. Les types $\mathbf{T}$, les u-types $\mathbf{T}_u$ et les u-types atomiques $\mathbf{T}_u^\circ$ sont définis par les équations récursives :

$$\left\{ \begin{array}{lcl} \mathbf{T} & = & \prod_u \mathbf{T}_u \\ \mathbf{T}_u & = & \mathcal{P}_f(\mathcal{P}_f(\mathbf{T}_u^\circ) \times \mathcal{P}_f(\mathbf{T}_u^\circ)) \\ \mathbf{T}_{\mathbf{product}}^\circ & = & \mathbf{T}^2 \\ \mathbf{T}_{\mathbf{arrow}}^\circ & = & \mathbf{T}^2 \end{array} \right.$$

(les égalités sont à prendre comme des isomorphismes implicites)

Il y a plusieurs choix possibles de solutions pour ce système. On peut construire une solution (la plus petite, ou initiale) en prenant des termes finis ; si l'on fait cela, il ne sera pas possible de considérer des types récursifs. La plus grande solution est donnée par des termes infinis.

Definition 2.2.2.1. *Les* types récursifs *sont ceux, parmi les termes infinis, qui n'ont qu'un nombre fini de sous-termes distincts. On les appelle simplement types.*

On voit que l'ensemble des types récursifs (et les u-types et u-types atomiques associés) satisfait encore les équations, et c'est cette solution que nous allons considérer.

Remarque 9. *Les termes sont définis comme d'habitude par des arbres ; ici, les nœuds types,* **product***-types atomiques, et les* **arrow***-types atomiques ont une arité fixée (3,2,2) et leurs fils sont ordonnés, alors que pour les nœuds u-types, c'est un peu plus complexe, mais cela ne pose pas de problème pour définir la notion de terme fini, de terme infini, et de sous-terme.*

Remarque 10. *Nous pourrions travailler avec les types infinis pour tout le traitement formel et l'étude du sous-typage, mais pour obtenir un algorithme effectif pour décider la relation de sous-typage, il faut évidemment prendre des types représentables en machine, donc de manière finie. La classes des types récursifs est la plus naturelle ; elle correspond à ce que l'on appelle en général type récursif en théorie des types. Nous n'allons pas formaliser ici la notation de point fixe $\mu\alpha.t$ qui permet d'écrire effectivement les types récursifs. Signalons seulement que la définition de l'algèbre de types interdit les types mal formés comme $\mu\alpha.\alpha \lor \alpha$. En effet, par construction, on « croise » un constructeur $\times$ ou $\rightarrow$ dans chaque branche infinie.*

2.2.3 Les connecteurs booléens

Nous avons construit les types de manière à pouvoir exprimer les combinaisons booléennes dans chaque univers. Nous allons utiliser l'intuition que l'on a sur les combinaison booléennes en forme normale pour définir les opérations sur les u-types puis les types.

Pour les u-types Définissons deux u-types par :

$$\begin{aligned} 0_u &\stackrel{\text{def}}{=} \{\} \\ 1_u &\stackrel{\text{def}}{=} \{(\{\},\{\})\} \end{aligned}$$

Définissons des opérations de réunion et d'intersection sur les u-types, notés $\lor$ et $\land$ par :

$$\begin{aligned} s_u \lor t_u &\stackrel{\text{def}}{=} s_u \cup t_u \\ s_u \land t_u &\stackrel{\text{def}}{=} \{(P \cup P', N \cup N') \mid (P,N) \in s_u \land (P',N') \in t_u\} \end{aligned}$$

On voit que ces opérateurs sont commutatifs et associatifs, et qu'ils ont comme éléments neutres 0_u et 1_u. Cela justifie d'utiliser les notations $\bigvee_i t_u^i$ et $\bigwedge_i t_u^i$.

Définissons enfin l'opérateur complémentaire, noté $\neg_u$ par :

$$\neg_u t \stackrel{\text{def}}{=} \bigwedge_{(P,N)\in t_u} \bigvee_{a\in N} \{(\{a\},\{\})\} \lor \bigvee_{a\in P} \{(\{\},\{a\})\}$$

Nous pouvons faire la convention d'écriture suivante : on identifie un u-type atomique a avec le u-type $\{(\{a\},\{\})\}$. On peut alors écrire, pour tout u-type t_u :

$$t_u = \bigvee_{(P,N)\in t_u} \bigwedge_{a\in N} a \land \bigwedge_{a\in P} \neg_u a$$

Pour les types Si l'on voit intuitivement les types comme des ensembles de valeurs, l'ensemble des valeurs étant lui aussi partitionné en univers, de sorte que les u-types dénotent des ensembles de valeurs dans l'univers u, on constate que les opérations booléennes (réunion, intersection, complémentaire) se calculent univers par univers. Cela pousse à définir :

$$\begin{array}{lcl} 0 & = & (0_u)_{u \in \{\mathbf{basic}, \mathbf{product}, \mathbf{arrow}\}} \\ 1 & = & (1_u) \\ s \wedge t & = & (s_u \wedge t_u) \\ s \vee t & = & (s_u \vee t_u) \\ \neg t & = & (\neg_u t_u) \end{array}$$

Nous utilisons les mêmes notations de connecteurs $\vee, \wedge$ que pour les u-types. Cela ne posera pas de problème.

Nous allons encore adopter une convention d'écriture, et identifier un u-type t_u avec le type dont les autres composantes sont nulles. Il faut faire attention à ce que $\neg t_u \neq \neg_u t_u$ (dans le membre gauche, les autres composantes sont pleines). C'est la raison pour laquelle nous utilisons une notation différente pour $\neg$ et $\neg_u$.

Avec cette convention, si t_1 et t_2 sont deux types, alors $t_1 \rightarrow t_2$ et $t_1 \times t_2$ désignent encore des types.

2.2.4 Bases

Les notions de base et d'opérateur stable sont utilisées pour assurer la terminaison des algorithmes.

Definition 2.2.4.1. *Le* socle $\theta(\_)$ *d'un type, d'un u-type, ou d'un u-type atomique est l'ensemble des types défini par :*

$$\begin{array}{lcl} \theta(t) & = & \bigcup_u \theta(t_u) \\ \theta(t_u) & = & \bigcup_{(P,N) \in t_u} \bigcup_{a \in P \cup N} \theta(a) \\ \theta(b) & = & \{b\} \\ \theta(t_1 \rightarrow t_2) & = & \{t_1, t_2\} \\ \theta(t_1 \times t_2) & = & \{t_1, t_2\} \end{array}$$

Definition 2.2.4.2. *On appelle* base de représentation *(simplement* base*) tout ensemble* fini $\mathcal{X}$ *de types stable par socle (si $t \in \mathcal{X}$, alors $\theta(t) \subseteq \mathcal{X}$).*

Fait 2.2.4.3. *Étant donné un ensemble fini A de types, le plus petit ensemble de types stable par socle et contenant A est une base, dite base engendrée par A.*

Démonstration. Cet ensemble est fini parce que les types, vu comme des arbres, n'ont qu'un nombre fini de sous-arbres. □

Definition 2.2.4.4. *Soit $\mathcal{X}$ une base. Un type t est* $\mathcal{X}$-définissable *si $\theta(t) \subseteq \mathcal{X}$.*

Fait 2.2.4.5. *Pour une base $\mathcal{X}$ donnée, l'ensemble des types $\mathcal{X}$-définissables est fini.*

Fait 2.2.4.6. *Tout type t est $\mathcal{X}$-définissable pour une certaine base.*

Démonstration. Il suffit de prendre la base engendrée par $\theta(t)$. □

Nous définissons de même la notion de u-type $\mathcal{X}$-définissable et nous avons les mêmes propriétés.

Definition 2.2.4.7. *Un opérateur (unaire, binaire, ...) des types (ou u-types) dans les types (ou u-types) est* stable *si pour toute base $\mathcal{X}$, l'image de types $\mathcal{X}$-définissables est $\mathcal{X}$-définissable.*

(une autre manière de dire cela, c'est que le socle de l'image est inclus dans la clôture transitive du socle des arguments)

Les notions de base de définition et d'opérateur stable sont importantes pour assurer la terminaison des algorithmes que nous allons présenter. Ceux-ci procèdent en saturant des ensembles de types par certains opérateurs. Si on peut garantir que ces opérateurs ne donnent que des types $\mathcal{X}$-définissables pour une base $\mathcal{X}$ fixée, on sait que l'algorithme n'aura à considérer qu'un nombre fini de types.

Fait 2.2.4.8. *Les opérateurs booléens, sur les types ou les u-types sont stables.*

Les archétypes des opérateurs qui ne sont pas stables sont $(t,s) \mapsto t \to s$ et $(t,s) \mapsto t \times s$. Ainsi, si l'on part d'un ensemble $\{b\}$ où $b \in \mathbf{T}^{\circ}_{\mathbf{basic}}$, et qu'on le sature par l'un de ces opérateurs, on obtient un ensemble infini : $\{b, b \times b, b \times (b \times b), (b \times b) \times b, \dots\}$. L'intuition est que les opérateurs stables ne créent pas de nouveaux constructeurs $\times$ ou $\to$: les constructeurs que l'on trouve dans l'image sont déjà présents dans les arguments. Il y a essentiellement deux genres d'opérateurs stables :

- les opérateurs booléens ;
- les opérateurs de déconstruction (comme par exemple l'opérateur de projection π_1 introduit plus loin).

Remarque 11. *La notion de base de définition est intéressante également pour le problème de l'affichage des types sous une forme lisible par l'utilisateur. Nous verrons que l'algorithme de typage n'introduit pas de récursion dans les types, sauf pour le calcul des types acceptés par les motifs récursifs ; cela permet, lorsque que l'on veut afficher un type de garder les variables de récursion données par l'utilisateur.*

2.3 Modèles

Pour l'instant, les types ne sont que des expressions formelles. Nous allons maintenant leur donner une sémantique en les interprétant comme des ensembles d'objets dans une structure donnée. Une telle interprétation permet de définir la relation de sous-typage comme l'inclusion des ensembles dénotés. Cette relation de sous-typage sera utilisée dans le système de types du langage ; pour que le typage assure la sûreté de la sémantique, il faut bien entendu certaines propriétés sur la relation de sous-typage. Lorsqu'elles sont vérifiées, on dit que l'interprétation est un modèle.

Nous allons naturellement interpréter les types de base comme des ensembles de constantes ; les types produit seront interprétés comme des ensembles de couples. Pour les types flèche, il est moins clair de voir ce qu'il faut prendre comme interprétation. Dans l'approche extensionnelle, on voudrait considérer toutes les relations, mais alors il se pose un problème de cardinalité pour définir la structure. Dans l'approche intentionnelle, on introduirait un λ-calcul typé, son système de types, et on interpréterait les types flèche comme les abstractions ayant ce type. Le problème, c'est que pour définir le système de types, on doit déjà disposer de la relation de sous-typage.

La démarche est la suivante :
- Nous allons laisser en suspens le choix de l'interprétation des types flèche, et poser simplement les conditions dont nous aurons besoin pour étudier notre langage et prouver la sûreté du typage.
- En supposant l'existence d'un modèle vérifiant ces conditions, dit modèle d'amorce, nous pourrons mener l'étude du langage, et voir qu'il donne naissance à son tour à un modèle.
- Il serait gênant que le modèle du langage et le modèle d'amorce donnent une relation de sous-typage différente ; nous verrons que ce n'est pas le cas.
- Il ne restera plus qu'à exhiber un tel modèle.

Lorsque que nous parlerons d'algorithmes dans l'étude du langage, nous supposerons que la relation de sous-typage définie par le modèle est calculable par un algorithme. Pour le modèle que l'on exhibe, nous donnerons un tel algorithme.

2.3.1 Types de base

Nous nous donnons pour chaque **basic**-type atomique b un ensemble, noté $[\![b]\!]_{\mathbf{basic}}$. Posons $\mathcal{D}_{\mathbf{basic}} = \bigcup_{b \in \mathbf{T}^{\circ}_{\mathbf{basic}}} [\![b]\!]_{\mathbf{basic}}$ et appelons les éléments de $\mathcal{D}_{\mathbf{basic}}$ les constantes.

Nous allons introduire une notion d'interprétation pour les autres types, et nous travaillerons parfois avec plusieurs interprétations différentes, mais cette interprétation des **basic**-type atomiques est fixée une fois pour toutes.

Le choix des types de base Nous verrons que dans le langage, l'opération de filtrage ne peut distinguer les valeurs que par leur type. Il faut donc choisir une algèbre de types suffisamment « fine ». Par exemple, si l'on représente les *tags* XML comme des simples chaînes des caractères, il n'est pas suffisant d'avoir un type **string**. Il faut également disposer des types singletons, sous-type de **string**, qui dénotent chacun une unique chaîne. De manière générale, pour chaque type « classique », **string**, **int** ..., il est utile d'avoir un sous-type singleton pour chaque constante.

On peut envisager des types plus larges, qui correspondent à des domaines abstraits. Par exemple pour les entiers, on peut considérer des types de base intervalles, de la forme $[a;b]$ (a, b entiers) ; pour les chaînes de caractères, on peut envisager d'utiliser comme types de base des expressions régulières.

Nous verrons que la seule contrainte sur le choix des **basic**-types atomiques est qu'il faut être capable de décider l'inclusion d'une intersection finie de tels types dans une réunion finie. Pour les intervalles, ou les expressions régulières, on sait le faire.

2.3.2 Prémodèle : structure et interprétation

Une structure est un domaine des « valeurs » $\mathcal{D}$; de même que pour les types, ce domaine est partitionné en univers :

$$\mathcal{D} = \mathcal{D}_{\mathbf{basic}} + \mathcal{D}_{\mathbf{product}} + \mathcal{D}_{\mathbf{arrow}}$$

Le sous-domaine $\mathcal{D}_{\mathbf{basic}}$ est fixé ; c'est l'ensemble des constantes. Pour le sous-domaine $\mathcal{D}_{\mathbf{product}}$, il est naturel de supposer que $\mathcal{D}_{\mathbf{product}} = \mathcal{D}^2$ (en fait, il s'agit d'une bijection que l'on garde implicite). Nous ne voulons pas considérer de valeur infinie (cyclique), donc nous supposons que l'ordre défini sur $\mathcal{D}$ par $(e_1, e_2) > e_1$, $(e_1, e_2) > e_2$ est bien fondé. Pour $\mathcal{D}_{\mathbf{arrow}}$, nous ne précisons rien pour le moment.

Definition 2.3.2.1 (Structure). *Une structure est un ensemble $\mathcal{D}$ partitionné en $\mathcal{D}_{\mathbf{basic}}, \mathcal{D}_{\mathbf{product}}, \mathcal{D}_{\mathbf{arrow}}$:*

$$\mathcal{D} = \mathcal{D}_{\mathbf{basic}} + \mathcal{D}_{\mathbf{product}} + \mathcal{D}_{\mathbf{arrow}}$$

tel qu'il existe une bijection implicite

$$\mathcal{D}_{\mathbf{product}} = \mathcal{D}^2$$

et que de plus l'ordre défini sur $\mathcal{D}$ par $(e_1, e_2) > e_1$, $(e_1, e_2) > e_2$ est bien fondé.

(on utilise la bijection implicite dans cette dernière condition). Pour définir une structure, on doit donc simplement donner $\mathcal{D}_{\mathbf{arrow}}$ (rappelons que $\mathcal{D}_{\mathbf{basic}}$ est choisi une fois pour toutes) ; le domaine $\mathcal{D}$ est alors caractérisé comme la plus petite solution de l'équation suivante (dans la catégorie des ensembles) :

$$\begin{cases} \mathcal{D} & = & \mathcal{D}_{\mathbf{basic}} + \mathcal{D}_{\mathbf{product}} + \mathcal{D}_{\mathbf{arrow}} \\ \mathcal{D}_{\mathbf{product}} & = & \mathcal{D}^2 \end{cases}$$

Lorsque l'on dispose d'une structure $\mathcal{D}$, on cherche à interpréter les types comme des ensembles d'objets de $\mathcal{D}$. Nous demandons que l'interprétation des **basic**-types atomiques, des combinaisons booléennes et des types produit se comporte comme on s'y attend ; pour les types flèche, nous ne demandons rien pour l'instant.

Definition 2.3.2.2 (Interprétation). *Une interprétation est une application $[\![\_]\!] : \mathbf{T} \longrightarrow \mathcal{P}(\mathcal{D})$ vérifiant, pour tous types t, t_1, t_2 :*

$$\begin{array}{lcl} [\![b]\!] & = & [\![b]\!]_{\mathbf{basic}} \subseteq \mathcal{D}_{\mathbf{basic}} \\ [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \subseteq \mathcal{D}_{\mathbf{product}} \\ [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathcal{D} \backslash [\![t]\!] \\ [\![\mathbb{0}]\!] & = & \varnothing \\ [\![\mathbb{1}_u]\!] & = & \mathcal{D}_u \end{array}$$

Par convention d'écriture, nous avions dit qu'un u-type t_u peut être vu comme un type ; l'égalité $\mathbb{1}_u \wedge t_u = t_u$ montre que $[\![t_u]\!] \subseteq \mathcal{D}_u$. Notons $[\![\_]\!]_u : \mathbf{T}_u \longrightarrow \mathcal{P}(\mathcal{D}_u)$ les restrictions de $[\![\_]\!]$ à chaque univers ; pour un **basic**-type atomique b, cela coïncide bien avec $[\![b]\!]_{\mathbf{basic}}$ que nous nous étions donné. Il est facile de voire que les conditions de la définition sont alors équivalentes à :

$$\begin{array}{lcl} [\![t]\!] & = & \bigcup_u [\![t_u]\!]_u \\ [\![t_u]\!]_u & = & \bigcup_{(P,N) \in t_u} \mathcal{D}_u \cap \bigcap_{a \in P} [\![a]\!]_u \backslash \bigcup_{a \in N} [\![a]\!]_u \\ [\![t_1 \times t_2]\!]_{\mathbf{product}} & = & [\![t_1]\!] \times [\![t_2]\!] \end{array}$$

(dans la première égalité, t_u désigne la u-composante du type t) Une interprétation est définie simplement par la donnée de l'application $[\![\_]\!]_{\mathbf{arrow}} : \mathbf{T}^{\circ}_{\mathbf{arrow}} \longrightarrow \mathcal{P}(\mathcal{D}_{\mathbf{arrow}})$ (encore une fois, l'interprétation des **basic**-types atomiques est fixée une fois pour toutes) ; en effet, la fonction $[\![\_]\!]$ est alors définie par induction sur e (avec l'ordre bien fondé) par les équivalences suivantes :

$$\begin{array}{lll} e \in [\![t]\!] & \Leftrightarrow & \exists u.\ e \in [\![t_u]\!]_u \\ e \in [\![t_u]\!]_u & \Leftrightarrow & (\exists (P,N) \in t_u. \\ & & (\forall a \in P.\ e \in [\![a]\!]_u) \wedge (\forall a \in N.\ e \notin [\![a]\!]_u)) \\ (e_1, e_2) \in [\![t_1 \times t_2]\!]_{\mathbf{product}} & \Leftrightarrow & (e_1 \in [\![t_1]\!]) \wedge (e_2 \in [\![t_2]\!]) \end{array}$$

Definition 2.3.2.3 (Prémodèle, sous-typage). *Un prémodèle est donné par une structure et une interprétation. Dans un prémodèle, on peut définir un préordre de sous-typage sur* $\mathbf{T}$ *par :*

$$t \leq s \Leftrightarrow [\![t]\!] \subseteq [\![s]\!]$$

On note $\simeq$ *l'équivalence induite par ce préordre. Deux prémodèles sont équivalents s'ils induisent la même relation de sous-typage.*

Retenons que pour définir un prémodèle, il suffit de donner $\mathcal{D}_{\mathbf{arrow}}$ et $[\![\_]\!]$ sur les **arrow**-types atomiques.

2.3.3 Étude du problème de sous-typage

Réduction du problème Nous cherchons un algorithme pour décider si $t \leq s$, lorsque t et s sont deux types. Compte-tenu de :

$$t \leq s \Leftrightarrow t \wedge \neg s \simeq 0$$

puis de :

$$t \simeq 0 \Leftrightarrow \forall u.\ [\![t_u]\!]_u = \varnothing$$

nous ramenons le problème à un univers u donné. Enfin :

$$[\![t_u]\!]_u = \varnothing \Leftrightarrow \forall (P,N) \in t_u. \bigcap_{a \in P} [\![a]\!]_u \subseteq \bigcup_{a \in N} [\![a]\!]_u$$

Nous devons donc étudier, pour chaque univers u, le problème du sous-typage avec à gauche une intersection de u-types atomiques et à droite une réunion de u-types atomiques.

Univers des types de base Nous ne pouvons pas décomposer plus. Il faut savoir décider, lorsque $b_1, \ldots, b_n, b'_1, \ldots, b'_m$ sont des types de base, si :

$$\bigcap_{1 \leq i \leq n} [\![b_i]\!]_{\mathbf{product}} \subseteq \bigcup_{1 \leq j \leq m} [\![b'_j]\!]_{\mathbf{product}}$$

Nous supposons donc que l'on dispose d'un algorithme pour décider si cette inclusion est vraie.

Univers des types produit Nous étudions le problème dans $\mathcal{D}_{\mathbf{product}}$

$$\bigcap_{i \in I} [\![t_i]\!] \times [\![s_i]\!] \subseteq \bigcup_{j \in J} [\![t'_j]\!] \times [\![s'_j]\!]$$

En utilisant le fait B.1.0.1 (section B.1), nous voyons que le problème est équivalent à :

$$\forall J' \subseteq J.(\bigcap_{i \in I} [\![t_i]\!] \subseteq \bigcup_{j \in J'} [\![t'_j]\!]) \vee (\bigcap_{i \in I} [\![s_i]\!] \subseteq \bigcup_{j \in J \setminus J'} [\![s'_j]\!])$$

Nous prouvons dans le fait B.2 le théorème suivant :

Théorème 2.3.3.1 (Opérateur de projection). *Tout prémodèle induit sur l'algèbre de types un opérateur $\pi_1 : \mathbf{T} \longrightarrow \mathbf{T}$ vérifiant :*

$$[\![\pi_1(t)]\!] = \{e_1 \mid \exists e_2.\ (e_1, e_2) \in [\![t]\!]\}$$

De plus, cet opérateur est stable et deux prémodèles équivalents donnent le même opérateur.

Bien entendu, nous définissons de même π_2. Ces opérateurs sont croissants : c'est clair d'après leur définition. La stabilité sera utilisée pour prouver la terminaison de l'algorithme d'inférence du type des résultats des filtrages.

Univers des types flèche Nous étudions le problème dans $\mathcal{D}_{\mathbf{arrow}}$

$$\bigcap_{i \in I} [\![t_i \to s_i]\!]_{\mathbf{arrow}} \subseteq \bigcup_{j \in J} [\![t'_j \to s'_j]\!]_{\mathbf{arrow}}$$

Sans plus d'information sur l'application $[\![\_]\!]_{\mathbf{arrow}}$, nous ne pouvons rien dire. Nous allons nous laisser guider par l'intuition et des calculs simples pour trouver les bonnes conditions sur la relation de sous-typage dans l'univers des types flèche. Cela va donner la définition d'un modèle.

2.3.4 Inclusion des types flèche ; définition d'un modèle

Intuition

Nous partons de l'intuition suivante : les fonctions sont non déterministes et peuvent ne pas finir (ou alors, du point de vue extensionnel, elles peuvent ne pas être partout définies). Si $e_1 \in \mathcal{D}_{\mathbf{arrow}}$ et $e_2 \in \mathcal{D}$, nous supposons donné un sous-ensemble de $\mathcal{D}$, noté $e_1.e_2$, qui donne les résultats possibles de l'« application » de la fonction e_1 à l'argument e_2. Un tel opérateur $\mathcal{D}_{\mathbf{arrow}} \times \mathcal{D} \longrightarrow \mathcal{P}(\mathcal{D})$ est appelé notion d'application.

Dans le langage, pour typer l'application, on doit disposer, étant donnés un **arrow**-type $t_{\mathbf{arrow}}$ et un type t, d'un type $t_{\mathbf{arrow}}.t$ qui représente l'ensemble des objets que l'on peut obtenir en appliquant une fonction de type $t_{\mathbf{arrow}}$ à un objet de type t, c'est-à-dire :

$$[\![t_{\mathbf{arrow}}.t]\!] = \bigcup_{e_1 \in [\![t_{\mathbf{arrow}}]\!], e_2 \in [\![t]\!]} e_1.e_2$$

Avec l'interprétation intuitive du type flèche comme une contrainte, nous voyons que toutes les fonctions de type $t_{\mathbf{arrow}}$ sont aussi de type $t \to t_{\mathbf{arrow}}.t$:

$$t_{\mathbf{arrow}} \ \leq \ t \to t_{\mathbf{arrow}}.t$$

et $t_{\mathbf{arrow}}.t$ est le plus petit type qui vérifie cette inéquation.

La disjonction forte Toujours avec l'interprétation intuitive, nous pouvons motiver une première propriété fondamentale : la disjonction forte qui signifie qu'une intersection de types flèche est plus petite qu'une réunion de types flèche si et seulement si elle est plus petite que l'un d'entre eux :

$$(\star) \qquad \left[\bigwedge t_i \to s_i \leq \bigvee t'_j \to s'_j\right] \Rightarrow \left[\exists j.\ \bigwedge t_i \to s_i \leq t'_j \to s'_j\right]$$

(la réciproque est toujours vraie dans un prémodèle)

Il est plus simple de comprendre la contraposée. Supposons que pour tout j, $\bigwedge t_i \to s_i \not\leq t'_j \to s'_j$, c'est-à-dire : $(\bigwedge t_i \to s_i).t'_j \not\leq s'_j$ Nous disposons donc de $f_j \in [\![\bigwedge t_i \to s_i]\!]$, de $e_j \in [\![t'_j]\!]$ et de $e'_j \notin [\![s'_j]\!]$ de sorte que $e'_j \in f_j.e_j$. Comme nous avons des fonctions non déterministes, nous disposons intuitivement également d'une fonction f qui opère, quand on l'appelle avec un argument, en tirant au hasard un j et en se comportant ensuite comme f_j ; c'est-à-dire que $f.e = \bigcup_j f_j.e$ pour tout $e \in \mathcal{D}$. Cette fonction f satisfait toujours les contraintes $t_i \to s_i$, mais elle ne satisfait aucune des contraintes $t'_j \to s'_j$; en effet, si on lui donne e_j, il est possible que le tirage au sort choisisse la branche j et un résultat possible est alors $e'_j \notin [\![s'_j]\!]$. Nous avons donc exhibé une fonction $f \in [\![\bigwedge t_i \to s_i]\!]$, $f \notin [\![\bigvee t'_j \to s'_j]\!]$, ce qui prouve $(\star)$.

Sous-typage Nous cherchons à trouver la bonne définition pour la relation de sous-typage dans l'univers des types flèche.

Étudions $(\bigwedge t_i \to s_i).t$. Prenons $f \in [\![\bigwedge t_i \to s_i]\!]$ et $e \in [\![t]\!]$. Si $e \in [\![t_i]\!]$, alors $f.e \subseteq [\![s_i]\!]$, et donc :

$$[\![(\bigwedge t_i \to s_i).t]\!] \subseteq \bigcup_{e \in [\![t]\!]} \ \bigcap_{i | e \in [\![t_i]\!]} [\![s_i]\!]$$

Pour assurer la sûreté du typage du langage, nous aurons besoin que le type du résultat soit effectivement dans $t_{\mathbf{arrow}}.t$ lorsque l'on applique une fonction de type $t_{\mathbf{arrow}}$ à un objet de type t ; il faut donc une inégalité dans l'autre sens pour $t_{\mathbf{arrow}}.t$. Comme le langage est suffisamment riche, nous verrons que l'on a besoin de l'égalité :

$$[\![(\bigwedge t_i \to s_i).t]\!] = \bigcup_{e \in [\![t]\!]} \ \bigcap_{i | e \in [\![t_i]\!]} [\![s_i]\!]$$

Un calcul simple montre que la réunion est égale à :

$$\bigcup_{\substack{I' \subseteq I \\ [\![t]\!] \not\subseteq \bigcup_{i \in I \setminus I'} [\![t_i]\!]}} \ \bigcap_{i \in I'} [\![s_i]\!]$$

Autrement dit, il faut avoir :

$$(\bigwedge t_i \to s_i).t \simeq \bigvee_{\substack{I' \\ t \not\leq \bigvee_{i \in I \setminus I'} t_i}} \ \bigwedge_{i \in I'} s_i$$

Comme $(\bigwedge t_i \to s_i).t$ est la plus petite solution s de l'inéquation : $\bigwedge t_i \to s_i \leq t \to s$, nous voyons, en utilisant la propriété de disjonction forte, que :

$$\begin{array}{ll} & \bigwedge_{i \in I} t_i \to s_i \leq \bigvee_{j \in J} t'_j \to s'_j \\ \Leftrightarrow & \exists j \in J.\ \forall I' \subseteq I.\ (t'_j \leq \bigvee_{i \in I \setminus I'} t_i) \vee (\bigwedge_{i \in I'} s_i \leq s'_j) \end{array}$$

Définition

Definition 2.3.4.1. *Un modèle est un prémodèle dans lequel, pour toutes familles finies de* **arrow***-types atomiques* $(t_i \to s_i)_{i\in I}$ *et* $(t'_j \to s'_j)_{j\in J}$ *:*

$$\begin{array}{rl} & \bigwedge_{i\in I} t_i \to s_i \leq \bigvee_{j\in J} t'_j \to s'_j \\ \Leftrightarrow & \exists j \in J. \forall I' \subseteq I.\ (t'_j \leq \bigvee_{i\in I\setminus I'} t_i) \vee (\bigwedge_{i\in I'} s_i \leq s'_j) \end{array}$$

Il n'est pas évident avec cette définition qu'il existe des modèles. C'est bien le cas :

Théorème 2.3.4.2. *Il existe un modèle.*

Ce résultat sera prouvé plus loin (section 4.1), en construisant un modèle dans lequel les fonctions sont modélisées par des graphes relationnels finis.

De même que dans l'univers des produits nous avons l'opérateur π_1, dans l'univers des types flèche, nous disposons d'un opérateur d'application entre les **arrow**-types et les types (notation : $t_{\mathbf{arrow}}.t$) ; il est défini modulo $\simeq$ par la caractérisation sémantique suivante : $t_{\mathbf{arrow}}.t$ est la plus petite solution s de l'inéquation $t_{\mathbf{arrow}} \leq t \to s$. Cela est détaillé dans la section B.3.

Nous constatons en particulier sur la formule donnant cet opérateur que lorsque que l'on prend un $t_{\mathbf{arrow}}$ de la forme $(\bigwedge t_i \to s_i)$, alors :

$$t_{\mathbf{arrow}}.t \not\simeq 1 \Rightarrow t \leq \bigvee_i t_i$$

Cette remarque sera utilisée pour supprimer la plupart des tests dynamiques de types dans les applications lors de l'exécution des programmes.

Dans l'analyse intuitive du problème, nous avons utilisé une notion d'application ; pour la définition formelle de modèle, on n'en a pas besoin. On verra néanmoins que les modèles que nous étudierons ont une telle notion d'application, qui vérifie l'égalité utilisée pour caractériser $[\![t_{\mathbf{arrow}}.t]\!]$:

Definition 2.3.4.3. *Une notion d'application* $\_.\_ : \mathcal{D}_{\mathbf{arrow}} \times \mathcal{D} \longrightarrow \mathcal{P}(\mathcal{D})$ *est fidèle si :*

$$[\![t_{\mathbf{arrow}}.t]\!] = \bigcup_{e_1 \in [\![t_{\mathbf{arrow}}]\!], e_2 \in [\![t]\!]} e_1.e_2$$

2.4 Filtrage

Une opération essentielle dans notre langage est le filtrage, qui permet en particulier de diriger le calcul suivant le type d'une valeur, et donc d'implémenter la surcharge. L'algèbre de motifs que l'on choisit est largement indépendante du langage, et il n'y a pas d'interaction avec les types flèche. En fait, ce qui compte, c'est l'algèbre de types elle-même, et plus précisément la structure des types produit. Nous pouvons formuler l'opération de filtrage dans n'importe quel prémodèle. Dans cette partie, nous travaillons donc dans un prémodèle arbitraire.

2.4.1 Motifs : syntaxe et sémantique informelle

Definition 2.4.1.1. *On fixe un ensemble d'identificateurs de variables* $\mathbb{V}$*, notées* $x, y, \ldots$ *On définit les motifs* $\mathbb{P}$ *comme les termes éventuellement* **infinis** *engendrés par les productions suivantes, avec les conditions de bonne formation :*

1. *sur toute branche infinie, il y a un nombre infini de nœuds couple ;*
2. *si on note $V(p) \subseteq X$ l'ensemble des variables qui apparaissent dans p, on a, pour toute sous-expression $p_1 \wedge p_2$ de p : $V(p_1) \cap V(p_2) = \varnothing$ et pour toute sous-expression $p_1|p_2$: $V(p_1) = V(p_2)$;*
3. *il n'y a qu'un nombre fini de sous-termes motifs distincts.*

$$
\begin{array}{llll}
p & ::= & & \textit{motifs} \\
 & & x & \textit{variable de capture, } x \in \mathbb{V} \\
 & | & t & \textit{contrainte de type} \\
 & | & p_1 \wedge p_2 & \textit{conjonction} \\
 & | & p_1|p_2 & \textit{alternative} \\
 & | & (p,\_) & \textit{couple 1} \\
 & | & (\_,p) & \textit{couple 2}
\end{array}
$$

L'ordre sur les motifs induit par $p_1 \wedge p_2 > p_1, p_2$ et $p_1|p_2 > p_1, p_2$ est bien fondé, compte-tenu de la première condition de bonne formation. Quand nous parlerons d'induction sur des motifs, nous ferons référence à cet ordre.

La première condition de bonne formation interdit les motifs tels que $p = t|p$ qui s'interprètent mal. La troisième condition permet de représenter les motifs par des expressions finies, de même que les types, et d'assurer la terminaison des algorithmes que nous allons présenter.

2.4.2 Sémantique du filtrage

Nous définissons le résultat du filtrage de la valeur v par le motif p. C'est un élément de $\{\Omega\} \cup \mathcal{D}^{V(p)}$, où Ω désigne un objet particulier qui représente l'échec du filtrage. Lorsque e/p n'est pas Ω, nous pouvons donc le voir comme une substitution de domaine $V(p)$, à valeur dans $\mathcal{D}$.

$$
\begin{array}{lcll}
e/t & = & \{\} & \text{si } e \in [\![t]\!] \\
e/t & = & \Omega & \text{si } e \in [\![\neg t]\!] \\
e/x & = & \{x \mapsto e\} & \\
e/p_1 \wedge p_2 & = & e/p_1 \cup e/p_2 & \text{si } (e/p_1 \neq \Omega) \wedge (e/p_2 \neq \Omega) \\
e/p_1 \wedge p_2 & = & \Omega & \text{si } (e/p_1 = \Omega) \vee (e/p_2 = \Omega) \\
e/p_1|p_2 & = & e/p_1 & \text{si } e/p_1 \neq \Omega \\
e/p_1|p_2 & = & e/p_2 & \text{si } e/p_1 = \Omega \\
(e_1,e_2)/(p,\_) & = & e_1/p & \\
e/(p,\_) & = & \Omega & \text{si } e \notin \mathcal{D}_{\mathbf{product}} \\
(e_1,e_2)/(\_,p) & = & e_2/p & \\
e/(\_,p) & = & \Omega & \text{si } e \notin \mathcal{D}_{\mathbf{product}}
\end{array}
$$

Ces égalités constituent bien une définition de e/p, par induction sur le couple (e, p) ordonné lexicographiquement.

2.4.3 Type accepté par un motif

Théorème 2.4.3.1. *Pour tout motif p, il existe un type $\wr p \int$ tel que :*

$$[\![\wr p \int]\!] = \{e \mid e/p \neq \Omega\}$$

Démonstration. Notons $\lbag p \rbag$ le membre droit de l'égalité. Des calculs simples donnent :

$$
\begin{array}{lcl}
\lbag x \rbag & = & \mathcal{D} \\
\lbag t \rbag & = & [\![t]\!] \\
\lbag p_1 \wedge p_2 \rbag & = & \lbag p_1 \rbag \cap \lbag p_2 \rbag \\
\lbag p_1 | p_2 \rbag & = & \lbag p_1 \rbag \cup \lbag p_2 \rbag \\
\lbag (p,\_) \rbag & = & \lbag p \rbag \times \mathcal{D} \\
\lbag (\_,p) \rbag & = & \mathcal{D} \times \lbag p \rbag
\end{array}
$$

Il est naturel de définir $\pmb{\lbag} p \pmb{\rbag}$ par les formules :

$$
\begin{array}{lcl}
\pmb{\lbag} x \pmb{\rbag} & = & 1 \\
\pmb{\lbag} t \pmb{\rbag} & = & t \\
\pmb{\lbag} p_1 \wedge p_2 \pmb{\rbag} & = & \pmb{\lbag} p_1 \pmb{\rbag} \wedge \pmb{\lbag} p_2 \pmb{\rbag} \\
\pmb{\lbag} p_1 | p_2 \pmb{\rbag} & = & \pmb{\lbag} p_1 \pmb{\rbag} \vee \pmb{\lbag} p_2 \pmb{\rbag} \\
\pmb{\lbag} (p,\_) \pmb{\rbag} & = & \pmb{\lbag} p \pmb{\rbag} \times 1 \\
\pmb{\lbag} (\_,p) \pmb{\rbag} & = & 1 \times \pmb{\lbag} p \pmb{\rbag}
\end{array}
$$

Il est facile de voir, en utilisant les première et dernière conditions de bonne formation des motifs, que ces égalités définissent bien de manière unique des types, et que nous avons effectivement :

$$[\![\pmb{\lbag} p \pmb{\rbag}]\!] = \lbag p \rbag$$

$\square$

2.4.4 Filtrage des types

Si p est un motif et t un type tel que $t \leq \pmb{\lbag} p \pmb{\rbag}$ (autrement dit : $[\![t]\!] \subseteq \lbag p \rbag$), on dit que le couple (t, p) est compatible. Nous cherchons à dénoter par un type l'ensemble des résultats que l'on peut obtenir, pour une variable $x \in V(p)$ donnée, lorsque l'on filtre une valeur de type t par le motif p.

Théorème 2.4.4.1. *Pour tout couple compatible (t, p) et toute variable $x \in V(p)$, il existe un type $(t \Downarrow p)(x)$ tel que :*

$$[\![(t \Downarrow p)(x)]\!] = \{(e/p)(x) \mid e \in [\![t]\!]\}$$

Ces types sont effectivement calculables. Notons $(t \downarrow p)(x)$ le membre droit de l'égalité. Des calculs simples donnent :

$$
\begin{array}{lcll}
(t \downarrow x)(x) & = & t & \\
(t \downarrow p_1 \wedge p_2)(x) & = & (t \downarrow p_i)(x) & \text{si } x \in V(p_i) \\
(t \downarrow p_1 | p_2)(x) & = & (t \wedge \pmb{\lbag} p_1 \pmb{\rbag} \downarrow p_1)(x) \cup (t \wedge \neg \pmb{\lbag} p_1 \pmb{\rbag} \downarrow p_2)(x) & \\
(t \downarrow (p,\_))(x) & = & (\pi_1(t) \downarrow p)(x) & \\
(t \downarrow (\_,p))(x) & = & (\pi_2(t) \downarrow p)(x) &
\end{array}
$$

Si on remplace dans le membre droit la réunion par l'opérateur $\vee$, on obtient presque des égalités qui définissent des types. Ce n'est pas tout à fait le cas, à cause de la circularité potentielle dans les motifs. Nous nous en sortons grâce à la stabilité des opérateurs π_1 et π_2. Le théorème est prouvé dans la section B.4.

La caractérisation sémantique du théorème donne directement :

Fait 2.4.4.2. *L'opérateur $t \mapsto (t \Downarrow p)(x)$, pour p et x fixés, est croissant.*

Le type $\wr t \int$ ne dépend pas du choix du prémodèle. Par contre, $(t \Downarrow p)$ en dépend, à cause de l'intervention des opérateurs π_1 et π_2. Si deux prémodèles sont équivalents, ils ont les mêmes opérateurs π_1 et π_2, et donc ils donnent les mêmes résultats pour $(t \Downarrow p)$.

Chapitre 3

Le langage

3.1 Syntaxe

Donnons nous un ensemble d'opérateurs $\mathbb{O}$. L'ensemble $\mathbb{E}$ des expressions du langage est défini par les productions :

$$
\begin{array}{lllll}
\mathtt{e} & ::= & & & \text{expressions} \\
& & & \mathtt{c} & \text{constante, } \mathtt{c} \in \mathcal{D}_{\mathbf{basic}} \\
& & \mid & \mathtt{op}(\mathtt{e}) & \text{opérateur, } \mathtt{op} \in \mathbb{O} \\
& & \mid & x & \text{variable, } x \in \mathbb{V} \\
& & \mid & \mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathtt{e} & \text{abstraction} \\
& & \mid & (\mathtt{e}_1, \mathtt{e}_2) & \text{couple} \\
& & \mid & \text{match } \mathtt{e} \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2 & \text{filtrage, } p_1, p_2 \in \mathbb{P}
\end{array}
$$

La construction d'abstraction lie les variables f et x dans son corps $\mathtt{e}$; pour chaque branche du filtrage, le motif p_i lie ses variables $V(p_i)$ dans le corps $\mathtt{e}_i$. Les motifs ont été définis dans le chapitre précédent.

Nous raisonnons toujours implicitement modulo α-conversion.

Nous supposons qu'il existe dans $\mathbb{O}$ un opérateur $\mathtt{apply}$ et nous notons $\mathtt{e}_1 \mathtt{e}_2$ pour $\mathtt{apply}((\mathtt{e}_1, \mathtt{e}_2))$.

3.2 Système de types

Pour définir le système de types, nous fixons un modèle, dit modèle d'amorce ; il définit la relation de sous-typage, l'opérateur d'application sur les types $\_ . \_$, et les opérations de filtrage. Dans toute cette partie, lorsque nous utiliserons la notation générique pour les interprétations $[\![\_]\!]$, nous ferons référence à celle du modèle d'amorce.

Nous allons utiliser tout cela pour définir un jugement de typage $\Gamma \vdash \mathtt{e} : t$ où Γ est un environnement de typage (une fonction partielle des variables dans les types).

Pour chaque opérateur $\mathtt{op}$, nous supposons donné un type $t_{\mathtt{op}}$, et une fonction qui associe à tout sous-type t de $t_{\mathtt{op}}$ un type noté $\mathtt{op}(t)$. Nous supposons que cette fonction est croissante : si $t_1 \leq t_2 \leq t_{\mathtt{op}}$, alors $\mathtt{op}(t_1) \leq \mathtt{op}(t_2)$.

Pour les opérateur qui sont intuitivement binaires, $t_{\mathtt{op}}$ sera de la forme $t^1_{\mathtt{op}} \times t^2_{\mathtt{op}}$ et $\mathtt{op}(t)$ de la forme $\mathtt{op}'(\pi_1(t), \pi_2(t))$.

Nous prenons $t_{\mathtt{apply}} = 1_{\mathbf{arrow}} \times 1$ et $\mathtt{apply}(t) = \pi_1(t).\pi_2(t)$.

Pour chaque constante c, nous supposons donné un **basic**-type t_{c} tel que $c \in [\![t_{\mathsf{c}}]\!]$ et pour tout type t, $t < t_{\mathsf{c}} \Rightarrow t \simeq 0$ (ou de manière équivalente : $t_{\mathsf{c}} \not\leq t \Rightarrow t_{\mathsf{c}} \leq \neg t$). Remarquons que cette propriété ne dépend pas du modèle d'amorce.

Remarque 12. *Nous attribuons donc à chaque constante un type de base minimal (le plus précis possible) ; une condition suffisante pour vérifier la condition est d'être un type singleton (c'est-à-dire que $[\![t_{\mathsf{c}}]\!] = \{\mathsf{c}\}$), mais ce n'est pas nécessaire.*

3.2.1 Règles de typage

Pour un couple compatible (t, p), $(t \Downarrow p)$ peut être vu comme un environnement de typage de domaine $V(p)$. La concaténation de deux environnements est notée $\Gamma_1 + \Gamma_2$. La convention d'α-conversion implicite permet en particulier de supposer que les domaines de Γ_1 et Γ_2 sont disjoints lorsque nous écrivons $\Gamma_1 + \Gamma_2$.

$$\frac{}{\Gamma \vdash \mathsf{c} : t_{\mathsf{c}}}\ (const) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Gamma \vdash \mathsf{e} : t \leq t_{\mathsf{op}}}{\Gamma \vdash \mathsf{op}(\mathsf{e}) : \mathsf{op}(t)}\ (op)$$

$$\frac{\begin{array}{l} (\forall 1 \leq i \leq n).\ \Gamma + (x : t_i) + (f : t) \vdash \mathsf{e} : s_i^0 \leq s_i \\ (\forall 1 \leq j \leq m).\ t \not\leq t'_j \to s'_j \\ t = (t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n) \\ t' = \neg(t'_1 \to s'_1) \wedge \ldots \wedge \neg(t'_m \to s'_m) \end{array}}{\Gamma \vdash \mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e} : t \wedge t'}\ (abstr)$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : t_1 \quad \Gamma \vdash \mathsf{e}_2 : t_2}{\Gamma \vdash (\mathsf{e}_1, \mathsf{e}_2) : t_1 \times t_2}\ (couple)$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathsf{e} : s \leq \wr p_1 \int \vee \wr p_2 \int \\ s \wedge \wr p_1 \int \simeq 0 \Rightarrow t_1 = 0 \\ s \wedge \wr p_1 \int \not\simeq 0 \Rightarrow \Gamma + ((s \wedge \wr p_1 \int) \Downarrow p_1) \vdash \mathsf{e}_1 : t_1 \\ s \leq \wr p_1 \int \Rightarrow t_2 = 0 \\ s \not\leq \wr p_1 \int \Rightarrow \Gamma + ((s \wedge \neg \wr p_1 \int) \Downarrow p_2) \vdash \mathsf{e}_2 : t_2 \end{array}}{\Gamma \vdash \text{match } \mathsf{e} \text{ with } p_1 \to \mathsf{e}_1 \mid p_2 \to \mathsf{e}_2 : t_1 \vee t_2}\ (match)$$

$$\frac{\Gamma \vdash \mathsf{e} : s \leq t}{\Gamma \vdash \mathsf{e} : t}\ (subsomption)$$

3.2.2 Commentaires

Il y a une règle par construction syntaxique pour les expressions, plus la règle de subsomption (*subsomption*). Les règles (*const*), (*couple*), (*var*), (*subsomption*) sont tout à fait classiques.

Règle (op) Elle est facile à comprendre ; elle permet de typer très précisément l'application des opérateurs, ce qui permet d'avoir une certaine forme faible d'interprétation abstraite. Par exemple, si l'on a les types intervalles d'entiers et un opérateur $+$, on peut se débrouiller pour avoir $+([n_1..n_2] \times [m_1..m_2]) = [n_1 + m_1..n_2 + m_2]$.

Règle $(abstr)$ Si l'on enlève t' dans la règle $(abstr)$, on retrouve le typage classique des abstractions dans un λ-calcul simplement typé. Ici, il faut typer le corps une fois pour chaque type flèche donné dans l'interface. Cependant, il est très important d'autoriser l'ajout des types flèche négatifs. En effet, nous avons besoin, pour définir la sémantique opérationnelle que les valeurs du langage (définies plus bas, et dont font partie les abstractions bien typées) aient, pour tout type t, soit le type t, soit le type $\neg t$. Nous verrons qu'ajouter les types flèche négatifs dans la règle $(abstr)$ permet d'avoir cette propriété.

Remarque 13. *Ces types flèche négatifs sont importants pour l'étude formelle du système, et sont indispensables pour la sémantique opérationnelle. Par contre, ils n'apportent pas grand chose d'intéressant au programmeur : savoir qu'une fonction q le type $\neg(t \to s)$ n'apporte pas d'information lorsqu'on l'applique à un argument (pour le voir, il suffit d'utiliser la caractérisation sémantique de l'opérateur d'application sur les types, comme plus petite solution d'une inéquation, et d'utiliser la propriété de disjonction faible). Pour obtenir un algorithme de typage à partir du système, nous supprimerons donc ces types négatifs. Pour la sémantique opérationnelle, il suffit de savoir dire si une* valeur *(l'ensemble des valeurs est une sous-classe de celle des expressions) a un type donné ou non, et cela, nous pourrons le faire facilement.*

Remarque 14. *Notons que grâce à la règle de subsomption, on peut remplacer dans les prémisses de la règle $(abstr)$ les inégalités par des égalités. Nous les laissons pour faciliter l'énoncé du résultat d'élimination de la subsomption.*

Règle $(match)$ Elle doit se lire de la manière suivante. On commence par obtenir un type pour l'expression filtrée et les motifs. La condition d'exhaustivité de filtrage est exprimée par $s \leq \wr p_1 \int \vee \wr p_2 \int$. Le type du résultat du filtrage est la réunion des résultats de chaque branche. Pour la deuxième branche, on sait non seulement que l'expression filtrée a le type s, mais aussi qu'elle n'a pas le type $\wr p_1 \int$, sinon la première branche aurait réussi. Comme à l'exécution, l'expression filtrée aura été réduite à une valeur, et comme nous verrons qu'une valeur qui n'a pas le type $\wr p_1 \int$ a le type $\neg \wr p_1 \int$, on peut mettre comme hypothèse, pour typer le résultat de la deuxième branche, que l'expression a le type $s \wedge \neg \wr p_1 \int$.

Lorsque le filtrage de la première branche ne peut pas réussir (c'est-à-dire $s \wedge \wr p_1 \int \simeq 0$), il ne faut pas tenir compte du résultat t_1, et on prend à la place $t_1 = 0$. De même pour la seconde branche. Cela est important pour définir le corps d'une fonction surchargée avec un filtrage. Le schéma classique pour définir une fonction surchargée de types $t_1 \to s_1$ et $t_2 \to s_2$, avec par exemple $t_1 \wedge t_2 \simeq 0$ pour fixer les idées, c'est de commencer par un filtrage :

$$\mu f : (t_1 \to s_1; t_2 \to s_2).\lambda x.\text{match } x \text{ with } x \wedge t_1 \to \mathtt{e}_1 \mid x \wedge t_2 \to \mathtt{e}_2$$

avec $(x : t_i) \vdash \mathtt{e}_i : s_i$. Lorsque que l'on type le corps de cette abstraction sous l'hypothèse $(x : t_2)$, on sait que la première branche est inutile, et ne doit pas contribuer au type du résultat ; cela dit, rien ne garantit que $(x : 0) \vdash \mathsf{e}_1 : 0$, par exemple si e_1 est une constante. Il faut donc explicitement, dans ce cas, ne pas tenir compte du type de e_1, sinon on ne peut pas prouver que le résultat est de type s_2.

Remarque 15. *Une autre approche aurait été de considérer la règle plus simple pour le filtrage :*

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{e} : s \leq \wr p_1 \int \vee \wr p_2 \int \\ \Gamma + ((s \wedge \wr p_1 \int) \Downarrow p_1) \vdash \mathsf{e}_1 : t_1 \\ \Gamma + ((s \wedge \neg \wr p_1 \int) \Downarrow p_2) \vdash \mathsf{e}_2 : t_2\end{array}}{\Gamma \vdash \mathit{match}\ \mathsf{e}\ \mathit{with}\ p_1 \to \mathsf{e}_1 \mid p_2 \to \mathsf{e}_2 : t_1 \vee t_2}\ (\mathit{match}')$$

et de rajouter la règle :

$$\frac{\Gamma(x) \simeq 0}{\Gamma \vdash \mathsf{e} : 0}\ (\mathit{zero})$$

Pour comprendre cette règle, il faut garder en tête que la sémantique que l'on propose est stricte (dans la mesure où les types deviennent plus précis à l'exécution et où les calculs sont dirigés de manière essentielle par les types, cela n'est pas étonnant), et nous verrons, ce qui est naturel, qu'il n'y a pas de valeur dans le type vide. Donc si l'on type une expression avec dans l'environnement une liaison $(x : 0)$, on sait que cette expression ne sera jamais évaluée à l'exécution, et on peut donc lui donner n'importe quel type (0 suffit grâce à la subsomption).

Remarquons que pour conserver le théorème de substitution, il faut le restreindre au cas où l'on substitue des valeurs aux variables.

Si l'on suppose qu'il n'y a pas d'abstraction avec un type flèche $t_i \to s_i$, $t_i \simeq 0$ dans l'interface, le système obtenu est équivalent.

Le cas intéressant est bien celui des variables introduites par la règle (match) dans l'environnement, et cela explique le choix de la présentation.

Remarque 16. *Le typage des filtrages garantit leur exhaustivité. Il est également possible de vérifier qu'aucune branche n'est inutile, ce qui est un signe probable d'erreur dans le programme (par exemple, si le motif de la première branche est mal écrit, il peut cacher la deuxième branche). Pour cela, il faut faire attention au fait que les expressions peuvent être typées plusieurs fois (à cause de la règle de typage des abstractions), et il suffit qu'une branche soit utilisée lors d'un de ces typages pour qu'elle ne soit pas inutile.*

3.3 Propriétés syntaxiques du système

Nous allons énoncer dans cette section des propriétés syntaxiques simples sur le système de types. Les arguments des preuves se trouvent en annexe B.5

Lemme 3.3.0.1 (Renforcement des hypothèses). *Soient Γ_1, Γ_2 deux environnements de typage de même domaine et tels que pour tout x, $\Gamma_2(x) \leq \Gamma_1(x)$. Alors tout jugement dérivable avec Γ_1 est dérivable avec Γ_2.*

La règle de subsomption montre que le système de types est sémantique dans le sens suivant : le fait de pouvoir obtenir $\Gamma \vdash \mathsf{e} : t$ ne dépend que de $[\![t]\!]$. Comme elle introduit beaucoup de non déterminisme dans les règles, il est plus agréable d'étudier le système sans cette règle ; nous allons montrer que nous ne perdons rien.

Definition 3.3.0.2. *On note* $\Gamma \Vdash \mathsf{e} : t$ *le jugement défini par les mêmes règles que* $\vdash$*, moins la règle de subsomption.*

Théorème 3.3.0.3 (Élimination de la subsomption). *Si* $\Gamma \vdash \mathsf{e} : t$*, alors il existe* $t' \leq t$ *tel que* $\Gamma \Vdash \mathsf{e} : t'$*.*

La réciproque est évidemment vraie.

Même sans la règle de subsomption, le système est non déterministe : une même expression peut avoir deux types t_1 et t_2. Nous allons prouver qu'elle a alors le type plus précis $t_1 \wedge t_2$, comme on pouvait l'espérer.

Théorème 3.3.0.4 (Conjonction). *La règle suivante est admissible dans le système :*

$$\frac{\Gamma \vdash \mathsf{e} : t_1 \quad \Gamma \vdash \mathsf{e} : t_2}{\Gamma \vdash \mathsf{e} : t_1 \wedge t_2}$$

Rappelons qu'une règle est dite admissible si on peut l'ajouter au système sans changer l'ensemble des jugements dérivables (autrement dit si pour toutes ses instances, si les prémisses sont vraies, la conclusion l'est aussi).

La sémantique opérationnelle du langage utilise de manière essentielle l'opération de substitution pour définir les calculs. Pour relier le système de types et cette sémantique, et ainsi obtenir des résultats de sûreté, nous devons étudier le comportement du système de types vis-à-vis de la substitution. Nous avons, de manière tout à fait classique :

Théorème 3.3.0.5 (Substitution). *La règle suivante est admissible dans le système :*

$$\frac{\Gamma \vdash \mathsf{e}_1 : s \quad \Gamma + (x : s) \vdash \mathsf{e}_2 : t}{\Gamma \vdash \mathsf{e}_2[\mathsf{e}_1/x] : t}$$

3.4 Le modèle des valeurs

3.4.1 Structure

Nous distinguons parmi les expressions **closes** et **bien typées** les valeurs :

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

Notons $\mathcal{V}$ l'ensemble des valeurs, $\mathcal{V}_{\mathbf{basic}}$ celles qui sont des constantes, $\mathcal{V}_{\mathbf{product}}$ celles qui sont des couples, $\mathcal{V}_{\mathbf{arrow}}$ celles qui sont des abstractions.

L'ensemble des valeurs peut être vu comme une structure ; quand nous parlerons d'induction sur les valeurs, nous utiliserons l'ordre bien fondé de cette structure (celui engendré par $(v_1, v_2) > v_1$, $(v_1, v_2) > v_2$).

3.4.2 Le type vide n'est pas habité

Le lemme suivant dit que le type vide n'est pas habité par des valeurs :

Lemme 3.4.2.1. *Si* $\vdash v : t$*, alors* $t \not\simeq 0$*.*

Démonstration. Il suffit de prouver que $\Vdash v : t \Rightarrow t \not\simeq 0$. Nous travaillons par induction sur v et nous regardons la dernière règle utilisée pour prouver ce jugement. Pour les constantes, c'est simplement l'hypothèse $t_{\mathsf{c}} \not\simeq 0$. Pour les couples, l'hypothèse de récurrence s'applique directement. Le cas délicat est celui des abstractions. Supposons $\Vdash \mathsf{e}_0 = \mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e} : t \wedge \neg t'$; $t = \bigwedge t_i \to s_i$ et $t' = \bigvee (t'_j \to s'_j)$, avec $t \not\leq t'_j \to s'_j$. La propriété de disjonction forte montre que $t \not\leq t'$ et donc $t \wedge \neg t' \not\simeq 0$. □

Par contre, le type vide est habité par certaines expressions dont l'évaluation ne peut pas terminer. Par exemple, pour n'importe quelle expression e bien typée, $(\mu f : (1 \to 0).\lambda x.fx)\mathsf{e}$ a le type 0. Retenons :

Lemme 3.4.2.2. *Il existe une expression close* e *telle que* $\vdash \mathsf{e} : 0$.

3.4.3 Interprétation

Nous allons maintenant définir sur la structure des valeurs une interprétation pour en faire un prémodèle, dont nous notons $\mathbb{V}[\![\_]\!]$ la fonction d'interprétation. Il suffit de définir l'interprétation des types flèche :

$$\mathbb{V}[\![t \to s]\!] = \{\mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e} \in \mathcal{V}_{\mathbf{arrow}} \mid (\bigwedge_{i=1}^{n} t_i \to s_i) \leq t \to s\}$$

3.4.4 Le prémodèle des valeurs est compatible avec le système de types

Le prémodèle donne une certaine notion de typage pour les valeurs : v est dans le type t si $v \in \mathbb{V}[\![t]\!]$. Le système de types permet lui aussi de définir une telle notion, les valeurs étant des expressions du langage. Nous montrons en annexe B.6 que ces deux notions coïncident.

Théorème 3.4.4.1 (Typage des valeurs). *Soit t un type. Alors :*

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

Corollaire 3.4.4.2.
- $\vdash v : t_1 \wedge t_2 \Leftrightarrow (\vdash v : t_1) \wedge (\vdash v : t_2)$
- $\vdash v : t_1 \vee t_2 \Leftrightarrow (\vdash v : t_1) \vee (\vdash v : t_2)$
- $\vdash v : \neg t \Leftrightarrow \not\vdash v : t$

Remarque 17. *Nous avons déjà vu que l'équivalence pour les intersections est vraie en fait pour toutes les expressions, pas seulement les valeurs. C'est faux pour les deux autres équivalences. Par exemple, considérons l'expression* $(\mu f : (t \to t \vee s).\lambda x.x)\mathsf{e}$ *où* $\vdash \mathsf{e} : t$ *; elle a le type $t \vee s$, mais ni le type t, ni le type s en général.*

Remarque 18. *Dans une remarque précédente, nous avons signalé la possibilité d'ajouter la règle suivante au système :*

$$\frac{\Gamma(x) = 0}{\Gamma \vdash \mathsf{e} : 0}\ (zero)$$

La justification est qu'il n'y a pas de valeur de type 0*, et donc que l'expression* e *ne sera jamais évaluée. Avec le théorème de typage des valeurs, on voit qu'il serait sûr de considérer également la règle :*

$$\frac{\Gamma + (x : t_1) \vdash \mathsf{e} : t \quad \Gamma + (x : t_2) \vdash \mathsf{e} : t}{\Gamma + (x : t_1 \vee t_2) \vdash \mathsf{e} : t} \; (union)$$

En effet, avec la sémantique stricte, lorsque e *sera exécutée,* x *aura été remplacé par une valeur de type* $t_1 \vee t_2$*, donc de type* t_1 *ou de type* t_2*.*

Encore une fois, pour conserver le théorème de substitution, il faut le restreindre au cas où l'on substitue des valeurs aux variables.

Cette règle est bien entendu difficile à intégrer dans un algorithme de typage, et l'on a donc choisi de ne pas la considérer. Néanmoins, elle peut être intéressante en pratique (elle correspond à faire une preuve par disjonction de cas) ; par exemple, si l'on a comme types de base les intervalles d'entiers, avec un typage fin des opérateurs arithmétiques, on peut par exemple prouver que tout carré est positif, en distinguant suivant les cas $x \leq 0$*,* $x \geq 0$*. La solution, en général, c'est d'utiliser un filtrage pour distinguer explicitement les cas (avec du sucre syntaxique, cela revient à annoter le programme pour indiquer où appliquer la règle (union)).*

3.4.5 Le modèle des valeurs

Le type vide n'étant pas habité par les valeurs, nous avons : $[\![t]\!] = \varnothing \Rightarrow \mathbb{V}[\![t]\!] = \varnothing$. Nous prouvons en annexe B.7 la réciproque, ce qui donne :

Théorème 3.4.5.1. *Soit* t *un type. Alors :*

$$\mathbb{V}[\![t]\!] = \varnothing \Leftrightarrow [\![t]\!] = \varnothing$$

et nous en déduisons tout de suite :

Corollaire 3.4.5.2. *Soient* t, s *deux types. Les assertions suivantes sont équivalentes :*

- $t \leq s$
- $\forall v.\ (\vdash v : t \Rightarrow \vdash v : s)$
- $\forall \Gamma. \forall \mathsf{e}.\ (\Gamma \vdash \mathsf{e} : t \Rightarrow \Gamma \vdash \mathsf{e} : s)$
- $\mathbb{V}[\![t]\!] \subseteq \mathbb{V}[\![s]\!]$

Discussion Le prémodèle des valeurs est donc équivalent au modèle d'amorce. Nous avons défini dans le cadre général des prémodèles la notion de filtrage. Nous utilisons la notation générique v/p pour désigner le résultat du filtrage de la valeur v par le motif p. C'est soit Ω, soit une fonction $V(p) \to \mathcal{V}$. Les types $\wr p \int$ et $(t \Downarrow p)(x)$ (lorsque $t \leq \wr p \int$ et $x \in V(p)$) sont les mêmes dans le modèle d'amorce et dans le modèle du langage, c'est-à-dire :

$$\vdash v : \wr p \int \Leftrightarrow v/p \neq \Omega$$

et

$$\vdash v : (t \Downarrow p)(x) \Leftrightarrow \exists v'.\ (\vdash v' : t) \wedge ((v'/p)(x) = v)$$

3.4.6 Algorithme de vérification de type pour les valeurs

Dans la sémantique opérationnelle (introduite dans la section suivante) nous avons besoin de vérifier si une valeur a un type donné (en particulier pour exécuter les filtrages).

Le théorème de typage des valeurs et son corollaire permettent de donner un algorithme pour décider si $\vdash v : t$, lorsque v est une valeur et t un type (on sait déjà en particulier que v est bien typée). Il suffit de savoir le décider lorsque t est un type atomique. Si v et t ne sont pas dans le même univers, par exemple si v est un couple, et t est un **arrow**-type atomique, on ne peut pas avoir $\vdash v : t$.

Si $v = (v_1, v_2)$ et $t = t_1 \times t_2$, on doit vérifier $(\vdash v_1 : t_1) \wedge (\vdash v_2 : t_2)$, ce qui ramène à des instances plus petites du problème.

Dans le cas où $v = \mu f : (t_1 \to s_1; \dots; t_n \to s_n).\lambda x.\mathtt{e}$ et t est un **arrow**-type atomique, il faut vérifier que $(t_1 \to s_1) \wedge \dots \wedge (t_n \to s_n) \leq t$ (on sait déjà que v est bien typée, donc il n'y a pas de vérification à faire avec $\mathtt{e}$). Les types négatifs que la règle de typage permet d'ajouter n'interviennent pas, en raison de la propriété de disjonction forte pour les types flèche.

Lorsque $v = c$, il suffit de regarder si $t_c \leq t$.

3.5 Sémantique opérationnelle

Nous supposons donnée pour chaque opérateur $\mathtt{op}$ une relation binaire $\xrightarrow{\mathtt{op}} \subseteq \mathbb{V}[\![t_{\mathtt{op}}]\!] \times \mathbb{E}$ telle que si $\vdash v : t$, $t \leq t_{\mathtt{op}}$ et $v \xrightarrow{\mathtt{op}} \mathtt{e}$, alors : $\vdash \mathtt{e} : \mathtt{op}(t)$. Nous supposons également que pour toute valeur v telle que $\vdash v : t_{\mathtt{op}}$, il existe $v \xrightarrow{\mathtt{op}} \mathtt{e}$. Nous notons **wrong** un objet spécial qui représente l'erreur de type, et qui n'est pas une expression du langage (*a fortiori* pas une valeur).

Nous allons définir une sémantique opérationnelle à petits pas. Les règles de réduction à la racine sont (dans la colonne de droite, les conditions de bord) :

$$\begin{array}{lcl|l}
\mathtt{op}(v) & \to & \mathtt{e} & \vdash v : t_{\mathtt{op}}, v \xrightarrow{\mathtt{op}} \mathtt{e} \\
\text{match } v \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2 & \to & \mathtt{e}_1[v/p_1] & v/p_1 \neq \Omega \\
\text{match } v \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2 & \to & \mathtt{e}_2[v/p_2] & v/p_1 = \Omega, v/p_2 \neq \Omega \\
\hline
\mathtt{op}(v) & \to & \textbf{wrong} & \not\vdash v : t_{\mathtt{op}} \\
\text{match } v \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2 & \to & \textbf{wrong} & v/p_1 = \Omega, v/p_2 = \Omega
\end{array}$$

Les contextes d'évaluation sont, de manière tout à fait classique, des expressions avec trous :

$$C[\,] ::= [\,] \mid \mathtt{op}([\,]) \mid ([\,], \mathtt{e}) \mid (\mathtt{e}, [\,]) \mid (\text{match } [\,] \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2)$$

On ajoute les réductions $C[\mathtt{e}_1] \to C[\mathtt{e}_2]$ lorsque $\mathtt{e}_1 \to \mathtt{e}_2$, et de même en remplaçant $\mathtt{e}_2$ par **wrong** ; on ajoute aussi $C[\textbf{wrong}] \to \textbf{wrong}$.

Pour définir $\xrightarrow{\mathtt{apply}}$, nous choisissons une valeur particulière **reject** $\in \mathcal{V}$ et nous prenons :

$$\begin{array}{lcl|l}
(\mathtt{f}, v) & \xrightarrow{\mathtt{apply}} & \mathtt{e}[v/x; \mathtt{f}/f] & \exists i.\ \vdash v : t_i \\
(\mathtt{f}, v) & \xrightarrow{\mathtt{apply}} & \textbf{reject} & \forall i.\ \not\vdash v : t_i
\end{array}$$

lorsque $\mathtt{f} = (\mu f : (t_1 \to s_1; \dots; t_n \to s_n).\lambda x.\mathtt{e})$. Avec le théorème de substitution, nous voyons que cette relation vérifie bien les hypothèses pour les opérateurs (lorsque que l'on est dans le second cas, $\vdash v : \bigwedge \neg t_i$ et $(\bigwedge t_i \to s_i).(\bigwedge \neg t_i) = 1$).

Nous pouvons déjà énoncer un lemme facile à établir :

Lemme 3.5.0.1. *Si* $\vdash \mathtt{e} : t$ *et* $\mathtt{e}$ *ne se réduit pas, alors* $\mathtt{e} \in \mathcal{V}$.

3.5.1 Sûreté du typage

Théorème 3.5.1.1 (Subject reduction). *Si* $\vdash \mathtt{e} : t$*, alors :* $\mathtt{e} \not\xrightarrow{*} \mathbf{wrong}$ *et si* $\mathtt{e} \xrightarrow{*} \mathtt{e}'$*, alors* $\vdash \mathtt{e}' : t$.

Corollaire 3.5.1.2. *Si* $\vdash \mathtt{e} : t$*,* $\mathtt{e} \xrightarrow{*} \mathtt{e}'$ *et* $\mathtt{e}'$ *ne se réduit pas, alors* $\mathtt{e}' \in \mathcal{V}$.

Démonstration. Le corollaire est une conséquence immédiate du théorème et du lemme ci-dessus. Prouvons le théorème. Il suffit de considérer une seule étape de réduction. On peut supposer $\Vdash \mathtt{e} : t$ et dans ce cas, la structure de la dérivation de typage correspond à celle de $\mathtt{e}$. Si nous montrons le résultat pour les réductions à la racine, nous aurons donc directement le cas général (la réduction correspond à récrire une sous-expression, et le résultat pour les réductions à la racine montre que l'on peut « corriger » en parallèle la dérivation de typage). Regardons donc les différents cas de réduction $\mathtt{e} \longrightarrow \mathtt{e}'$, et considérons la dernière règle utilisée dans la dérivation de $\Vdash \mathtt{e} : t$. Par exemple, traitons les cas où $\mathtt{e} = \text{match } v \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2$, et regardons l'instance de la règle $(match)$ utilisée :

$$\frac{\begin{array}{l}\Gamma \vdash v : s \leq \{\!\!\{p_1\}\!\!\} \vee \{\!\!\{p_2\}\!\!\} \\ s \wedge \{\!\!\{p_1\}\!\!\} \simeq 0 \Rightarrow t_1 = 0 \\ s \wedge \{\!\!\{p_1\}\!\!\} \not\simeq 0 \Rightarrow ((s \wedge \{\!\!\{p_1\}\!\!\}) \Downarrow p_1) \vdash \mathtt{e}_1 : t_1 \\ s \leq \{\!\!\{p_1\}\!\!\} \Rightarrow t_2 = 0 \\ s \not\leq \{\!\!\{p_1\}\!\!\} \Rightarrow ((s \wedge \neg \{\!\!\{p_1\}\!\!\}) \Downarrow p_2) \vdash \mathtt{e}_2 : t_2\end{array}}{\Gamma \vdash \text{match } v \text{ with } p_1 \to \mathtt{e}_1 \mid p_2 \to \mathtt{e}_2 : t = t_1 \vee t_2}\ (match)$$

Si $v/p_1 \neq \Omega$, c'est que $\vdash v : \{\!\!\{p_1\}\!\!\}$ et donc $s \wedge \{\!\!\{p_1\}\!\!\} \not\simeq 0$. On a : $((s \wedge \{\!\!\{p_1\}\!\!\}) \Downarrow p_1) \vdash \mathtt{e}_1 : t_1$ et $\vdash (v/p_1)(x) : ((s \wedge \{\!\!\{p_1\}\!\!\}) \Downarrow p_1)$ (comme indiqué à la fin de la section 3.4.5). Le théorème de substitution donne bien $\vdash \mathtt{e}_1[v/p_1] : t_1 \leq t$.

Si $v/p_1 = \Omega$, c'est que $\vdash v : \neg \{\!\!\{p_1\}\!\!\}$, et donc, comme $\vdash v : \{\!\!\{p_1\}\!\!\} \vee \{\!\!\{p_2\}\!\!\}$, on a : $\vdash v : \{\!\!\{p_2\}\!\!\}$, ce qui empêche d'avoir $v/p_2 = \Omega$. En raisonnant alors comme dans le premier cas, nous obtenons $\vdash \mathtt{e}_2[v/p_2] : t_2 \leq t$. □

Remarque 19. *Les dernières règles, qui introduisent* **wrong** *ne sont donc pas utilisées dans la réduction d'un programme bien typé. La phase de vérification des types permet donc de les supprimer de l'évaluateur (ou du code compilé).*

On peut vouloir éliminer le test du type de l'argument lors de la réduction de l'opérateur `apply`. Il suffit de trouver une condition suffisante sur le typage de l'application pour qu'elle soit toujours satisfaite.

Nous avons vu, juste après la définition des modèles, que si : $t = (\bigwedge t_i \to s_i) \wedge \neg(\bigvee t'_j \to s'_j) \not\simeq 0$ alors :

$$t.s \not\simeq 1 \Rightarrow s \leq \bigvee_i t_i$$

Pendant le typage du programme, si l'on tombe sur une expression $\mathtt{e}_1\mathtt{e}_2$, avec $\Gamma \vdash \mathtt{e}_1 : t \leq 1_{\mathbf{arrow}}$, $\Gamma \vdash \mathtt{e}_2 : s$, et si $t_{\mathbf{arrow}}.s \not\simeq 1$, on peut marquer cette application de sorte à ne pas faire le test de type pour l'argument à l'exécution (par exemple, un compilateur peut générer un saut vers un point d'entrée spécial dans la fonction).

Remarque 20. *On peut choisir n'importe quelle valeur* **reject** *pour signaler l'erreur : si l'application est bien typée et qu'elle « échoue », le typage fait que le contexte s'attend à recevoir n'importe quoi. Il serait possible de donner pour chaque abstraction une valeur par défaut, renvoyée lorsqu'elle ne sait pas (ou ne veut pas) traiter son argument.*

3.5.2 L'application est fidèle

Nous allons munir le modèle des valeurs d'une notion d'application naturelle ($v_1 \in \mathcal{V}_{\mathbf{arrow}}$, $v_2 \in \mathcal{V}$) :

$$v_1.v_2 = \{v' \mid v_1 v_2 \xrightarrow{*} v'\}$$

Théorème 3.5.2.1. *Cette notion d'application est fidèle.*

Posons, si $t_{\mathbf{arrow}}$ est un **arrow**-type et t un type :

$$t_{\mathbf{arrow}} \bullet t = \bigcup_{v_1 \in \mathbb{V}[\![t_{\mathbf{arrow}}]\!]_{\mathbf{arrow}}, v_2 \in \mathbb{V}[\![t]\!]} v_1.v_2$$

Il s'agit de montrer que $\mathbb{V}[\![t_{\mathbf{arrow}}.t]\!] = t_{\mathbf{arrow}} \bullet t$. C'est fait dans l'annexe B.8.

Discussion Le fait que cette notion d'application soit fidèle justifie après coup la condition imposée dans la définition de modèle. En effet, dans les motivations intuitives que nous avions utilisées, nous avions supposé que le langage était assez puissant pour obliger à prendre $t.s$ égal à son majorant déjà obtenu. C'est ce que confirme ce résultat.

Remarquons que l'on a utilisé très peu de choses dans le langage. Nous obtenons la propriété amusante :

Fait 3.5.2.2. *Tout modèle est équivalent à un modèle muni d'une notion d'application fidèle.*

3.6 Typage algorithmique

Si l'on supprime la règle de subsomption, ce qui ne change pas l'ensemble des expressions bien typées, le système obtenu est presque dirigé par la syntaxe. La seule règle qui pose problème est celle pour le typage des abstractions. Si on la remplace par la règle suivante, on obtient effectivement un système dirigé par la syntaxe (qui type moins d'expressions), dans lequel il y a toujours élimination de la subsomption :

$$\frac{\begin{array}{l}(\forall i).\ \Gamma + (x : t_i) + (f : t) \vdash \mathsf{e} : s_i^0 \leq s_i \\ t = (t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)\end{array}}{\Gamma \vdash \mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e} : t}$$

Notons $\Vdash$ le système obtenu avec cette nouvelle règle, et sans la subsomption (et dans la règle (*match*), on prend bien sûr la première alternative dans les deux disjonctions lorsque c'est possible). Les règles donnent un algorithme qui décide si une expression est bien typée et calcule le cas échéant son unique type dans le nouveau système.

L'algorithme n'est évidemment pas complet vis-à-vis du système de règles initial.

Remarque 21. *Je pense qu'il est possible de représenter, en introduisant une notion de schéma de types, l'ensemble des types (pour le système complet) d'une expression par un schéma fini, et d'avoir un algorithme pour calculer ce schéma.*

Une autre approche, peut-être plus simple, pourrait convenir. Elle s'appuie sur le fait que les types flèche négatifs introduits par la règle (*abstr*) *n'interviennent pas dans le résultat de l'opérateur d'application _._ sur les types. Le seul moyen de les utiliser, c'est d'utiliser une contrainte de type dans un motif. On en a aussi besoin pour pouvoir satisfaire éventuellement les interfaces des abstractions. Si l'on suppose qu'à part l'opérateur* `apply`*, aucun autre ne « travaille » sur des types flèche (dans un sens à définir), il doit être possible de collecter tous les types flèche négatifs qui apparaissent explicitement dans le programme (dans les interfaces d'abstractions ou les motifs), ce qui en donne un nombre fini, et d'essayer de rajouter seulement ceux-là dans la règle* (*abstr*).

En pratique, cependant, la restriction imposée par le système restreint n'est pas gênante, et ne justifie probablement pas de compliquer ainsi l'algorithme de typage.

3.7 Remarque sur la condition de « covariance » du λ&- calcul

Dans le λ&-calcul [CGL95], une des conditions de bonne formation des fonctions surchargées est la suivante. Si une fonction surchargée a deux branches de types $t_1 \to s_1$ et $t_2 \to s_2$, alors : $t_1 \leq t_2 \Rightarrow s_1 \leq s_2$. Pour comprendre cette condition, il faut se rappeler que le type le plus précis d'un programme diminue lors de l'évaluation. Si, statiquement, on attribue à une expression le type t_2 et que par conséquent on déduit que le résultat de la fonction surchargée appliquée à cette expression a le type s_2, il est possible qu'à l'exécution, on se rende compte que l'expression a en fait le type plus précis t_1, et c'est donc la branche $t_1 \to s_1$ qui est utilisée, donnant un résultat de type s_1. Pour que le typage soit sûr, il faut que $s_1 \leq s_2$.

Dans notre langage, nous ne donnons aucune condition sur les types flèche donnés dans l'interface d'une fonction. Considérons l'expression $\mathtt{e}_0 = \mu f : (t_1 \longrightarrow s_1; \ldots; t_n \longrightarrow s_n).\lambda x.\mathtt{e}$, bien typée dans le contexte Γ et étudions ce qu'il se passe lorsque par exemple $t_1 \leq t_2$.

Pour le typage du corps, on doit avoir : $\Gamma + (x : t_1) \vdash \mathtt{e} : s_1$ et $\Gamma + (x : t_2) \vdash \mathtt{e} : s_2$. Le lemme de renforcement des hypothèses montre que l'on a aussi : $\Gamma + (x : t_1) \vdash \mathtt{e} : s_2$, et la règle de la conjonction permet de conclure : $\Gamma + (x : t_1) \vdash \mathtt{e} : s_1 \wedge s_2$.

Pour l'interface de la fonction, on peut remplacer $t_1 \longrightarrow s_1$ par $t_1 \longrightarrow s_1 \wedge s_2$, sans changer les typages possibles pour $\mathtt{e}_0$, ni sa sémantique. Il suffit de constater que :

$$(t_1 \to s_1) \wedge (t_2 \to s_2) \simeq (t_1 \to s_1 \wedge s_2) \wedge (t_2 \to s_2)$$

On a bien maintenant : $s_1 \wedge s_2 \leq s_2$. Autrement dit, il est toujours possible de récrire l'interface d'une abstraction de sorte que la condition de covariance du λ&-calcul soit vérifiée.

Remarque 22. *L'équivalence ci-dessus peut être prouvée par des règles algébriques élémentaires. Mais il est tout aussi simple de raisonner directement dans* *un modèle et on constate très rapidement que les deux membres expriment exactement la même contrainte lorsque* $t_1 \leq t_2$.

Notre étude permet donc de retrouver une condition posée *a priori* dans le λ&-calcul ; dans un sens, nous avons donné une justification sémantique à cette condition.

Chapitre 4

Modèle réduit et algorithme de sous-typage

4.1 Modèle réduit

Pour donner le système de types et la sémantique du langage, il faut disposer d'un modèle d'amorce. Nous allons construire un modèle extensionnel, où les fonctions sont modélisées par des graphes relationnels finis. Nous l'appelons modèle réduit, parce qu'il ressemble de l'extérieur (c'est-à-dire par la relation de sous-typage qu'il crée) à un modèle intéressant (celui du langage, où il y a des calculs), mais à l'intérieur ce n'est qu'un petit modèle inerte (sans sémantique dynamique).

4.1.1 Structure

Nous considérons la plus petite solution $\mathcal{D}$ de l'équation de domaine :

$$\mathcal{D} = \mathcal{D}_{\mathbf{basic}} + \mathcal{D}^2 + \mathcal{P}_f(\mathcal{D}^2)$$

Cela donne déjà une structure, avec $\mathcal{D}_{\mathbf{product}} = \mathcal{D}^2$ et $\mathcal{D}_{\mathbf{arrow}} = \mathcal{P}_f(\mathcal{D}^2)$

On peut construire $\mathcal{D}$ comme l'ensemble des termes *finis* engendrés par les productions :

$$\begin{array}{lll} e & ::= & & \\ & & c & c \in \mathcal{D}_{\mathbf{basic}} \\ & \mid & (e_1, e_2) & \\ & \mid & \{(e_1, e'_1), \ldots, (e_n, e'_n)\} & \end{array}$$

Les éléments de $\mathcal{D}$ sont simplement appelés termes du modèle, ou simplement termes lorsqu'il n'y a pas d'ambiguïté. Nous allons travailler avec l'ordre sous-terme qui est bien fondé (il est plus gros que l'ordre bien fondé dont on dispose dans toutes les structures, pour lequel les fonctions sont minimales ; ici, nous pouvons faire des inductions qui descendent dans le graphe des fonctions).

4.1.2 Interprétation

Nous voulons poser :

$$\begin{array}{rcl} [\![t_1 \to t_2]\!]_{\mathbf{arrow}} & \stackrel{\text{def}}{=} & \mathcal{P}_f\left(\complement_{\mathcal{D}^2}([\![t_1]\!] \times \complement_{\mathcal{D}}[\![t_2]\!])\right) \\ & = & \{\{(e_i, e'_i)\} \in \mathcal{D}_{\mathbf{arrow}} \mid \forall i.\ e_i \in [\![t_1]\!] \Rightarrow e'_i \in [\![t_2]\!]\} \end{array}$$

Pour voir que ceci constitue bien une définition, procédons comme dans le cas général des prémodèles (l'ordre bien fondé permet maintenant de « descendre » dans les fonctions) ; les équivalences suivantes définissent bien de manière unique les fonctions d'interprétation :

$$\begin{array}{lcl} e \in [\![t]\!] & \Leftrightarrow & \exists u.\ (e \in \mathcal{D}_u) \wedge (e \in [\![t_u]\!]_u) \\ e \in [\![t_u]\!]_u & \Leftrightarrow & \exists (P, N) \in t_u. \\ & & \quad (\forall a \in P.\ e \in [\![a]\!]_u) \wedge (\forall a \in N.\ e \notin [\![a]\!]_u) \\ (e_1, e_2) \in [\![t_1 \times t_2]\!]_{\mathbf{product}} & \Leftrightarrow & (e_1 \in [\![t_1]\!]) \wedge (e_2 \in [\![t_2]\!] \\ \{(e_i, e'_i)\} \in [\![t_1 \to t_2]\!]_{\mathbf{arrow}} & \Leftrightarrow & \forall i.\ e_i \in [\![t_1]\!] \Rightarrow e'_i \in [\![t_2]\!] \end{array}$$

Remarque 23. *Supposons que dans la définition du modèle réduit, on autorise les termes infinis tels que l'ordre induit par $(e_1, e_2) > e_1$, $(e_1, e_2) > e_2$ reste bien fondé. Alors on a toujours une structure, mais il n'y a pas d'interprétation qui valide les équivalences ci-dessus. En effet, considérons le terme infini (régulier même) $e = \{(e, x)\}$ où x est arbitraire, et le type $t = t \to 0$. Alors les équivalences donnent : $e \in [\![t]\!] \Leftrightarrow \neg(e \in [\![t]\!])$, contradiction. Cela suggère qu'il n'est pas facile de construire des modèles fondamentalement différents du modèle réduit ; si l'on autorise des graphes infinis (en largeur, mais avec une structure bien fondée), de cardinal borné pour éviter des problèmes de théorie des ensembles (par exemple, on peut prendre des graphes dénombrables), on obtient un prémodèle équivalent.*

4.1.3 Modèle

Pour voir que le prémodèle est en fait un modèle, il faut encore vérifier les conditions sur le sous-typage dans l'univers des types flèche.

Nous étudions le problème dans $\mathcal{D}_{\mathbf{arrow}}$

$$\bigcap_{1 \le i \le n} [\![t_i \to s_i]\!] \subseteq \bigcup_{1 \le j \le m} [\![t'_j \to s'_j]\!]$$

En utilisant les faits B.1.0.1 et B.1.0.2 (section B.1), on voit que le problème est équivalent à :

$$\exists j. \forall I \subseteq \{1, \ldots, n\}. ([\![t'_j]\!] \subseteq \bigcup_{i \in I} [\![t_i]\!]) \vee (\bigcap_{i \notin I} [\![s_i]\!] \subseteq [\![s'_j]\!])$$

Nous avons bien un modèle, et cela donne encore une nouvelle justification de la condition imposée sur la relation de sous-typage dans la définition des modèles. Par définition des types flèche, la notion d'application suivante est fidèle :

$$\{(e_i, e'_i)\}.e = \{e'_i \mid e_i = e\}$$

4.2 Algorithme de sous-typage

Pour que le système de types et la sémantique du langage soit utilisables, il faut être capable de décider la relation de sous-typage, ou, de manière équivalente le problème du vide (on a sous-typage $t \leq s$ si et seulement si $t \wedge \neg s \simeq 0$). Nous allons décrire un algorithme. Il procède en suivant l'étude que l'on a faite de la relation de sous-typage. Nous avons vu comment décomposer le problème dans chaque univers, puis revenir au problème initial, sur des « sous-termes » de l'expression de type initiale. Comme les types sont des expressions infinies, il faut faire attention à assurer la terminaison de l'algorithme. Nous montrons d'abord comment caractériser de manière coinductive le prédicat de vide pour les types interprétés dans le modèle réduit.

Definition 4.2.0.1. *Soit $R \subseteq \mathbf{T}$. On dit que R est une simulation si pour tout $t^0 \in R$:*

- $(P, N) \in t^0_{\mathbf{basic}} \Rightarrow \bigcap_{b \in P} [\![b]\!]_{\mathbf{basic}} \subseteq \bigcup_{b \in N} [\![b]\!]_{\mathbf{basic}}$
- $(P, N) \in t^0_{\mathbf{product}} \Rightarrow \forall N' \subseteq N.$

$$\left(\bigwedge_{(t \times s) \in P} t \wedge \bigwedge_{(t' \times s') \in N'} \neg t' \right) \in R$$

$$\vee$$

$$\left(\bigwedge_{(t \times s) \in P} s \wedge \bigwedge_{(t' \times s') \in N \setminus N'} \neg s' \right) \in R$$

- $(P, N) \in t^0_{\mathbf{arrow}} \Rightarrow \exists (t' \rightarrow s') \in N. \forall P' \subseteq P.$

$$\left(t' \wedge \bigwedge_{(t \rightarrow s) \in P'} \neg t \right) \in R$$

$$\vee$$

$$\left(\bigwedge_{(t \rightarrow s) \in P \setminus P'} s \wedge \neg s' \right) \in R$$

Notons qu'aucun modèle n'intervient dans cet algorithme : seule l'interprétation des types de base est utilisée, et elle a été choisie une fois pour toute. Deux lemmes techniques (section B.9) permettent d'établir que :

Théorème 4.2.0.2. *Soit t un type. Alors $[\![t]\!] = \varnothing$ si et seulement s'il existe une simulation R telle que $t \in R$.*

Insistons sur le fait que ce théorème est vrai pour le modèle réduit. Il donne un algorithme pour décider si un type t a une interprétation vide, et donc si deux types sont en relation de sous-typage. L'algorithme part d'un prédicat $R = \{t\}$ et essaie de le saturer jusqu'à obtenir une simulation. On voit que les types que l'on doit ajouter sont $\mathcal{X}$-définissables, où $\mathcal{X}$ est une base qui contient t, ce qui assure la terminaison de l'algorithme.

Remarque 24. *Nous pourrions formaliser l'algorithme et prouver sa correction, suivant l'approche de [AC93]. Nous préférons ne pas le faire : à partir du théorème et de la discussion ci-dessus, on peut facilement dériver un algorithme et se convaincre de sa correction. Les détails d'implémentation (en particulier, la représentation concrète des expressions de type et du prédicat R en cours de calcul) ne doivent pas obscurcir le cœur de l'algorithme, qui est le théorème énoncé ci-dessus.*

Dans notre implémentation, nous utilisons par exemple plusieurs optimisations pour garder en mémoire les résultats de sous-typage déjà prouvés ou infirmés.

4.3 Universalité du modèle réduit

En fait, un des lemmes utilisés dans la section précédente dit que dans tout modèle, si $[\![t]\!] = \varnothing$, alors t est dans une simulation. On en déduit :

Théorème 4.3.0.3. *Le modèle réduit est celui dans lequel la relation de sous-typage est la plus grosse : si $t \leq s$ est vrai dans un modèle, c'est vrai aussi dans le modèle réduit.*

Ce théorème est intéressant dans la mesure où l'algorithme de typage est monotone par rapport à la relation de sous-typage : plus cette relation contient de couples, plus on peut typer de termes (c'est le cas parce que l'on a supprimé les types flèche négatifs dans le typage des abstractions). Le modèle réduit donne donc le système de types qui permet de typer le plus de programmes ; ça tombe bien : c'est pour cette relation de sous-typage que nous avons obtenu un algorithme.

Remarque 25. *Je ne sais pas s'il existe des modèles qui donnent des relations de sous-typage différentes ; j'ai l'impression que non, mais je ne sais pas le montrer pour le moment.*

4.4 D'autres formules pour le sous-typage

Regardons l'équivalence énoncée dans le fait B.1.0.1. Dans le membre droit, que l'on note ($\star$) il faut tester 2^m disjonctions de la forme $A_J \vee B_{\{1,\ldots,m\}\backslash J}$, où :

$$\begin{array}{lll} A_J & ::= & \bigcap_i X_i \subseteq \bigcup_{j \in J} X'_j \\ B_J & ::= & \bigcap_i Y_i \subseteq \bigcup_{j \in J} Y'_j \end{array}$$

Ces assertions ne sont pas indépendantes les unes des autres. Précisément, on a, pour $J \subseteq J'$:

$$A_J \Rightarrow A_{J'}$$

et

$$B_J \Rightarrow B_{J'}$$

On peut organiser les tests de manière à profiter de ces implications. On procède avec un arbre de choix, en décidant successivement pour $j = 1, \ldots, m$ si on aura $j \in J$ ou $j \notin J$. On commence donc par tester si $A_\varnothing \vee B_\varnothing$. Si c'est vrai, on sait que le résultat du test ($\star$) sera vrai.

Sinon, on pose $J_A = \varnothing$ et $J_B = \varnothing$. On choisit de rajouter 1 dans J_A ou dans J_B. Cela fait deux branches à tester. Si on rajoute dans J_A, on doit tester A_{J_A}. Si c'est vrai, on peut arrêter la branche correspondante. Sinon, on procède de même pour 2, ...

En notant j_A lorsque on ajoute j à A et de même pour B, on voit que les tests s'organisent en un arbre :

$$\begin{tikzcd}[column sep=small]
 & & & {} \arrow[dll, "1_A"'] \arrow[drr, "1_B"] & & & \\
 & {} \arrow[dl, "2_A"'] \arrow[dr, "2_B"] & & & & {} \arrow[dl, "2_A"'] \arrow[dr, "2_B"] & \\
\dots & & \dots & & \dots & & \dots
\end{tikzcd}$$

Lorsque l'on emprunte une branche, on effectue le test A_{J_A} si on descend à gauche, et B_{J_B} si on descend à droite. Si c'est vrai, on peut élaguer le sous-arbre. Lorsque que l'on arrive à $j = m$, si le test a échoué, cela signifie que le test $(\star)$ est faux.

L'élagage permet souvent d'éviter de tester 2^m disjonctions. À titre d'exemple, voyons comment tester $X \times Y \subseteq X_1 \times Y_1 \cup X_2 \times Y_2$:

$$\begin{tikzcd}[column sep=small]
 & & & \begin{array}{c} X = \varnothing \\ \vee \\ Y = \varnothing \end{array} \arrow[dll] \arrow[drr] & & & \\
 & X \subseteq X_1 \arrow[dl] \arrow[dr] & & & & Y \subseteq Y_1 \arrow[dl] \arrow[dr] & \\
X \subseteq X_1 \cup X_2 & & Y \subseteq Y_2 & & X \subseteq X_2 & & Y \subseteq Y_1 \cup Y_2
\end{tikzcd}$$

Si par exemple $X \subseteq X_1$ et $Y \subseteq Y_1$, on voit que l'on évite tous les tests avec X_2 et Y_2. Évidemment, l'ordre dans lequel on considère les produits dans le membre droit est important ; un choix judicieux (commencer par les plus gros) permet d'avoir plus d'élagage.

Variante des algorithmes Nous pouvons utiliser ces constatations pour obtenir une variante du fait B.1.0.1, et en conséquence, une variante de l'algorithme de sous-typage, obtenu en changeant la définition de simulation (pour les cas des univers **product** et **arrow**).

De même, nous pouvons adapter les formules de calcul de π_1 et de $\_.\_$.

Chapitre 5

Prolongements, variantes

Dans ce chapitre, nous donnons quelques pistes pour étendre ou modifier légèrement le formalisme de notre travail. Dans la majorité des cas, notre implémentation (annexe A) utilise ces prolongements.

5.1 Enregistrements

5.1.1 Motivation : les attributs de XML

Revenons à la motivation initiale : nous voulons proposer une algèbre de types qui exprime des contraintes sur des documents XML, à la manière des DTDs ou des Schemas. Les types produit permettent de représenter la structure arborescente des documents et les contraintes d'expression régulière sur le contenu des éléments. Nous avons laissé de coté un aspect important de XML : les attributs. Dans un document XML, chaque élément a un ensemble fini d'attributs, qui sont des couples (étiquette,valeur) :

```
<tag label1=valeur1 label2=valeur2 ... labeln=valeurn>
...
</tag>
```

Notons plusieurs points intéressants :
- l'ordre des attributs n'a pas d'importance ;
- il y a un nombre fini d'attributs ;
- une même étiquette ne peut apparaître qu'une fois dans un ensemble d'attributs ;
- les étiquettes, contrairement aux valeurs qu'on leur associe, sont inertes : on ne fait pas de calcul dessus lorsque l'on manipule des documents XML ;
- on veut pouvoir exprimer des contraintes comme la présence ou l'absence d'un attribut d'étiquette donné, ou des contraintes sur la valeur de cet attribut.

Cela suggère de modéliser l'ensemble des attributs d'un élément par un *enregistrement*. Nous changeons le codage des documents XML ; un élément de *tag* l, de contenu v, et dont les attributs sont modélisés par un enregistrement r est représenté par exemple par $((l,r),v)$.

Il reste à préciser ce que l'on entend par enregistrement, quelles sont les contraintes que l'on s'autorise sur ces objets, et comment cela rentre dans notre formalisme. Généralement, dans une algèbre de types avec enregistrements, la relation de sous-typage permet d'« oublier » certains champs. Cela permet par exemple d'utiliser une fonction qui s'attendait à recevoir des enregistrements avec certains champs pour des enregistrements qui ont en fait plus de champs. Mais cela nous pose un problème : les attributs XML doivent en général avoir été déclarés pour pouvoir être présents dans un document. Si une DTD ne parle pas d'un attribut, il ne peut pas se trouver dans le document XML. Il faut donc garder un moyen d'exprimer par les types le fait qu'un enregistrement a tous ses champs dans un ensemble fini donné. S'il y a un nombre infini d'étiquettes, cela ne peut pas s'exprimer par une conjonction finie de contraintes élémentaires d'absence pour chaque champ en dehors de cet ensemble.

5.1.2 Les enregistrements comme fonctions totales

Fixons un ensemble d'étiquettes $\mathcal{L}$, qui peut être infini. Si X est un ensemble avec un élément distingué **undef**, notons $X^{\mathcal{L};\mathbf{undef}}$ l'ensemble des fonctions f de domaine $\mathcal{L}$, à valeurs dans X, et dont le support, défini par :

$$\mathrm{Supp}(f) = \{l \mid f(l) \neq \mathbf{undef}\}$$

est fini. Nous choisissons une constante **undef** dans $\mathcal{D}_{\mathbf{basic}}$ pour indiquer les champs « absents » et nous supposons que $[\![t_{\mathbf{undef}}]\!]_{\mathbf{basic}} = \{\mathbf{undef}\}$.

Les types enregistrements atomiques Nous allons définir un nouvel univers **record**. Les **record**-types atomiques ont deux formes possibles :
- $[l : t]$ où $l \in \mathcal{L}, t \in \mathbf{T}$
- $\mathbf{S}(L)$ où $L \in \mathcal{P}_f(\mathcal{L})$

Intuitivement, les enregistrements de type $[l : t]$ associent à l'étiquette l une valeur de type t ; les enregistrements de type $\mathbf{S}(L)$ ont leur support inclus dans L. Il n'y a aucun problème pour adapter la définition de l'algèbre de types, et les notions associées (en particulier, le socle de $[l : t]$ est $\{t\}$ et le socle de $\mathbf{S}(L)$ est vide).

Remarque 26. *Les types $\mathbf{S}(L)$ permettent, lorsque c'est nécessaire, d'interdire l'oubli de certains champs. Par exemple, une fonction en interne peut accepter des enregistrements de type $[x : int] \wedge [y : int]$, et elle fonctionnera tout aussi bien si on lui donne un enregistrement avec d'autres champs que x et y ; au contraire, si l'on doit envoyer le résultat d'un programme à un autre programme, et que ce dernier s'attend à recevoir un document XML validé par une certaine DTD donnée, il faut faire attention à ce que les éléments du document produit ne contiennent pas d'autres attributs que ceux autorisés par la DTD.*

En général, on utilisera une des deux formes suivantes :
- $[l_1 : t_1] \wedge \ldots \wedge [l_2 : t_2]$ *: type « ouvert »*
- $[l_1 : t_1] \wedge \ldots \wedge [l_2 : t_2] \wedge \mathbf{S}(\{l_1, \ldots, l_n\})$ *: type « fermé »*

Lorsque $t_{\mathbf{undef}} \leq t_i$, cela signifie que le champ l_i est optionnel.

Les enregistrements dans les modèles Passons aux modèles. Nous commençons par reprendre la définitions des structures. Une structure est maintenant de la forme :

$$\mathcal{D} = \mathcal{D}_{\mathbf{basic}} + \mathcal{D}_{\mathbf{product}} + \mathcal{D}_{\mathbf{record}} + \mathcal{D}_{\mathbf{arrow}}$$

avec toujours une bijection implicite :

$$\mathcal{D}_{\mathbf{product}} = \mathcal{D}^2$$

et aussi une bijection implicite :

$$\mathcal{D}_{\mathbf{record}} = \mathcal{D}^{\mathcal{L};\mathbf{undef}}$$

On suppose que l'ordre engendré par les relations $(e_1, e_2) > e_1$, $(e_1, e_2) > e_2$, $f > f(l)$ est bien fondé.

Nous adaptons la notion d'interprétation, en disant que l'interprétation des **record**-types atomiques vérifie :

$$\begin{array}{rcl} [\![[l : t]]\!] & = & \{f \in \mathcal{D}_{\mathbf{record}} \mid f(l) \in [\![t]\!]\} \\ [\![\mathbf{S}(L)]\!] & = & \{f \in \mathcal{D}_{\mathbf{record}} \mid \mathrm{Supp}(f) \subseteq L\} \end{array}$$

Modèle réduit Nous pouvons définir sans problème un modèle réduit pour la nouvelle définition de modèle. Il est toujours constitué de termes finis.

5.1.3 Sous-typage

Comme préliminaire à l'étude du problème du sous-typage dans l'univers des enregistrements, on constate que nécessairement, dans tout prémodèle :

$$\begin{array}{rcl} [l : t_1] \wedge [l : t_2] & \simeq & [l : t_1 \wedge t_2] \\ [l : t_1] \vee [l : t_2] & \simeq & [l : t_1 \vee t_2] \\ \neg_{\mathbf{record}}[l : \neg t] & \simeq & [l : \neg t] \end{array}$$

Pour étudier le sous-typage dans **record**, il faut savoir dans quels cas une intersection finie de **record**-types atomiques ou de leur négation a une interprétation vide. Avec les formules ci-dessus, on peut supposer provisoirement que tous les types $[l : t]$ apparaissent positivement (sans négation devant), et que les étiquettes sont deux à deux distinctes (pour une étiquette donnée l, on « pousse » les connecteurs booléens à l'intérieur des types $[l : t]$).

Avec la formule :

$$\mathbf{S}(L_1) \wedge \mathbf{S}(L_2) \simeq \mathbf{S}(L_1 \cap L_2)$$

on peut aussi supposer qu'il y a au plus un type de la forme $\mathbf{S}(L)$ en position positive.

On se ramène à étudier l'un des deux problèmes :

$$\bigcap_{1 \leq i \leq n} [\![[l_i : t_i]]\!]_{\mathbf{record}} \leq \bigcup_{1 \leq j \leq m} [\![\mathbf{S}(L_j)]\!]_{\mathbf{record}}$$

ou

$$[\![\mathbf{S}(L)]\!]_{\mathbf{record}} \cap \bigcap_{1 \leq i \leq n} [\![[l_i : t_i]]\!]_{\mathbf{record}} \leq \bigcup_{1 \leq j \leq m} [\![\mathbf{S}(L_j)]\!]_{\mathbf{record}}$$

avec les l_i deux à deux distincts.

La première forme est la plus facile. Le membre de gauche est vide si et seulement s'il existe un i tel que $t_i \simeq 0$. Si ce n'est pas le cas, alors on peut toujours construire un enregistrement qui satisfait les contraintes de gauche mais pas celles de droites (en prenant une étiquette l qui n'est aucune des l_i

et dans aucun des L_j et en lui associant un élément qui n'est pas **undef** ; cela marche toujours lorsque $\mathcal{L}$ est infini ; sinon, en adaptant ce qui suit, on trouve une condition un peu plus faible).

Regardons la deuxième forme. Il s'agit d'étudier quels sont les supports possibles pour des enregistrements qui satisfont les contraintes à gauche. Nécessairement, ces supports sont inclus dans L' où :

$$L' = L \backslash \{l_i \mid t_i \leq t_{\mathbf{undef}}\}$$

Est-il possible de construire un enregistrement avec exactement ce support ? Le seul problème, c'est si, pour un certain i, $l_i \notin L'$ et $t_i \wedge t_{\mathbf{undef}} \simeq 0$. Dans ce cas, le membre gauche du problème que l'on étudie est tout simplement vide, et le sous-typage a bien lieu. On fait donc l'hypothèse opposée. Alors effectivement, on peut construire un enregistrement f qui vérifie les contraintes du membre gauche, et qui a pour support L' (hors de L', $f(l) = \mathbf{undef}$; si $l_i \in L'$, $f(l) \in [\![t_i]\!] \backslash \{\mathbf{undef}\}$; et pour les autres $l \in L'$, on prend $f(l)$ quelconque différent de **undef**). Pour que le sous-typage soit vrai, il faut qu'il existe j tel que $L' = \mathrm{Supp}(f) \subseteq L_j$, et c'est également une condition suffisante.

Retenons que

$$[\![\mathbf{S}(L)]\!]_{\mathbf{record}} \cap \bigcap_{1 \leq i \leq n} [\![[l_i : t_i]]\!]_{\mathbf{record}} \leq \bigcup_{1 \leq j \leq m} [\![\mathbf{S}(L_j)]\!]_{\mathbf{record}}$$

est vrai si et seulement si :

$$\begin{array}{l} \exists i.\ (l_i \notin L) \wedge (t_i \wedge t_{\mathbf{undef}} \simeq 0) \\ \vee \\ \exists i.\ (l_i \in L) \wedge (t_i \simeq 0) \\ \vee \\ \exists j.\ L \backslash \{l_i \mid t_i \leq t_{\mathbf{undef}}\} \leq L_j \end{array}$$

Nous retrouvons la condition nécessaire et suffisante pour la première forme du problème (sans contrainte positive $\mathbf{S}(L)$), en prenant $L = \mathcal{L}$.

Souvenons nous que l'on avait poussé les connecteurs booléens à l'intérieur des types t_i, de sorte à avoir des l_i deux à deux distincts. Si l'on tient compte de cela, on voit que la condition obtenue ramène le problème du sous-typage dans l'univers **record** au sous-typage généralisé (intersection à gauche, réunion à droite) sur des sous-expressions de types (dans le socle). Cela permet d'adapter, dans le modèle réduit, l'algorithme de sous-typage.

5.1.4 Motifs

Nous étendons naturellement l'algèbre de motifs avec une nouvelle construction :

$$p ::= \ \ldots \mid \{l : p\}$$

(avec $l \in \mathcal{L}$)

Le comportement de ce motif est ce que l'on pense ; il se définit formellement par :

$$\begin{array}{lcll} f/\{l : p\} & = & f(l)/p & \text{si } f \in \mathcal{D}_{\mathbf{record}} \\ f/\{l : p\} & = & \Omega & \text{si } f \notin \mathcal{D} \end{array}$$

Le type accepté est :

$$\wr\{l:p\}\int = [l : \wr p \int]$$

Pour calculer le résultat du filtrage sur les types, on doit disposer pour toute étiquette l d'un opérateur stable qui associe à un type t le plus petit type s, lorsqu'il y en a un, vérifiant : $t \leq [l : s]$. Notons ce type $t.l$. Pour voir que cet opérateur existe (et est bien stable), nous procédons comme pour les opérateurs de projection π_1 et d'application sur les types, en reprenant l'étude de la relation de sous-typage.

Si t est une intersection de **record**-types atomiques et de négations de tels types, on peut le mettre sous la forme $t \simeq \mathbf{S}(L) \wedge \bigwedge_{1\leq i\leq n}[l_i : t_i] \wedge \bigwedge_{1\leq j\leq m} \neg\mathbf{S}(L_j)$ (ou la même, sans $\mathbf{S}(L)$), avec $l = l_1$ (si l n'est pas dans les l_i, on peut toujours le rajouter avec $t_1 = 1$). Supposons $t \not\simeq 0$. On cherche s tel que $t \leq [l : s]$. Si $l \notin L$, on voit que cela équivaut à $t_{\mathbf{undef}} \leq s$. Si $l \in L$, cela équivaut soit à $t_1 \wedge \neg t_{\mathbf{undef}} \leq s$ si $L \backslash \{l_i \mid t_i \leq t_{\mathbf{undef}}\} \backslash \{l\} \leq L_j$ pour un certain j, soit à $t_1 \leq s$ dans le cas contraire. Lorsque $t = \bigwedge_{1\leq i\leq n}[l_i : t_i] \wedge \bigwedge_{1\leq j\leq m} \neg\mathbf{S}(L_j) \not\simeq 0$, la condition est toujours $t_1 \leq s$ (si $\mathcal{L}$ est infini).

5.1.5 Langage

Pour terminer l'étude des enregistrements, il reste à voir comment ils s'intègrent dans le langage. Pour la destruction des enregistrements, le filtrage suffit. Il permet d'extraire un champ, de tester la présence ou l'absence d'un champ (en utilisant les motifs $\{l : \neg t_{\mathbf{undef}}\}$,$\{l : t_{\mathbf{undef}}\}$), ou si l'ensemble des étiquettes des champs présents est inclus dans un ensemble fini donné (motif contrainte de type $\mathbf{S}(L)$).

Enregistrements non extensibles

Il reste à donner les constructeurs. Le plus simple est de prendre une seule construction $\{l_1 = t_1; \ldots; l_n = t_n\}$ (avec les l_i deux à deux distincts) ; la règle de typage est :

$$\frac{\Gamma \vdash \mathsf{e}_1 : t_1 \quad \ldots \quad \Gamma \vdash \mathsf{e}_n : t_n}{\Gamma \vdash \{l_1 = \mathsf{e}_1; \ldots; l_n = \mathsf{e}_n\} : [l_1 : t_1] \wedge \ldots \wedge [l_n : t_n] \wedge \mathbf{S}(\{l_1, \ldots, l_n\})}$$

Cela permet de retrouver facilement les résultats théoriques sur le langage (en particulier le fait que le prémodèle des valeurs est équivalent au modèle d'amorce : il a assez d'expressions pour distinguer deux types qui ne sont pas équivalents dans le modèle d'amorce). Nous étendons bien entendu la classe des valeurs, pour accepter comme valeurs les $\{l_1 = v_1; \ldots; l_n = v_n\}$, ainsi que les contextes d'évaluation.

Enregistrements extensibles

Il y a d'autres choix intéressants. Par exemple, en plus de la forme précédente, nous pouvons introduire un opérateur d'ajout de champ et un opérateur de suppression.

L'opérateur $\oplus_l$ est binaire. Notons $\mathsf{e}_1 \oplus \{l = \mathsf{e}_2\}$ pour $\oplus_l((\mathsf{e}_1, \mathsf{e}_2))$. Il suppose que le premier argument est un enregistrement dont le champ d'étiquette l vaut **undef**, et il remplace cette valeur par le deuxième argument. On a donc :

$$t_{\oplus_l} = [l : t_{\mathbf{undef}}] \times 1$$

Le deuxième, noté de manière postfixe $\_\backslash l$, suppose que son argument est un enregistrement dont le champ d'étiquette l ne vaut pas **undef**, et il y met **undef** à la place. On a donc :

$$t_{\backslash l} = [l : \neg t_{\mathbf{undef}}]$$

Le comportement opérationnel de ces opérateurs est clair. Il faut définir comment ils se comportent sur les types, c'est-à-dire donner des formules pour $s \oplus_l t$ et $s\backslash l$. Nous allons suivre l'approche habituelle pour les définir sémantiquement, en définissant l'analogue des nouvelles opérations du langage dans un modèle quelconque. Si $f \in \mathcal{D}_{\mathbf{record}}$, $f(l) = \mathbf{undef}$ et $e \in \mathcal{D}$, alors $f \oplus \{l = e\}$ est l'enregistrement f' tel que $f'(l) = e$, $f'(l') = f(l')$ pour toute étiquette $l' \neq l$. Si s et t sont deux types, $s \leq [l : t_{\mathbf{undef}}]$ on définit $s \oplus_l t$ (à équivalence $\simeq$ près) par :

$$[\![s \oplus_l t]\!] = \{ f \oplus \{l = e\} \mid f \in [\![s]\!] \wedge e \in [\![t]\!] \}$$

Grâce à cette définition sémantique, on obtient par exemple directement la croissance de l'opérateur. On obtient aussi la garantie que l'on ne peut pas faire mieux (obtenir un type plus précis pour le résultat).

Il reste à donner les formules pour calculer effectivement l'opérateur. Seul $s_{\mathbf{record}}$ intervient, et dans la mesure où l'opérateur commute avec la réunion sur s, il suffit d'étudier le cas où s est une intersection de **record**-types atomiques et de négations de tels types, *d'interprétation non vide.* Nous avons vu qu'une telle intersection peut se mettre soit sous la forme :

$$s_0 = \bigwedge_{1 \leq i \leq n} [l_i : t_i] \wedge \mathbf{S}(L) \wedge \bigwedge_{1 \leq j \leq m} \neg \mathbf{S}(L_j)$$

soit sous cette même forme sans le $\mathbf{S}(L)$. Un calcul simple montre qu'alors :

$$s_0.t = \bigwedge_{l_i \neq l} [l_i : t_i] \wedge [l : t] \wedge \mathbf{S}(L \cup \{l\}) \wedge \bigwedge_{1 \leq j \leq m} \neg \mathbf{S}(L_j \cup \{l\})$$

(resp. la même chose sans $\mathbf{S}(L \cup \{l\})$)

On procède de même pour l'autre opérateur, et l'on obtient :

$$s_0 \backslash l = \bigwedge_{l_i \neq l} [l_i : t_i] \wedge \mathbf{S}(L \backslash \{l\}) \wedge \bigwedge_{j \mid l \in L_j} \neg \mathbf{S}(L_j \cup \{l\})$$

(resp. la même chose avec $[l : t_{\mathbf{undef}}]$ à la place de $\mathbf{S}(L\backslash\{l\})$)

Remarque 27. *On peut chercher à définir un unique opérateur $f \oplus \{l = e\}$, qui fonctionne même lorsque $f(l) = $ **undef**. Il permet de définir l'élimination d'un champ (par $f \oplus \{l = $ **undef**$\}$). Le problème, c'est que pour établir les formules ci-dessus, il faut distinguer les cas $f(l) = $ **undef**, $f(l) \neq $ **undef**. Il nous semble plus simple d'introduire effectivement deux constructions différentes. La construction générale peut évidemment s'exprimer à partir de ces deux-là (avec un filtrage).*

Remarque 28. *On peut aussi étudier un opérateur de concaténation $r_1 \oplus r_2$ qui fusionne les champs de ses deux arguments (en définissant une politique en cas de conflit ; par exemple, on peut rejeter les cas où la même étiquette est* *présente dans r_1 et r_2, ou alors donner la priorité à l'un des deux). Il faut pour cela pouvoir caractériser sur le type des arguments les cas rejetés, et calculer le type exact du résultat. Il s'agit essentiellement de calculs et raisonnements du même type que ceux utilisés ci-dessus.*

5.2 Polymorphisme paramétrique

Avertissement Cette section technique est la moins formalisée de notre travail. Il s'agit des premières étapes d'une étude qui reste largement à mener, et le traitement reste très informel. Ce qui suit est une description assez proche de l'implémentation (annexe A), assez éloigné de l'approche sémantique que nous avons utilisée jusqu'à présent.

5.2.1 Motivation, approche naïve

Dans le langage, on peut donner aux fonctions plusieurs types flèche $t \to s$; cela permet de définir des fonctions surchargées. En général, le corps de la fonction utilise le filtrage pour faire des calculs différents suivant le type de l'argument. C'est ce que l'on appelle le polymorphisme *ad hoc*.

La plupart des langages fonctionnels, comme ceux de la famille ML par exemple, donnent au contraire la possibilité d'avoir du polymorphisme paramétrique : le même code peut traiter des arguments de type différent. Le polymorphisme peut être implicite (ML) ou explicite (système F).

Prenons un exemple simple pour commencer. On peut définir dans le langage la fonction première projection :

$$\mathrm{fst} = \mu f : (t \times s \to t).\lambda x.\mathrm{match}\ x\ \mathrm{with}\ (x, \_) \to x \mid (x, \_) \to x$$

Pour pouvoir appliquer cette fonction à n'importe quel couple, on doit prendre $t = s = 1$. Mais alors, le système de types oublie toute information sur le type des arguments : si $\vdash e : t_0 \times s_0$, alors $\vdash fe : 1$. Et on voudrait obtenir $\vdash e : t_0$. On voit qu'il suffit pour cela de changer la définition de f avec $t = t_0$ (et $s = 1$ par exemple).

Cela suggère une approche naïve : on considère des définitions de fonction avec des *variables de type* dans les types spécifiés, qui se comportent comme des *macros* syntaxiques. Les objets définis ne sont pas des expressions du langage ; quand on veut utiliser une fonction ainsi déclarée, on associe des types aux variables de type, ce qui donne une expression du langage, dont on peut vérifier le type :

$$\mathrm{function\ fst}\ (\alpha)(\alpha \times 1 \to \alpha) \ldots .\ \mathrm{in}\ \ldots [\mathrm{fst\ with}\ \alpha = t_1] \ldots [\mathrm{fst\ with}\ \alpha = t_2] \ldots$$

Évidemment, cette solution n'est pas satisfaisante, pour les raisons suivantes :

- on ne vérifie jamais si la définition est bien typée : il se peut que dans les différentes situations où on l'utilise, tout se passe bien, mais qu'un jour, lorsque l'on veut utiliser la définition pour un nouveau type, une erreur de type apparaisse ;

– on doit disposer du texte complet de la définition pour pouvoir l'utiliser ; cela empêche d'avoir un système de modules, où, de l'extérieur de la fonction, on ne voit que son « interface » ;
– les algorithmes utilisés pour le typage sont déjà assez complexes, et si l'on doit typer de nouveau les fonctions à chaque utilisation dans le code (et cela récursivement, car les fonctions polymorphes peuvent faire appel à d'autres fonctions polymorphes), le coût du typage risque de devenir trop important.

La première raison est peut-être la plus problématique. Donnons un exemple. Supposons que l'on représente des listes par des couples imbriqués : $(x_1, \ldots, x_n)$ est codée en $(x_1, (x_2, (\ldots, (x_{n-1}, x_n) \ldots)))$. On veut définir une fonction polymorphe qui renvoie le *dernier* élément (x_n) d'une telle liste. Un simple filtrage avec le motif suivant suffit :

$$p = (\_, p) | x$$

Si l'on ne fait pas attention, on risque de définir une fonction polymorphe de type $t \to \alpha$ où $t = (1 \times t) \vee \alpha$. Et cela convient tant que l'on utilise la fonction en instanciant α avec des types de base. Mais c'est faux si le type que l'on donne à α peut contenir des couples ! Précisément, on doit supposer que $\alpha \leq \neg(1 \times 1)$, mais on ne se rend pas compte tout de suite que l'on a besoin d'une telle hypothèse.

Remarque 29. *En fait, une contrainte plus faible suffit dans cet exemple :* $\alpha \wedge (1 \times 1) \leq 1 \times \alpha$, *mais la fonction ne fait probablement pas ce que l'on attend d'elle si la contrainte plus forte n'est pas satisfaite. On voit qu'il est intéressant de pouvoir spécifier une contrainte plus forte si on le désire.*

La solution vient naturellement : il faut typer une fois pour toutes les fonctions polymorphes, et garder suffisamment d'information pour pouvoir simuler le comportement de l'approche naïve. Nous allons décrire l'approche la plus simple, que l'on a mise en œuvre dans l'implémentation : on introduit explicitement lors des définitions de fonctions polymorphes les variables de type et les contraintes sur ces variables utilisées pour typer le corps de la fonction, et pour utiliser ces fonctions, on donne explicitement les types avec lesquels on instancie les variables de type.

5.2.2 Aperçu

On se donne un ensemble de variables de type, notées $\alpha, \beta, \ldots$ Une définition de fonction polymorphe a la forme suivante :

$$\begin{array}{l} \text{function } f(\alpha_1, \ldots, \alpha_n; t)(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathtt{e}_1 \\ \text{in } \mathtt{e}_2 \end{array}$$

L'identificateur f désigne le nom de la fonction ; les variables $\alpha_1, \ldots, \alpha_n$ peuvent être utilisées dans les types t_i, t'_i, dans les types qui apparaissent à l'intérieur de $\mathtt{e}_1$ (dans l'interface d'autres fonctions, ou dans des motifs) ; elle sont aussi utilisées dans le type t qui donne une *contrainte* sur ces variables. On interprète cette contrainte en disant que t doit être vide. Par exemple, si $t = (\alpha_1 \wedge \alpha_2 \wedge \neg\alpha_3) \vee (\alpha_4 \wedge \neg\alpha_5)$, la contrainte signifie que l'on ne peut instancier les α_i avec des types s_i que si $s_1 \wedge s_2 \leq s_3$ et $s_4 \leq s_5$.

Dans $\mathtt{e}_2$, on s'autorise à utiliser f sous la forme suivante :

$$[f \text{ with } \alpha_1 = s_1, \ldots, \alpha_n = s_n]$$

à condition que la contrainte t soit satisfaite par ces types. Cette expression se comporte alors comme :

$$\mu f(\phi(t_1) \to \phi(t'_1); \ldots ; \phi(t_m) \to \phi(t'_m)).\lambda x.\phi(\mathtt{e}_1)$$

où ϕ désigne la substitution $[s_i/\alpha_i]$. La contrainte s'écrit : $\phi(t) \simeq 0$.

En fait, on peut aussi utiliser f sous cette forme dans $\mathtt{e}_1$, ce qui autorise des appels récursifs de f avec des instanciations différentes des variables de type (récursion polymorphe). Bien entendu, on peut également utiliser simplement f dans $\mathtt{e}_1$, ce qui équivaut à $[f \text{ with } \alpha_1 = \alpha_1, \ldots, \alpha_n = \alpha_n]$.

Remarque 30. *Insistons sur le fait que la déclaration polymorphe n'est pas une expression du langage : « function $f(\alpha_1, \ldots, \alpha_n; t)(t_1 \to t'_1; \ldots ; t_m \to t'_m).\lambda x.\mathtt{e}_1$ » n'a pas de sens sans le « in $\mathtt{e}_2$ ». Cela signifie par exemple que l'on ne peut pas passer en argument à une fonction une déclaration polymorphe. Bien entendu, les instances de la déclaration sont des expressions tout à fait normales.*

On veut typer une seule fois le corps de la fonction (plus exactement, $\mathtt{e}_1$ est typé autant de fois qu'il y a de flèches dans son interface, soit m ; cela correspond au polymorphisme par surcharge). On utilise une règle semblable à celle utilisée dans l'algorithme de typage du langage. Pour cela, il faut être capable de prouver, supposant x de type t_i, et sous la contrainte t, que $\mathtt{e}_1$ est de type t'_i.

Techniquement, avant de présenter formellement le nouveau système de type pour le langage, on doit traiter les questions suivantes :

- il faut revoir la définition de l'algèbre de types pour y intégrer les variables de type ;
- il faut définir une relation de sous-typage entre des expressions types avec variables, modulo une contrainte ;
- il faut prolonger les différents opérateurs de types (π_1, $\_ . \_$, ...), ce qui permet de définir entre autres le filtrage.

Remarque 31. *Dans [Pie89], il est noté que la première étape pour ajouter un polymorphisme du style système F à un système avec sous-typage est de définir la relation de sous-typage pour les types avec variables, sous une contrainte (qui donne des bornes inférieures ou supérieures aux variables). Il se trouve que cette étape n'est pas triviale en présence par exemple des types intersection. On peut voir ce qui suit comme une approche assez générale au problème.*

Remarque 32. *L'intégration du polymorphisme «ad hoc» du $\lambda\&$ calcul et du polymorphisme paramétrique du système $F_{\leq}$ a été étudiée dans [Cas95].*

5.2.3 L'algèbre de types avec variables

Description informelle

Nous allons reprendre une idée déjà utilisée pour définir l'algèbre de types (que nous appelons maintenant algèbre close), en représentant directement les combinaisons booléennes en forme normale.

Informellement, nous dirons qu'un type t est une réunion $t_1 \vee \ldots \vee t_n$, où chaque t_i est une intersection de variables α, de négations de variable $\neg\alpha$, et d'un type *clos à la racine* (défini ci-dessous). On suppose qu'il n'y a pas, dans la réunion, deux t_i, t_j avec la même intersection de variables (sinon, on les regroupe en prenant l'intersection des types clos à la racine associés) ; on suppose également que dans chaque t_i, aucune variable ne peut apparaître en même temps que sa négation (sinon l'intersection est vide, et on peut la supprimer de la réunion !).

Chaque type t_i peut ainsi être représenté par une fonction partielle de domaine fini, qui associe à certains couples (p, n) un type close à la racine ; p et n sont deux ensembles finis et disjoints de variables de type (les positives, les négatives).

Dans un type clos à la racine, les variables de type ne peuvent apparaître que sous un constructeur (flèche, produit, enregistrement). Au niveau supérieur, les types clos à la racine ressemblent donc aux types de l'algèbre close. On les note g (g pour *ground*).

Définition

Notons A l'ensemble des variables de type. Formellement, on définit l'algèbre de types par les équations suivantes :

$$\begin{cases} \mathbf{T} & = & (\mathcal{P}_f(A) \rtimes \mathcal{P}_f(A)) \to_f \mathbf{T}^c \\ \mathbf{T}^c & = & \prod_u \mathbf{T}_u \\ \mathbf{T}_u & = & \mathcal{P}_f(\mathcal{P}_f(\mathbf{T}_u^\circ) \times \mathcal{P}_f(\mathbf{T}_u^\circ)) \\ \mathbf{T}^\circ_{\mathbf{product}} & = & \mathbf{T}^2 \\ \mathbf{T}^\circ_{\mathbf{arrow}} & = & \mathbf{T}^2 \end{cases}$$

(et éventuellement aussi $\mathbf{T}^\circ_{\mathbf{record}}, \ldots$)

On a noté $X \to_f Y$ l'ensemble des fonctions *partielles* de domaine fini inclus dans X et de codomaine Y ; $X \rtimes Y$ représente $\{(x, y) \in X \times Y \mid x \cap y = \varnothing\}$. Comme pour l'algèbre close, on considère la solution de ce système d'équations qui est constituée des types récursifs (qui n'ont qu'un nombre fini de sous-expressions distinctes).

On note $(p, n, g) \in t$ pour $t(p, n) = g$ lorsque $t \in \mathbf{T}$. Dans $\mathbf{T}^c$, on peut définir les connecteurs booléens, ainsi que 0 et 1, comme pour l'algèbre close. On plonge $\mathbf{T}^c$ et A dans $\mathbf{T}$ par :

$$g = [(\varnothing, \varnothing) \mapsto g]$$

$$\alpha = [(\{\alpha\}, \varnothing) \mapsto 1]$$

En se guidant par l'intuition qu'un type $t \in \mathbf{T}$ représente en fait :

$$\bigvee_{(p,n,g) \in t} (\bigwedge_{\alpha \in p} \alpha \wedge \bigwedge_{\alpha \in n} \neg\alpha \wedge g)$$

on peut définir dans $\mathbf{T}$ les connecteurs booléens ainsi que 0 et 1 (par exemple 0 est le graphe vide et $1 = [(\varnothing, \varnothing) \mapsto 1]$; il y a un petit abus de notation à noter de même $0 \in \mathbf{T}$ et $0 \in \mathbf{T}^c$ parce qu'ils ne sont pas identifiés par le plongement $\mathbf{T}^c \to \mathbf{T}$, mais cela ne pose pas de problème).

On définit naturellement $A(t)$ comme l'ensemble des variables de type α qui apparaissent dans t. Un type est clos si $A(t) = \varnothing$; cela signifie que pour tout (p, n, g) dans t ou dans une sous-expression de t (c'est-à-dire dans les sous-expressions des types clos à la racine qui sont dans son image), $p = n = \varnothing$.

Substitution

Si on note $\mathbf{T}^A$ l'algèbre de types construite sur l'ensemble de variables A et si $A_1 \subseteq A_2$, alors on a un plongement naturel $\mathbf{T}^{A_1} \longrightarrow \mathbf{T}^{A_2}$. Pour aller dans l'autre sens, on doit « instancier » les variables de $A_2 \backslash A_1$.

Dans le cas général (A_1 et A_2 deux ensembles de variables), si $\phi : A_1 \longrightarrow \mathbf{T}^{A_2}$, on peut l'étendre en une application, encore notée ϕ, $\mathbf{T}^{A_1} \longrightarrow \mathbf{T}^{A_2}$, qui commute avec les connecteurs booléens, les constructeurs de type, et le plongement des types clos.

De même, on voit que $\mathbf{T}^{\varnothing}$ est isomorphe naturellement à l'algèbre close.

5.2.4 Sous-typage modulo des hypothèses

Dans l'absolu, cela n'a pas de sens de parler de sous-typage entre types avec variables. Ce que l'on peut commencer à faire, c'est de supprimer les variables en les instanciant. Si $\phi : A \longrightarrow \mathbf{T}^{\varnothing}$, on note $t_1 \leq_\phi t_2$ pour $\phi(t_1) \leq \phi(t_2)$.

En fait, ce qui nous intéresse, c'est de voir si on peut déduire un sous-typage $t_1 \leq t_2$ en faisant une hypothèse t (on dit aussi sous la contrainte t). On note $t_1 \leq_t t_2$ si pour toute substitution ϕ telle que $\phi(t) \simeq 0$, alors $t_1 \leq_\phi t_2$.

Par exemple, si $t = (\alpha_1 \wedge \alpha_2) \vee (g \wedge \neg\alpha_1)$, alors $g \wedge \alpha_2 \simeq_t 0$. Autre exemple : si $t = (\alpha_1 \wedge \neg(\alpha_1 \rightarrow 1)) \vee ((\alpha_1 \rightarrow 1) \wedge \neg\alpha_1)$, alors $\alpha_1 \simeq_t s$ où s est l'unique type clos tel que $s = s \rightarrow 1$.

Remarque 33. *Il est important de noter que la conjonction des hypothèses t_1 et t_2 est $t_1 \vee t_2$ (et non pas l'intersection). En effet, $t_1 \simeq 0 \wedge t_2 \simeq 0 \Leftrightarrow t_1 \vee t_2 \simeq 0$.*

Nous n'allons pas étudier en détail le problème du calcul de la relation $\leq_t$. Nous nous contenterons de donner une variation de l'algorithme de sous-typage dans le modèle réduit pour calculer une approximation inférieure (correcte, pas forcément complète) de cette relation. C'est-à-dire que si l'algorithme affirme $t_1 \leq_t t_2$ (ce que l'on note $t_1 \leq_t^A t_2$), alors effectivement $t_1 \leq_t t_2$. N'importe quel algorithme convient pour la suite ; bien entendu, plus l'algorithme donne de relations, plus on pourra typer d'expressions dans le langage.

Remarque 34. *Cela a quand même un intérêt d'introduire la relation $\leq_t$ même si on ne la calcule pas : elle permet d'utiliser l'approche sémantique pour prouver des propriétés formelles qui peuvent donner des informations quand on travaille avec l'approximation $\leq_t^A$.*

On se ramène à étudier le prédicat $\simeq_t 0$. Le principe de l'algorithme ne change pas : on décompose les types en gardant trace de ceux rencontrés pour assurer la terminaison de l'algorithme. Dire que $s \simeq_t 0$, c'est dire que pour tout $(p, n, g) \in s$, $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha \wedge g \simeq_t 0$. Si l'on ne sait rien sur les α, on peut juste dire qu'une condition suffisante est : $g \simeq_t 0$, ce qui décompose bien le problème (après, g est clos à la racine, et on utilise la même décomposition que dans l'algorithme original). Mais on sait des choses sur les α, grâce à t. Supposons que l'on peut trouver un type close à la racine g' tel que $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha \leq_t g'$. Alors une condition suffisante plus faible est : $g \wedge g' \simeq_t 0$.

Pour assurer la terminaison de l'algorithme, il faut voir que l'on n'introduit pas de nouvelle variable de type : toutes celles qui apparaissent étaient déjà là, soit dans le type qu'on teste, soit dans t. On conclut en étendant le concept de base (on inclut toutes les variables de type dans le socle, et étant donnée une base, il n'y a toujours qu'un nombre fini de types définissables dans cette base).

On voit que l'on obtient un algorithme de sous-typage dès que l'on se donne une méthode pour trouver une approximation supérieure (modulo t) d'une intersection de la forme $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha$.

Regardons t :

$$t = \bigvee_{(p,n,g)\in t} \bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha \wedge g$$

Alors, pour tout $(p, n, g) \in t$:

$$\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha \leq_t \neg g$$

Cela donne déjà des majorations pour certains (p, n). On peut en obtenir d'autres en saturant le système d'inéquations par transitivité (cette méthode est utilisée par exemple dans [AW93]). En effet, si :

$$\bigwedge_{p_1} \alpha \wedge \bigwedge_{n_1} \neg\alpha \leq_t \neg g_1$$

et

$$\bigwedge_{p_2} \alpha \wedge \bigwedge_{n_2} \neg\alpha \leq_t \neg g_2$$

et, disons $\alpha_0 \in p_1 \cap n_2$, alors :

$$\alpha_0 \leq_t \neg(g_1 \wedge \bigwedge_{p_1\backslash\{\alpha_0\}} \alpha \wedge \bigwedge_{n_1} \neg\alpha)$$

et

$$\bigwedge_{p_2} \alpha \wedge \bigwedge_{n_2\backslash\{\alpha_0\}} \neg\alpha \wedge g_2 \leq_t \alpha_0$$

et donc par transitivité sur α_0 :

$$\bigwedge_{p_2\cup p_1\backslash\{\alpha_0\}} \alpha \wedge \bigwedge_{n_1\cup p_2\backslash\{\alpha_0\}} \neg\alpha \leq \neg(g_1 \wedge g_2)$$

Si $p_1 \cap n_2$ n'est pas un singleton, ou si $p_2 \cap n_1$ n'est pas vide, alors on n'obtient rien d'intéressant puisque le membre gauche est de toute façon vide (il contient une variable et sa négation). On applique donc la règle de transitivité lorsque $p_1 \cap n_2 = \{\alpha_0\}$ et $p_2 \cap n_1 = \varnothing$.

On sature ainsi dans un premier temps l'ensemble des majorations obtenues directement à partir de t. Ensuite, lorsque l'on veut trouver une majoration pour $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha$, on cherche dans le système de majorations tous les $\bigwedge_{p'} \alpha \wedge \bigwedge_{n'} \neg\alpha \leq g'$ tels que $p' \subseteq p, n' \subseteq n$ et on prend comme majoration l'intersection $\bigwedge g'$ des g' qui correspondent.

Il est facile de voir que l'on n'aurait rien gagné à ajouter au système les inéquations ainsi obtenues et à le saturer de nouveau par transitivité.

Remarque 35. *L'algorithme obtenu n'est* pas *complet. Par exemple, si* $t = (t_0 \times t_1) \wedge \neg(t_0 \times t_2)$*, et si par exemple* t_0 *est un type clos non vide, alors l'algorithme ne voit pas* $t_1 \leq_t t_2$*, ce qui est pourtant vrai. Dans la pratique, il nous semble en général possible de formuler les contraintes pour que l'algorithme puisse trouver tout ce dont on a besoin pour typer les programmes. Dans l'exemple ci-dessus, on doit évidemment formuler la contrainte sous la forme* $t_1 \wedge \neg t_2$*.*

5.2.5 Opérateurs sur les types

Que vaut $\pi_1(s)$ lorsque s est un type avec variables ? Dans le cas clos, on avait défini $\pi_1(s)$ comme le plus petit type vérifiant $s \leq \pi_1(s) \times 1$. Si l'on se place sous une hypothèse t, comme dans la section précédente, on peut définir $\pi_{1,t}(s)$ comme le plus petit type vérifiant $s \leq_t \pi_{1,t}(s) \times 1$.

En fait, on va encore chercher une approximation supérieure de $\pi_{1,t}(s)$. Comme l'opérateur commute avec la réunion, on se ramène au cas où $s = \bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha \wedge g$, avec g clos à la racine. On prend alors un majorant de $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha$ clos à la racine, disons g', et on majore $\pi_{1,t}(s)$ par $\pi_{1,t}(g \wedge g')$. On calcule ce dernier type avec la même formule que π_1 dans le cas clos. On voit que l'opérateur obtenu est toujours « stable » (pour la nouvelle définition des bases).

Essentiellement, donc, pour calculer les opérateurs tels π_1 dans le cas avec variable et sous hypothèse, on remplace « à la racine » les intersections $\bigwedge_p \alpha \wedge \bigwedge_n \neg\alpha$ par un majorant modulo t clos à la racine, comme pour le sous-typage. On en déduit une approximation supérieure de ces opérateurs, y compris de l'opérateur de filtrage $(t \Downarrow p)(x)$, et cela va suffire pour garantir la sûreté du typage.

5.2.6 Le langage

Nous allons donner informellement les modifications à apporter au système de types pour prendre en compte les deux nouvelles constructions (définition de fonction polymorphe et instanciation).

Le jugement de typage prend la forme :

$$t, \Gamma, \Pi \vdash \mathsf{e} : s$$

où :

- t est l'hypothèse courante (contraintes sur les variables de types) ;
- Γ est un environnement de typage comme dans le cas classique ;
- Π est un environnement de typage polymorphe ; il associe aux identificateurs de fonctions polymorphes f des triplets $((\alpha_1, \ldots, \alpha_n), t', s)$ où t' désigne la contrainte à satisfaire sur les α_i pour pouvoir instancier l'objet polymorphe et s est le type de cet objet.

La règle de typage des abstractions est la seule qu'il faut changer : on interdit d'ajouter des négations de flèches (on a fait la même restriction pour définir l'algorithme de typage). Cela se justifie facilement : lorsque que l'on raisonne sous une hypothèse, si l'on obtient $t_1 \leq_t t_2$, on est sûr qu'effectivement, lorsque les variables de type seront instanciées de sorte à satisfaire l'hypothèse, on aura $t_1 \leq t_2$. Par contre, si l'on n'a pas $t_1 \leq_t t_2$, on ne peut rien dire, et il ne faut donc pas utiliser cette information négative pour donner des types plus précis à des expressions. On prend :

$$\frac{\begin{array}{l}(\forall i).\ t, (\Gamma + (x : t_i) + (f : t_f)), \Pi \vdash \mathsf{e} : s_i^0 \leq_t s_i \\ t_f = (t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)\end{array}}{t, \Gamma, \Pi \vdash \mu f : (t_1 \to s_1; \ldots ; t_n \to s_n).\lambda x.\mathsf{e} : t_f}$$

On reprend toutes les autres règles du système en remplaçant la relation $\leq$ et les opérateurs tels $(t \Downarrow p)$ par leurs analogues dans le cas avec variable sous hypothèse (en prenant comme hypothèse celle qui apparaît dans le jugement). L'environnement polymorphe Π et l'hypothèse courante ne changent pas entre les prémisses et la conclusion de ces règles.

Deux nouvelles règles correspondent aux deux nouvelles constructions du langage :

$$\frac{(f : ((\alpha_1, \ldots, \alpha_n), t', s)) \in \Pi \quad t'[s_i/\alpha_i] \simeq_t 0}{t, \Gamma, \Pi \vdash [f \text{ with } \alpha_1 = s_1, \ldots, \alpha_n = s_n] : s[s_i/\alpha_i]}$$

$$\frac{\begin{array}{l}(\forall i).\ (t \vee t'), (\Gamma + (x : t_i) + (f : t_f)), (\Pi + (f : t^f)) \vdash \mathsf{e}_1 : s_i \\ t, \Gamma, (\Pi + (f : t^f)) \vdash \mathsf{e}_2 : s \\ t_f = (t_1 \to t'_1) \wedge \ldots \wedge (t_m \to t'_m) \\ t^f = ((\alpha_1, \ldots, \alpha_n), t', t_f)\end{array}}{t, \Gamma, \Pi \vdash \text{function } f(\alpha_1, \ldots, \alpha_n; t')(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathsf{e}_1 \text{in } \mathsf{e}_2 : s}$$

Remarque 36. *Pour obtenir un algorithme, il faut bien entendu remplacer $\leq_t$ par sa version algorithmique $\leq_t^A$, et de même pour les opérateurs $(t \Downarrow p)$, ... Néanmoins, il est plus simple de formuler le système avec les « vrais » opérateurs, définis sémantiquement, sur lesquels on peut plus facilement raisonner.*

Remarque 37. *Le lecteur attentif pourra se demander si une seule contrainte t suffit dans les jugements : en effet, lorsque l'on type des déclarations polymorphes emboîtées, il faut travailler modulo plusieurs contraintes. Nous utilisons la réunion sur les types pour exprimer la conjonction de ces contraintes : c'est le rôle du $t \vee t'$ dans la règle ci-dessus.*

La sémantique opérationnelle est un peu pénible à étendre, dans la mesure où les fonctions polymorphes ne sont pas des expressions du langages (on ne peut pas évaluer une déclaration avec une simple substitution comme un let-in classique).

On utilise la règle de réduction suivante :

$$(\text{function } f(\alpha_1, \ldots, \alpha_n; t')(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathsf{e}_1 \text{in } \mathsf{e}_2) \to \mathsf{e}'_2$$

où e'_2 est obtenu à partir de e_2 en remplaçant chaque $[f \text{ with } \alpha_1 = s_1, \ldots, \alpha_n = s_n]$ par :

$$\begin{array}{l}\mu f(\phi(t_1) \to \phi(t'_1); \ldots; \phi(t_m) \to \phi(t'_m)).\lambda x. \\ \text{function } f(\alpha_1, \ldots, \alpha_n; t')(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathsf{e}_1 \text{in} \\ \phi(\mathsf{e}_1)\end{array}$$

où ϕ est la substitution $[s_i/\alpha_i]$.

Pour établir le théorème de *subject reduction*, on doit prouver un lemme de substitution avec l'environnement Π des déclarations polymorphes.

Remarque 38. *Si l'on ne tient pas à garder la récursion polymorphe, on peut se passer de redéclarer la fonction polymorphe dans l'expression ci-dessus, et on peut adopter la règle de typage suivante, qui ressemble à celle d'un let-in classique :*

$$\frac{\begin{array}{l}(t \vee t'), \Gamma, \Pi \vdash \mu f(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathsf{e}_1 : t_f \\ f, \Gamma, (\Pi + (f : t^f)) \vdash \mathsf{e}_2 : s \\ t^f = ((\alpha_1, \ldots, \alpha_n), t', t_f)\end{array}}{t, \Gamma, \Pi \vdash \textit{function } f(\alpha_1, \ldots, \alpha_n; t')(t_1 \to t'_1; \ldots; t_m \to t'_m).\lambda x.\mathsf{e}_1 \textit{ in } \mathsf{e}_2 : s}$$

5.2.7 Extensions

Variables de rangées Le polymorphisme que l'on a décrit permet d'éviter de perdre certaines informations. Par exemple, on peut exprimer qu'une fonction qui modifie certains champs d'un enregistrement ne touche pas aux autres ; ainsi le type $[l : \alpha] \rightarrow [l : \alpha]$ permet-il d'exprimer que si le champ d'étiquette l était absent, il le reste, et sinon son type ne change pas (ce qui est très fort par exemple si l'on a un type singleton par constante). On voudrait pouvoir exprimer cela pour un ensemble infini (cofini en fait) de champs. Une idée serait d'introduire des variables de type dites de rangées [Rém93a, Rém93b]. Nous n'avons pas encore exploré cette idée.

Polymorphisme implicite ? Le polymorphisme paramétrique que l'on a présenté est explicite, comme celui du système F et contrairement à celui de ML (par rapport au système F, ou $F_{\leq}$ ou F-bounded, cependant, les fonctions polymorphes ne sont pas des expressions du langage).

On ne peut pas vraiment faire autrement dans le cas général avec la sémantique du langage : en effet, les types que l'on utilise pour instancier les variables peuvent changer la sémantique de la fonction polymorphe. Par exemple, considérons la déclaration :

$$\text{function } f(\alpha_1; 0)(1 \rightarrow 1).\lambda x.\text{match } x \text{ with } \alpha \rightarrow \mathsf{c}_1 \mid \neg\alpha \rightarrow \mathsf{c}_2 \text{ in } \ldots$$

On peut choisir n'importe quel type lorsqu'on instancie cette déclaration, et il n'y a pas de meilleur choix.

Vers le système $F_{\leq}$ On peut voir l'étude qui précède comme une première étape vers l'intégration de notre système de types et d'un polymorphisme du style système $F_{\leq}$ (ou plutôt $F_{\leq}^{\&}$ [Cas95], qui a aussi le polymorphisme *ad hoc* par surcharge et liaison tardive). L'idée est de voir les définitions polymorphes comme des expressions du langage, et d'avoir un opérateurs d'application $.t$ pour chaque type t (son argument doit être un objet polymorphe, c'est-à-dire une fonction des types dans les objets, et on l'applique précisément au type t). Cela permet d'avoir des fonctions qui prennent en argument ou renvoient des fonctions polymorphes.

La deuxième étape est donc d'ajouter à l'algèbre de types les quantificateurs : $\forall\alpha[t].s$ (quantification universelle sur la variable α sous la contrainte t). Il faut définir la relation de sous-typage pour ces types. Si l'on veut continuer dans la même voie, il faut donner une interprétation sémantique à ces types quantifiés, comme des fonctions des types vers le domaine. Pour coller au cas des types flèche, on peut considérer qu'il s'agit en fait de relations (fonctions partielles, non déterministes) ; on a vu que cela simplifie la définition de la relation de sous-typage en évitant les cas dégénérés où le domaine est fini. On voit le type $\forall\alpha[t].s$ comme la contrainte suivante : si le type a satisfait t (c'est-à-dire : $t[a/\alpha] \simeq 0$), alors l'image de a est un objet dans $[\![s]\!]$. Cela fait apparaître de nouveau une circularité : le sous-typage est défini en fonction de l'inclusion des valeurs, et la notion de modèle dépend de la relation de sous-typage.

On peut néanmoins essayer de voir à quoi ressemble cette relation de sous-typage, pour éventuellement l'axiomatiser partiellement. On a donc un nouvel univers des types quantifiés, et il faut étudier le sous-typage d'une intersection de tels types dans une réunion de tels types (par α-conversion, on suppose qu'ils ont tous la même variable de quantification) :

$$\bigwedge_{i \in I} \forall \alpha[t_i].s_i \leq \bigvee_{j \in J} \forall \alpha[t'_j].s'_j$$

Dans la mesure où l'on voit ces types comme des fonctions (des types vers le domaine), il est raisonnable de postuler une disjonction forte, et de dire que le sous-typage ci-dessus équivaut à :

$$\exists j.\ \bigwedge_i \forall \alpha[t_i].s_i \leq \forall \alpha[t'_j].s'_j$$

On peut continuer à adapter ce qui se passe pour les types flèche. Regardons d'abord le cas où l'intersection n'a qu'un terme :

$$\forall \alpha[t].s \leq \forall \alpha[t'].s'$$

La règle de $F_{\leq}$ dit que cela est vrai lorsque, sous l'hypothèse t', on peut prouver la contrainte t et établir $s \leq s'$. Autrement dit :

$$(t \simeq_{t'} 0) \wedge (s \leq_{t'} s')$$

Avec l'interprétation des objets polymorphes comme relations, il faut aussi autoriser les deux cas : $s' \simeq 1$ et t' impossible. Le deuxième signifie que l'on ne peut pas instancier α de sorte à rendre t' vide ; une condition suffisante, c'est si l'algorithme de sous-typage sous hypothèse donne $t'' \leq_{t'} 0$ avec t'' clos non vide ; par exemple, l'algorithme que l'on a décrit permet de trouver que la contrainte $(\alpha \wedge t) \vee (\neg\alpha \wedge t)$ est impossible lorsque $t \not\leq 0$ est un type clos. Après clôture par transitivité, on obtient en effet la contradiction $t \leq 0$.

Dans le cas général avec une intersection de plusieurs types à gauche, on peut adopter les même formules que pour le sous-typage des types flèche. Il faut être capable de dire si la contrainte t' implique la disjonction des contraintes t_i (pour $i \in I' \subseteq I$). Là encore, il est raisonnable de supposer qu'il y a disjonction forte : c'est le cas par exemple lorsque les contraintes sont des majorations simples $t_i = \alpha \wedge a_i$. Ensuite, dire que t' implique t, c'est dire que $t \simeq_{t'} 0$.

Finalement, le problème de sous-typage que l'on étudie serait équivalent à :

$$\exists j. \forall I' \subseteq I. (\exists i \in I'. t_i \simeq_{t'_j} 0) \vee (\bigwedge_{I \setminus I'} s_i \leq_{t'_j} s'_j)$$

Remarque 39. *Cette dernière section est à prendre avec des pincettes. L'étude de tout cela reste à faire et je n'ai pas du tout considéré les conséquences d'une telle définition. De plus, on sait qu'il se pose des problèmes de décidabilité [Pie94] pour calculer la relation de sous-typage dans ce genre de système avec polymorphisme à la système $F_{\leq}$.*

5.3 Inférence du type du résultat des fonctions non récursives

Dans le langage, on donne explicitement l'interface complète des fonctions, sous la forme d'une liste de types flèche. Pour les fonctions non récursives, il est possible d'inférer le type des résultats, en donnant juste les types possibles de l'argument. Cela revient à ajouter la construction syntaxique au langage :

$$\lambda(x : t_1; \ldots ; t_n).\mathtt{e}$$

et la règle de typage :

$$\frac{\begin{array}{l}(\forall 1 \leq i \leq n).\ \Gamma + (x : t_i) \vdash \mathtt{e} : s_i \\ (\forall 1 \leq j \leq m).\ t \not\leq t'_j \to s'_j \\ t = (t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n) \\ t' = \neg(t'_1 \to s'_1) \wedge \ldots \wedge \neg(t'_m \to s'_m)\end{array}}{\Gamma \vdash \lambda(x : t_1; \ldots ; t_n).\mathtt{e} : t \wedge t'}\ (abstr)$$

Cela pose un problème technique grave : le théorème d'élimination de la subsomption devient faux. Prenons $\mathtt{e}$ tel que $(x : t) \Vdash \mathtt{e} : s$ et $s \leq s'$ (mais pas $s' \leq s$). En appliquant d'abord la règle de subsomption puis celle-ci dessus, on peut donner à $\lambda(x : t).\mathtt{e}$ le type $(t \to s') \wedge \neg(t \to s)$. Mais si l'on n'interdit la règle de subsomption dans les prémisses, on ne peut évidemment pas donner à l'abstraction le type $\neg(t \to s)$.

Cela suggère de voir la nouvelle construction comme simplement une facilité syntaxique. Si l'on dispose d'un algorithme de typage (par exemple celui décrit dans la section 3.6), on remplace dans le programme, pendant la phase de typage, les abstractions de la nouvelle forme en des abstractions classiques, en complétant l'interface avec le résultat de l'algorithme de typage sur le corps $\mathtt{e}$. Suivant la précision de l'algorithme de typage, on donne des types différents à l'abstraction et cela *peut* changer la sémantique du programme (lorsqu'il teste si la fonction a comme type la négation d'une flèche ; cela semble peu courant dans la pratique de la programmation).

Il reste encore un problème : la règle de typage des abstractions impose de typer plusieurs fois le corps, sous des hypothèses différentes ; s'il y a dans celui-ci des abstractions non récursives, on peut leur attribuer un type de résultat égal à la réunion de ceux obtenus pour chaque typage.

5.4 Constructions dérivées

On peut définir du sucre syntaxique pour certaines constructions courantes, un peu lourdes à exprimer avec la syntaxe du langage.

Filtrage à n branches Dans le langage, on a le filtrage à 2 branches. On peut s'en servir pour définir le filtrage à n branches. Pour $n = 1$, on pose par exemple :

$$\begin{array}{l}\text{match } \mathtt{e} \text{ with } p_1 \longrightarrow \mathtt{e}_1 \equiv \\ \text{match } \mathtt{e} \text{ with } p_1 \to \mathtt{e}_1 \mid p_1 \to \mathtt{e}_1\end{array}$$

et pour $n \geq 3$, par récurrence :

$$\begin{array}{l}\text{match } \mathtt{e} \text{ with } p_1 \longrightarrow \mathtt{e}_1 \mid \ldots \mid p_n \longrightarrow \mathtt{e}_n \equiv \\ \text{match } \mathtt{e} \text{ with } p_1 \longrightarrow \mathtt{e}_1 \mid x \longrightarrow (\text{match } x \text{ with } p_2 \longrightarrow \mathtt{e}_2 \mid \ldots \mid p_n \longrightarrow \mathtt{e}_n)\end{array}$$

Liaison locale

$$\text{let } p = \mathtt{e}_1 \text{ in } \mathtt{e}_2 \equiv \text{match } \mathtt{e}_1 \text{ with } p \longrightarrow \mathtt{e}_2$$

Assertion de type

$$\text{assert}_t(\mathtt{e}) \equiv \text{match } \mathtt{e} \text{ with } x \wedge t \longrightarrow x$$

Cette construction est une assertion de type statique : elle demande au système de types de prouver que l'expression $\mathtt{e}$ a nécessairement le type t, et s'il n'en est pas capable, il donne une erreur de typage. Remarquons que grâce à la précision du calcul du type du résultat du filtrage, si l'on dispose d'un type plus précis pour $\mathtt{e}$, on n'oublie pas cette information lors du filtrage.

Contrainte de type

$$(\mathtt{e} : t) \equiv (\mu f : (t \to t).\lambda x.x)\mathtt{e}$$

Cette construction est un peu différente de l'assertion de type. Elle permet effectivement de contraindre le type de l'expression : non seulement le système de types doit vérifier que $\mathtt{e}$ a le type t, mais il doit aussi oublier toute autre information plus précise à son sujet.

Fonctions mutuellement récursives Supposons que l'on veuille définir n fonctions $f_1, \dots, f_n$ qui sont mutuellement récursives. Soit $\mathtt{e}_i$ le corps de f_i, et $(t_j \to s_j)_{j \in J_i}$ son interface. Posons $\tau_i = \bigwedge_{j \in J_i} t_j \to s_j$.

On suppose que l'on a à disposition n types singletons, notés $1, \dots, n$, avec $[\![i]\!] = \{i\}$ (par exemple, il peut s'agir de sous-types d'un type `int`). On peut alors poser :

$$\begin{array}{rcl} \mathtt{e} & = & \mu f(1 \to \tau_1; \dots; n \to \tau_n).\lambda x. \\ & & \quad \text{match } x \text{ with} \\ & & \quad\; \mid 1 \longrightarrow \mu f_1((t_j \to s_j)_{j \in J_1}).\lambda x.\mathtt{e}'_1 \\ & & \quad\; \dots \\ & & \quad\; \mid n \longrightarrow \mu f_1((t_j \to s_j)_{j \in J_n}).\lambda x.\mathtt{e}'_n \end{array}$$

où $\mathtt{e}'_i$ est obtenu en remplaçant dans $\mathtt{e}_i$ les références à f_j par $(f j)$.

Lorsque les différentes fonctions ont des domaines disjoints, ou lorsqu'elles ont le même comportement sur l'intersection des domaines, on peut simplement définir une seule fonction surchargée.

Chapitre 6

Conclusion et directions futures

6.1 Conclusion

Une algèbre de types complexe L'algèbre de types est complexe ; outre ce qu'on attend au minimum pour tout langage fonctionnel (types de base, produit, flèche), on a les types récursifs, les combinaisons booléennes. On a aussi une interprétation un peu inhabituelle du type flèche comme une simple contrainte, ce qui permet de typer les fonctions surchargées.

Ces caractéristiques n'ont pas été choisies au hasard ; nous n'avons pas cherché à les mettre ensemble pour le plaisir d'obtenir un système de type compliqué.

Au contraire, nous pensons que les possibilités offertes par l'algèbre apportent réellement une souplesse au programmeur, en mettant les types de son coté : les types ne sont plus seulement destinés à assurer l'absence d'une certaine classe de *bugs* dans les programmes ; le programmeur peut également les utiliser pour diriger les calculs.

L'interprétation des types comme ensembles de valeurs, qui semble très naturelle, pousse à introduire les combinaisons booléennes dans l'algèbre, et les types récursifs sont indispensables pour avoir des structures de données. La possibilité d'avoir des fonctions surchargées simplifie plutôt les choses, en donnant un sens concret aux intersections de types flèche.

Un système de types précis Il est important que le système de types soit assez précis et perde suffisamment peu d'information ; nous pensons que c'est le cas en pratique. Dans un sens, outre la perte d'information inhérente au typage, à savoir l'oubli des relations qui existent entre les valeurs associées aux identificateurs, il y a deux sources d'approximation :

- une fonction du langage est une boite noire pour le typage : de l'extérieur ou à l'intérieur, on n'a pas plus d'information que ce qui est donnée par son interface ;
- les opérateurs peuvent également perdre de l'information ; par exemple, si l'on travaille avec les types intervalles d'entiers, l'opérateur de multiplication va probablement vérifier $\times([1..2] \times [2..2]) \simeq [2..4]$, et pourtant le produit d'un entier dans $\{1, 2\}$ par 2 ne peut pas être 3.

Les exemples de code que l'on donne dans l'annexe A illustrent la précision du typage. Lorsque le système de types n'est pas assez précis, le programmeur a toujours la possibilité de faire des tests de type dynamiques pour raffiner le type des expressions manipulées.

Précision du système : des égalités et des équivalences L'approche sémantique a permi d'obtenir des égalités et des équivalences là où une axiomatisation incomplète aurait pu ne donner que des inclusions et des implications (suffisantes pour obtenir la sûreté du typage, mais moins satisfaisantes). En particulier, on a les équivalences remarquables :

$$
\begin{array}{lcl}
t \leq s & \Leftrightarrow & (\forall \vdash v : t.\ \vdash v : s) \\
\vdash v : t_1 \wedge t_2 & \Leftrightarrow & (\vdash v : t_1) \wedge (\vdash v : t_2) \\
\vdash v : t_1 \vee t_2 & \Leftrightarrow & (\vdash v : t_1) \vee (\vdash v : t_2) \\
\vdash v : \neg t & \Leftrightarrow & \not\vdash v : t \\
\vdash v : \wr p \int & \Leftrightarrow & v/p \neq \Omega \\
\vdash v : (t \Downarrow p)(x) & \Leftrightarrow & \exists \vdash v' : t.\ (v'/p)(x) = v \\
\vdash v : t_1.t_2 & \Leftrightarrow & \exists \vdash v_1 : t_1.\ \exists \vdash v_2 : t_2.\ v_1 v_2 \xrightarrow{*} v
\end{array}
$$

Ces équivalences peuvent permettre au programmeur d'avoir une meilleur compréhension des opérations qui interviennent dans le système de type. Elle permettent également au typeur d'expliquer ses messages d'erreur ; par exemple, si une relation de sous-typage $t \leq s$ n'est pas vraie, et rend le programme mal typé, le typeur peut exhiber une *valeur* du langage, qui est de type t mais pas de type s.

Approche sémantique Nous pensons que la complexité de l'algèbre de types ne devrait pas poser problème au programmeur, dans la mesure où il peut s'appuyer sur une interprétation ensembliste des opérations sur les types. Par exemple, on peut lui présenter la relation de sous-typage en disant « un type t est sous-type de s si toute expression de type t est aussi de type s, et en fait, il suffit de regarder les valeurs ». Le point délicat est probablement l'inclusion des types flèche ; nous pensons que l'interprétation de ces types comme des contraintes peut être comprise facilement, et donc la relation de sous-typage également.

Par contre, travailler avec une algèbre de types complexe aurait pu être problématique pour le traitement formel. Il serait par exemple possible de partir de l'algorithme de sous-typage, présenté peut-être pour compliquer encore les choses de manière plus classique sous forme de règles d'inférence (avec *memoization* explicite). Mais alors la moindre preuve qui fait intervenir la relation calculée devient difficile.

L'approche partiellement sémantique que l'on a adoptée est, à notre avis, simple, naturelle et efficace, même si le développement technique peu sembler long. Elle revient en fait à se mettre dans le peau du programmeur, et à raisonner le plus possible sur l'interprétation des types comme ensembles de valeurs. L'approche n'est que partiellement sémantique dans la mesure où l'on travaille toujours sur une algèbre de types symbolique, et où la sûreté du typage est obtenue par une preuve syntaxique ; ce choix a permis de garder la simplicité des preuves syntaxiques là où elles sont efficaces (induction sur la structure des dérivations de typage).

Et XML ? Le langage que nous avons étudié est généraliste. Par rapport à la représentation interne de XDuce :

- il n'y a rien de spécifique à XML ;
- nous avons explicitement toutes les combinaisons booléennes (intersection et complémentaire sont implicites dans l'algorithme pour le filtrage de XDuce)
- nous avons évidemment les fonctions comme objets de première classe ;
- les enregistrements permettent de représenter assez fidèlement les attributs XML.

Les exemples de code dans l'annexe A montrent qu'il est possible d'aller assez loin vers XML avec du simple sucre syntaxique sur les types et les motifs (pour écrire des types récursifs comme des expressions régulières). Néanmoins, pour avoir un langage vraiment adapté, disons, à l'écriture de transformateurs XML, il est préférable d'utiliser une représentation externe comme celle de XDuce. Il ne devrait pas y avoir de problèmes pour adapter celle-ci à notre langage.

6.2 Directions futures

Polymorphisme, inférence Un des points qui mérite encore le plus de travail dans le système de types est celui du polymorphisme paramétrique. Nous avons décrit et implémenté une approche simple, et signalé une piste pour aller vers le polymorphisme du système $F_{\leq}$. Cela reste à étudier.

On peut aussi chercher à mettre dans le système une certaine dose d'inférence de type (indépendamment du polymorphisme paramétrique, même) : certaines annotations de types dans les abstractions doivent pouvoir être évitées ; on pense d'une part aux fonctions qui sont définies par un filtrage sur lequel on peut lire le type de l'argument (si le filtrage est formulé avec assez de contraintes de types, on peut prendre $\lbag p_1 \rbag \vee \lbag p_2 \rbag$ comme type d'entrée), et d'autre part aux fonctions qui peuvent être typées en ML (fonction générique `map` sur les listes, par exemple).

Simplification des types, messages d'erreur Il y a de nombreuses heuristiques possibles pour simplifier les expressions de types. Par exemple, on peut enlever dans les réunions ou les intersections des termes qui n'ont pas d'influence ; on peut aussi pousser les connecteurs booléens sous les constructeurs. La simplification des expressions de type a deux objectifs :

- pour l'affichage des messages d'erreur, ou simplement du résultat, du typeur ;
- pour optimiser les performances du typeur (il peut être plus rapide dans certains cas de simplifier le type calculé pour une sous-expression avant de continuer le typage, même si la simplification prend elle-même du temps).

Concernant l'affichage des types, il est important de garder la trace des types nommés par le programmeur, en particulier, lorsqu'il s'agit de types récursifs.

Vers XML Nous avons presque tout ce qu'il faut pour manipuler les documents XML. Comme nous l'avons dit dans la section précédente, il reste à reprendre la traduction de la représentation externe (des valeurs, types et motifs) vers la représentation interne, présente dans XDuce. Dans l'implémentation que nous décrivons en annexe, nous travaillons avec la représentation interne, mais un peu de sucre syntaxique montre que cela est peut-être suffisant.

Extension du filtrage L'opération de filtrage est essentielle : elle permet de faire des requêtes complexes sur les objets, mais elle renvoie toujours des sous-objets de l'objet filtré. Nous pouvons chercher à étendre l'algèbre de motifs pour permettre au filtrage de reconstruire des objets ; un tel motif aurait la forme : $[p_1, p_2]$. Il opère en supposant que l'objet filtré est un couple ; il filtre chacune de ses composante avec le sous-motif p_i correspondant et, si une même variable x apparaît dans p_1 et p_2, il reconstruit un couple pour x à partir de ce que p_1 et p_2 ont extrait :

$$((e_1, e_2)/[p_1, p_2])(x) = ((e_1/p_1)(x), (e_2/p_2)(x))$$

Le motif

$$p = [x \wedge t, p] | (\_, p) | x$$

permet ainsi d'extraire tous les éléments d'une liste qui ont un certain type.

La difficulté technique est pour calculer $(t \Downarrow p)$ en présence de ce nouveau motif. En effet, si $x \in V(p_1) \cap V(p_2)$, on veut avoir :

$$(t_1 \times t_2 \Downarrow [p_1, p_2])(x) = (t_1 \Downarrow p_1)(x) \times (t_2 \Downarrow p_2)(x)$$

Mais le membre droit n'est pas « stable » : il introduit un nouveau constructeur. Cependant, cela ne doit pas poser trop de problème ; pour calculer $(t \Downarrow [p_1, p_2])(x)$, on crée un nouveau type t', pas encore défini, et on continue le calcul en se souvenant que $(t \Downarrow [p_1, p_2])(x)$ vaut t', ce qui assure la terminaison (on ne regarde jamais à l'« intérieur » des résultats déjà calculés), et on peut enfin définir t'. Il reste à formaliser cela.

Modèle d'exécution, compilation Nous n'avons pas du tout étudié le problème de l'évaluation ou de la compilation des programmes. Le filtrage, implémenté de manière naïve, peut se révéler extrêmement inefficace. Une première chose à faire est de transformer les filtrages en automates déterministes pour éviter les retours arrières. On peut utiliser les informations fournies par le typage pour savoir que certains tests de type sont inutiles à l'exécution (pour tirer parti au mieux des ces informations, il peut être nécessaire de changer l'ordre dans lequel on considère les deux sous-motifs des motifs conjonction).

Outre les filtrages, il y a des tests de type à l'entrée des fonctions ; on a déjà expliqué comment les supprimer dans la plupart des cas (lorsque le typage ne donne pas le type 1 à une application).

Les tests de type ont besoin de l'algorithme de sous-typage, pour pouvoir décider si une fermeture a un certain type. Si l'on s'interdit de mettre dans les motifs des contraintes avec des types flèche, on n'en a plus besoin, et on peut aussi supprimer l'information de type associée aux abstractions à l'exécution.

Une analyse de flot assez simple doit permettre de garder le cas général, en optimisant les cas usuels.

Programmation en grand Les langages modernes proposent des mécanismes pour structurer les programmes : langage de modules et/ou couche objet. Il y a probablement pas mal de travail à faire pour intégrer de tels systèmes à notre langage ; en particulier, il faut adapter l'approche sémantique pour prendre en compte les types abstraits, pour lesquels la relation de sous-typage ne doit pas être structurelle.

Annexe A

Implémentation

Pour tester les idées présentées dans ce rapport, j'ai implémenté un prototype de typeur et d'interpréteur pour le langage (le tout fait environ 5000 lignes de code). Il est écrit dans le langage Objective Caml [Ler01], qui s'est révélé particulièrement adapté à cette réalisation.

Mon travail a porté uniquement sur la partie typage du langage, je n'ai pas du tout étudié les problèmes d'interprétation et *a fortiori* de compilation. L'interpréteur est en conséquence tout à fait naïf, et il ne sert qu'à fin d'illustration.

A.1 Performances

Le typeur n'a pas été écrit particulièrement avec un soucis d'efficacité, mais il utilise néanmoins un certain nombre de techniques qui le rendent assez efficace. En particulier, il garde en cache les résultats de l'algorithme de sous-typage déjà prouvés ou infirmés (cependant, dans certains cas, la recherche en cache, implémenté comme une table de hachage, prend plus de temps que cela ne permet d'en gagner, et il vaut mieux désactiver le cache). Il utilise également la technique d'élagage dans l'étude du sous-typage des types produit et flèche. L'optimisation que l'on ne peut pas se permettre d'oublier, c'est de supprimer parmi les intersections qui constituent les formes normales disjonctives des combinaisons booléennes dans chaque univers, celles où un type atomique apparaît en même temps positivement et négativement (l'intersection est alors vide).

À un niveau plus bas, nous avons choisi d'utiliser autant que possible les fonctions génériques de comparaison et de hachage d'Objective Caml. Celles-ci sont implémentées en C, et on gagne beaucoup dans la mesure où les algorithmes doivent manipuler un grand nombre de structures de type ensembles (implémentés comme des listes ordonnées ou des arbres équilibrés) ou table de hachage. Ces fonctions ne sont pas prévues pour gérer les valeurs cycliques.

Nous n'avons pas mené d'étude détaillée et comparative des performances du typeur, mais, en tout cas sur les exemples que l'on a essayé, elles semblent plus qu'acceptable en pratique.

A.2 Les types

Le prototype implémente les enregistrements et le polymorphisme paramétrique. Les types de base sont :
- les intervalles d'entiers (machine) ;
- les singletons chaînes de caractère et l'ensemble de toutes les chaînes ;
- de même pour les « atomes » qui sont en fait une simple copie disjointe de l'ensemble des chaînes (avec des restrictions syntaxiques), utilisés comme *tag*, comme constructeur pour simuler les types construits de ML, ...

Le *parser* offre un mécanisme pour définir des abréviations de types ou de motifs. Cela est particulièrement utile pour définir des types récursifs. Il est également possible d'introduire des types et motifs récursifs par une notation de point fixe. Plus intéressant : une syntaxe spécifique est proposée pour définir des *expressions régulières* de types. En interne, cependant, tous les types utilisent une représentation directement dérivée de la présentation de l'algèbre de types avec variables. Cela explique que l'affichage de certains types ne ressemble pas du tout aux expressions de type fournies dans le programme.

A.3 Aperçu de l'implémentation de l'algèbre de types

Pour représenter les types récursifs, on utilise une indirection : on maintient une « base » de définition courante ; les types de cette base sont manipulés comme des entiers, et leur description est donnée par un tableau. Cela permet d'une part de créer ces objets (on peut attribuer un entier au type avant de connaître sa définition), et d'autre part de ne pas avoir de valeurs cycliques. On procède de même pour les motifs. Pour donner une idée, les types Caml utilisés pour représenter l'algèbre de types sont donnés par (pour simplifier, on donne les définitions sans enregistrements et sans variables de type) :

```
type t     = int  (* Types de la base courante X *)
and  descr =      (* Types X-définissables *)
    {
      basic   : basic;
      product : product bool;
      arrow   : arrow   bool
    }
and basic   = Consts.Type.descr
and product = t * t
and arrow   = t * t
and 'a bool = ('a list * 'a list) list
```

Pour les types de base, une représentation spécifique et optimisée est utilisée ; par exemple, dans le « sous-univers » des entiers, on manipule directement des réunions finies d'intervalles (cette classe est stable par combinaison booléenne).

A.4 Quelques exemples

Remarque 40. *Dans ce qui suit, les lignes précédées de* `>` *sont celles du programme ; les réponses du typeur sont préfixée par* `|-` *; les lignes précédées de* `=>`

sont les réponses de l'évaluateur.

A.4.1 Entiers, intervalles

On dispose des types intervalles d'entiers [a..b], donc en particulier des types singletons ; les constantes entières et les opérateurs arithmétiques ont donc un typage très précis.

```
> 5
|- 5
=> 5

> -3
|- -3
=> -3

> \incr ([0..100] -> [5..105]). x -> x + 5
|- [0..100] -> [5..105]
=> <fun>

> \incr ([0..100] -> [0..100]). x -> x + 5
Type error (file code/entiers.x; line 10, chars 35..40)
This expression has type
[5..105]
which is not subtype of
[0..100]
Residual type: [101..105]
Annotation in abstraction interface not satisfied

> \wrap ([0..100] -> [0..10]). x -> match x with
>       | [0..10] -> x
>       | _ -> wrap (x - 10)
> end
Type warning (file code/entiers.x; line 14, chars 13..26)
Application yields FULL; type of argument:
[-10..90]
Type error (file code/entiers.x; line 12, char 34 .. line 15, char 3)
This expression has type
FULL
which is not subtype of
[0..10]
Residual type: ~[0..10]
Annotation in abstraction interface not satisfied
```

Il faut réecrire cette fonction pour pouvoir utiliser dans le corps des motifs l'information précise obtenue par l'algorithme de calcul des résultats de filtrage.

```
> (\wrap ([0..100] -> [0..10]). x -> match x with
```

```
>       | x & [0..10] -> x
>       | x -> wrap (x - 10)
> end) 98
|- [0..10]
=> 8

> type int = [min_int..max_int]
> type nat = [0..max_int]
> \abs (int->nat). x -> match x with
>       | x & nat -> x
>       | x -> 0 - x
> end
Type error (file code/entiers.x; line 32, char 22 .. line 35, char 3)
This expression has type
int
which is not subtype of
nat
Residual type: [min_int..-1]
Annotation in abstraction interface not satisfied
```

Le problème, c'est qu'avec la représentation machine, pour OCaml, `0 - min_int = -min_int` ; le système de types, en cas de débordement, ne donne aucune information sur le résultat, ce qui permet de détecter l'erreur !

```
> \abs (int->nat). x -> match x with
>       | x & nat -> x
>       | min_int -> max_int
>       | x -> 0 - x
> end
|- int -> nat
=> <fun>
```

Si l'on en a besoin, on peut donner un type plus précis.

```
> (\abs (int->nat; [-20..20]->[0..20]). x -> match x with
>       | x & nat -> x
>       | min_int -> max_int
>       | x -> 0 - x
> end) (-15)
|- [0..20]
=> 15
```

On peut définir des fonctions mutuellement récursives.

```
> \f (1 -> (int->int); 2 -> (int->int)). x -> match x with
>  | 1 ->
>      \f1 (int->int). x ->
>      (f 2) (x - 10)
```

```
>   | 2 ->
>       \f2 (int->int). x ->
>       (match x with nat -> (f 1) x | _ -> x end)
> end
|- 1 -> int -> int & 2 -> int -> int
=> <fun>
```

A.4.2 Types et motifs récursifs

On peut définir une liste d'entiers par un type récursif. Contrairement à ML, il ne s'agit *pas* d'un type somme taggée. Le type `<null>` est un type singleton qui possède un seul élément, l'atome `<null>`.

```
> type int = [min_int..max_int]
> type nat = [0..max_int]
> type int_list = (int * int_list) | <null>
```

On peut utiliser une fonction récursive pour extraire le premier entier positif d'une liste.

```
> \first_pos (int_list -> nat). l -> match l with
>   | ((x & nat),_) -> x
>   | (_,l) -> first_pos l
> end
Type error (file code/recurs.x; line 16, char 35 .. line 19, char 3)
Pattern matching not exhaustive; residual type:
<null>
```

On a oublié le cas de la liste vide (donc de celles qui ne contiennent pas d'entier positif) ! On choisit de retourner une valeur particulière pour signaler l'erreur.

```
> \first_pos (int_list -> (nat | <nf>)). l -> match l with
>   | ((x & nat),_) -> x
>   | (_,l) -> first_pos l
>   | <null> -> <nf>
> end
|- int_list -> ([0..max_int] | <nf>)
=> <fun>
```

Il faut faire attention à l'ordre des branches. Dans l'exemple qui suit, la deuxième branche est inutile, car masquée par la première.

```
> \first_pos (int_list -> (nat | <nf>)). l -> match l with
>   | (_,l) -> first_pos l
>   | ((x & nat),_) -> x
>   | <null> -> <nf>
> end
```

```
Type warning (file code/recurs.x; line 37, chars 21..22)
Unused branch in pattern matching
|- int_list -> ([0..max_int] | <nf>)
=> <fun>
```

En fait, l'extraction peut se faire avec un seul motif récursif (introduit par un opérateur de point fixe $\backslash p$, qui se lit μp ; on a la même notation pour les types).

```
> \first_pos (int_list -> nat). l -> match l with
>   | \p. ((x & nat,_) | (_,p)) -> x
> end
Type error (file code/recurs.x; line 45, char 35 .. line 47, char 3)
Pattern matching not exhaustive; residual type:
<null> |
int * int_list & ~(FULL * (\a.(FULL * a | nat * FULL))) &
~(nat * FULL)
```

Le système signale que l'on a oublié le cas de toutes les listes d'entiers sans entier positif (dans un message d'erreur peu lisible, certes, mais c'est bien son sens). Ici, on choisit de restreindre le domaine aux bonnes listes.

```
> type one_nat = (nat * FULL) | (FULL * one_nat)
> \first_pos ((int_list & one_nat) -> nat). l -> match l with
>   | \p. (( (x & nat) ,_) | (_,p)) -> x
> end
|- (int * int_list & nat * FULL |
   int * int_list & FULL * one_nat) -> nat
=> <fun>
```

L'ordre dans les motifs alternatives à une importance (la notation de n-uplet est du sucre syntaxique pour des couples imbriqués, avec associativité à droite de la virgule).

```
> match (-15,-5,5,15,25) with \p. ((x & nat),_) | (_,p) -> x end
|- 5
=> 5

> match (-15,-5,5,15,25) with \p. ((_,p) | (x & nat),_) -> x end
|- 15
=> 15
```

A.4.3 Fonctions surchargées

Les fonctions surchargées sont surtout intéressantes pour la programmation « en grand ». L'exemple suivant montre un exemple simple de surcharge d'opérateur et le gain de précision par rapport au filtrage de XDuce (où toutes les branches doivent avoir le même type, ce qui correspond à `neg2`).

```
> type bool = <true> | <false>
> type int  = [min_int..max_int]
> type int_bool = int | bool
> let neg = \neg (bool -> bool; int -> int). x ->
>   match x with
>     | <true>  -> <false>
>     | <false> -> <true>
>     | n -> 0 - n  (* ici, on sait que n:int *)
>   end
> in
> let neg2 = \neg2 (int_bool -> int_bool). x -> neg x in
> (neg 1), (neg <true>), (neg2 1), (neg2 <true>)
|- int * (bool * (int_bool * int_bool))
=> (-1, (<false>, (-1, <false>)))
```

On peut oublier avec la subsomption qu'une fonction est surchargée, soit en oubliant des branches, soit en les fusionnant :

```
> let neg = \neg (bool -> bool; int -> int). x ->
>   match x with
>     | <true>  -> <false>
>     | <false> -> <true>
>     | n -> 0 - n  (* ici, on sait que n:int *)
>   end
> in
> function constraint (X;EMPTY) (X->X). x -> x in
> neg,
> ([constraint with X = bool -> bool] neg),
> ([constraint with X = int_bool -> int_bool] neg)
|- (bool -> bool & int -> int) *
   ((bool -> bool) * (int_bool -> int_bool))
=> (<fun>, (<fun>, <fun>))
```

Le cas où les différents domaines qui apparaissent dans l'interface ne sont pas disjoint est intéressant :

```
> type a = [1..10]
> type b = [5..20]
> type c = [14..25]
> let f = \f (a -> (0|1); b -> (0|2); c -> (2|3)). x ->
>   match x with
>     | a & b -> 0
>     | a -> 1
>     | b -> 2
>     | _ -> 3
>   end
```

```
> in
> (f 2), (f 8), (f 13), (f 15), (f 23), (f 30)
Type warning (file code/overload.x; line 57, chars 39..43)
Application yields FULL; type of argument:
30
|- [0..1] * (0 * ((0 | 2) * (2 * ([2..3] * FULL))))
=> (1, (0, (2, (2, (3, <reject>)))))
```

A.4.4 Enregistrements

Voici quelques exemples de manipulation d'enregistrements. Dans les types affichés, `[ $ x ;y ]` est la contrainte que l'on notait $\mathbf{S}(\{x, y\})$. Dans les types donnés par le programmeur, on combine les contraintes (un type enregistrement avec un `$` signifie qu'il est strict).

```
> { x = 2 ; y = "Hello, world !" }
|- [ $ x;y ] & [ x: 2 ] & [ y: "Hello, world !" ]
=> { x = 2; y = "Hello, world !" }

> type int = [min_int..max_int]
> (\f ([x:int]->[$ x:int]). r -> { x = r.x + 1})
>   { x=2; y="Hello world !" }
|- [ $ x ] & [ x: int ]
=> { x = 3 }

> (\f ([x:int]->[x:int]). r -> r -- x ++ { x = r.x + 1})
>   { x=2; y="Hello world !" }
|- [ x: int ]
=> { x = 3; y = "Hello world !" }

> let f = \f ([ x:int ]->int). r ->
>   match r with
>      | { y = y & int } -> r.x + y
>      | _ -> r.x
>   end
> in
> (f { x=2 }), (f { x=2; y=3 }), (f {x=2; y="Hello"})
|- int * (int * int)
=> (2, (5, 2))
```

On peut spécifier qu'un champ est soit absent, soit d'un certain type. Notons ce que donne l'application d'une fonction à un argument qu'elle ne sait pas traiter.

```
> let f = \f ([ x:int; y:(int|<undef>) ]->int). r ->
>   match r with
>      | { y = y & int } -> r.x + y
>      | _ -> r.x
```

```
>   end
> in
> (f { x=2 }), (f { x=2; y=3 }), (f {x=2; y="Hello"})
Type warning (file code/record.x; line 39, chars 32..50)
Application yields FULL; type of argument:
[ $ x;y ] & [ x: 2 ] & [ y: "Hello" ]
|- int * (int * FULL)
=> (2, (5, <reject>))

> type homme =
>   [ $ sexe : "M"; age : int; nom : string ]
> type femme =
>   [ $ sexe : "F"; age : int; nom_jf : string; epouse : string|<undef>
> ]
> type personne = homme | femme
> type femme_mariee = femme & [ epouse : string ]
> type femme_non_mariee = femme & ~ femme_mariee
> \age (personne->int). p -> p.age
|- personne -> int
=> <fun>

> \nom (personne->string). p -> match p with
>   | p & { sexe = "M" } -> p.nom (* ici, on sait que  |- p: homme *)
>   | p -> p.nom_jf              (* donc ici, |- p: femme *)
> end
|- personne -> string
=> <fun>

> \nom (personne->string). p -> match p with
>   | { nom_jf = n  } -> n (* ici, on sait que  |- p: femme *)
>   | p -> p.nom           (* donc ici, |- p: homme *)
> end
Type error (file code/record.x; line 60, char 30 .. line 63, char 3)
This expression has type
string | <undef>
which is not subtype of
string
Residual type: <undef>
Annotation in abstraction interface not satisfied
```

On s'est trompé ! Le premier motif accepte aussi le cas où `nom_jf=<undef>` ; on doit ajouter une contrainte de type pour interdire ce cas.

```
> \nom (personne->string). p -> match p with
>   | { nom_jf = n & ~<undef> } -> n (* ici, on sait que  |- p: femme *)
>   | p -> p.nom                       (* donc ici, |- p: homme *)
> end
|- personne -> string
=> <fun>
```

Ce qui peut s'écrire aussi avec un `:=` :

```
> \nom (personne->string). p -> match p with
>   | { nom_jf := n  } -> n    (* ici, on sait que  |- p: femme *)
>   | p -> p.nom                        (* donc ici, |- p: homme *)
> end
|- personne -> string
=> <fun>
```

Il est peut-être plus clair de procéder ainsi :

```
> \nom (personne->string). p -> match p with
>   | p & homme -> p.nom
>   | p & femme -> p.nom_jf
> end
|- personne -> string
=> <fun>
```

Ou encore :

```
> \nom (personne->string). p -> match p with
>   | { nom := n } | { nom_jf := n } -> n
> end
|- personne -> string
=> <fun>

> \nom_civil (personne->string). p -> match p with
>   | { epouse := n } | { nom := n } | { nom_jf := n } -> n
> end
|- personne -> string
=> <fun>

> \epouse ((homme * femme) -> femme). x ->
>   let (h,f) = x in
>   f ++ { epouse = h.nom }
Type error (file code/record.x; line 100, chars 2..25)
This expression has type
femme
which is not subtype of
[ epouse: <undef> ]
Residual type: femme_mariee
field epouse must provably be <undef>
```

Il faut en effet supposer qu'un champ n'est pas défini pour l'ajouter !

```
> \epouse ((homme * (femme & [epouse:<undef>] )) -> femme). x ->
>   let (h,f) = x in
```

```
>   f ++ { epouse = h.nom }
|- homme * femme_non_mariee -> femme
=> <fun>
```

Ou alors, on enlève le champ avant de le rajouter :

```
> \epouse ((homme * femme) -> femme). x ->
>   let (h,f) = x in
>   let f = match f with
>      | f & { epouse = <undef> } -> f
>      | f -> f -- epouse
>   end in
>   f ++ { epouse = h.nom }
|- homme * femme -> femme
=> <fun>
```

A.4.5 Séquences et expressions régulières

Les n-uplets sont codés par des couples, avec associativité à droite.

```
> (1,2,3)
|- 1 * (2 * 3)
=> (1, (2, 3))
```

On représente ainsi les listes, en choisissant comme terminateur l'atome `<null>`.

```
> (1,2,3,<null>)
|- 1 * (2 * (3 * <null>))
=> (1, (2, (3, <null>)))
```

On a une notation spécifique pour cela :

```
> [1 2 3]
|- 1 * (2 * (3 * <null>))
=> (1, (2, (3, <null>)))
```

Pour ajouter des éléments en tête, on peut préciser une terminaison :

```
> let a = [4 5 6] in
> [1 2 3 ; a ]
|- 1 * (2 * (3 * (4 * (5 * (6 * <null>)))))
=> (1, (2, (3, (4, (5, (6, <null>))))))
```

On a la même notation pour les types :

```
> type int=[min_int..max_int]
> print type [string int string]
string * (int * (string * <null>))
```

En fait, on peut écrire une expression régulière de types :

```
> type bool = <true> | <false>
> type string_list = [string*]
> print type [(int | bool+) string*]
int * string_list |
bool * (\a.(<null> | string * string_list | bool * a))
```

On peut aussi préciser une terminaison :

```
> print type [int?]
<null> | int * <null>
> print type [int?; string_list]
<null> | int * string_list | string * string_list
> print type [int?; string]
string | int * string
```

Pour les motifs, c'est la même chose (les accolades permettent d'écrire un motif complexe à l'intérieur d'une expression régulière de motif ; la commande `accept` affiche le type accepté par un motif) :

```
> accept [int* {x & string} int* ]
\A.((:int: , A) |
    ($x & :string: , (\B.((:int: , B) | :<null>:)))) accepts
\a.(int * a | string * (\b.(<null> | int * b)))

> let s = [1 2 3 4 "Hello" 5 6 7 "World" 9 10] in
> match s with [int* {x & string} ; _ ] -> x end,
> match s with [int* {x & string} (int|string)* ] -> x end
|- "Hello" * "Hello"
=> ("Hello", "Hello")
```

On peut capturer des éléments individuels d'une liste, mais pas une sous-séquence ; seule exception : la queue. Remarquons la précision de l'algorithme de calcul du type du résultat du filtrage.

```
> match [1 2 3 4 "Hello" 5 6 7 "World" 9 10] with
>   [int* string ; q ] -> q
> end
|- 5 * (6 * (7 * ("World" * (9 * (10 * <null>)))))
=> (5, (6, (7, ("World", (9, (10, <null>))))))
```

L'étoile dans les expressions régulières reconnaît la plus longue séquence possible ; on a une variante qui reconnaît la séquence la plus courte (on donne la forme équivalente).

```
> let s = [1 2 3 4 "Hello" 5 6 7 "World" 9 10] in
> match s with [_* {x & string} ; _ ] -> x end,
> match s with \p. (FULL , p) | ((x & string) , FULL) -> x end,
>
> match s with [_** {x & string} ; _ ] -> x end,
> match s with \p. ((x & string) , FULL) | (FULL , p) -> x end
|- "World" * ("World" * ("Hello" * "Hello"))
=> ("World", ("World", ("Hello", "Hello")))
```

Les motifs ne sont pas assez puissants pour reconstruire des valeurs ; on doit utiliser des fonctions récursives.

```
> (\filter_string ([(int|string)*]->[string*]). l ->
>   match l with
>     | [_** {x & string} ; q] -> (x, (filter_string q))
>     | _ -> <null>
>   end)
> [ 1 2 3 4 "Hello" 5 6 7 "World" 9 10]
|- string_list
=> ("Hello", ("World", <null>))

> (\filter_string ([(int|string)*]->[string*]). l ->
>   match l with
>     | ({x & string}, q) -> (x, (filter_string q))
>     | (_,q) -> filter_string q
>     | <null> -> <null>
>   end)
> [ 1 2 3 4 "Hello" 5 6 7 "World" 9 10]
|- string_list
=> ("Hello", ("World", <null>))
```

A.4.6 Polymorphisme paramétrique

Exemple classique : la fonction `map` sur les listes.

```
> type int = [min_int..max_int]
> function map (X,Y; EMPTY) ((X->Y) -> ([X*]->[Y*])). f ->
>   \aux ([X*]->[Y*]). l -> match l with
>       | (h,t) -> ((f h), (aux t))
>       | <null> -> <null>
>   end
> in
> let map_int_int = [map with X=int, Y=int] in
> map_int_int (\f (int->int). x -> x + 1) [1 2 3]
|- \a.(<null> | int * a)
=> (2, (3, (4, <null>)))
```

Extraire x_n d'une séquence $(x_1, (x_2, \ldots, (x_{n-1}, x_n) \ldots))$

```
> function last (X; EMPTY) ( (\t.FULL*t | X) -> X). l ->
>   match l with \p.((_,p)|x) -> x end
> in
> [last with X=int]
Type error (file code/poly.x; line 17, chars 2..36)
This expression has type
~1_prod
which is not subtype of
X
Residual type: ~1_prod & ~X
Annotation in abstraction interface not satisfied
```

Pour que cela marche, il suffit qu'il n'y ait pas de couples dans X :

```
> function last (X; X & (FULL*FULL)) ( (\t. FULL*t | X) -> X). l ->
>   match l with \p.((_,p)|x) -> x end
> in
> [last with X=int]
|- (\a.([min_int..max_int] | FULL * a)) -> int
=> <fun>
```

L'instanciation des variables de types joue sur la sémantique :

```
> function is_in (X; EMPTY) (X -> <true>; ~X -> <false>). x ->
>   match x with X -> <true> | _ -> <false> end
> in
> let is_int = [is_in with X=int] in
> (is_int 1), (is_int "Hello")
|- <true> * <false>
=> (<true>, <false>)
```

A.4.7 Vers XML

Voyons comment utiliser le langage pour représenter et manipuler des documents XML. On commence avec un exemple repris de [Hos01]. Contrairement à XDuce, on choisit de mettre l'accent sur les éléments, et non pas les séquences d'éléments. Un élément de tag t et dont la liste des éléments fils est l est représenté par le couple (t, l). Avec les notations que l'on a expliqué plus haut, on peut écrire :

```
> type Addrbook = [<addrbook> Person*]
> type Person   = [<person> Name Email* Tel?]
> type Name     = [<name> string]
```

```
> type Email    = [<email> string]
> type Tel      = [<tel> string]
> type Telbook  = [<telbook> Entry*]
> type Entry    = [<entry> Name Tel]
> let telbook = \telbook ([Person*] -> [Entry*]). ps ->
>  match ps with
>    | ( [<person> [<name> n] Email* [<tel> t]] , rest ) ->
>         ( [<entry> [<name> n] [<tel> t]] ) , ( telbook rest )
>    | ( [<person> [<name> n] Email*] , rest ) ->
>         telbook rest
>    | <null> -> <null>
>  end
> in
> \convert (Addrbook -> Telbook). x ->
>  let [<addrbook>; persons] = x in
>  [<telbook>; (telbook persons)]
|- Addrbook -> Telbook
=> <fun>

> type Enum  = [<ul> Line*]
> type Line  = [<li> Doc]
> type Doc   = [(string | Enum)*]
> \flatten ((Doc | [Line*]) -> string). x ->
>   match x with
>     | <null> -> ""
>     | ((s & string), rest) -> s ^ (flatten rest)
>     | ([<ul>; l], rest)  -> (flatten l) ^ (flatten rest)
>     | ([<li> d], rest)   -> (flatten d) ^ (flatten rest)
>   end
|- (<null> | Enum * Doc | Line * (\a.(<null> | Line * a)) |
   string * Doc) -> string
=> <fun>
```

Annexe B

Annexes techniques

B.1 Inclusions ensemblistes

Fait B.1.0.1. *Soit* $(X_i)_{1\leq i\leq n}, (Y_i)_{1\leq i\leq n}, (X'_j)_{1\leq j\leq m}, (Y'_j)_{1\leq j\leq m}$ *des familles d'ensembles. Alors :*

$$\begin{array}{ll} & \bigcap X_i \times Y_i \subseteq \bigcup X'_j \times Y'_j \\ \iff & \forall J \subseteq \{1,\dots,m\}.(\bigcap X_i \subseteq \bigcup_{j\in J} X_j) \vee (\bigcap Y_i \subseteq \bigcup_{j\notin J} Y'_j) \end{array}$$

Plus précisément, $(e_1,e_2) \in (\bigcap X_i \times Y_i)\backslash(\bigcup X'_j \times Y'_j)$ *si et seulement si :*

$$\exists J \subseteq \{1,\dots,m\}.e_1 \in (\bigcap X_i \backslash \bigcup_{j\in J} X'_j) \wedge e_2 \in (\bigcap Y_i \backslash \bigcup_{j\notin J} Y'_j)$$

Fait B.1.0.2. *Soit* $(X_i)_{1\leq i\leq n}, (X'_j)_{1\leq j\leq m}$ *des familles d'ensembles. Alors :*

$$\begin{array}{ll} & \bigcap \mathcal{P}_f(X_i) \subseteq \bigcup \mathcal{P}_f(X'_j) \\ \iff & \exists j. \bigcap X_i \subseteq X'_j \end{array}$$

B.2 Opérateur de projection

Théorème B.2.0.3 (Opérateur de projection). *Tout prémodèle induit sur l'algèbre de types un opérateur* $\pi_1 : \mathbf{T} \longrightarrow \mathbf{T}$ *vérifiant :*

$$[\![\pi_1(t)]\!] = \{e_1 \mid \exists e_2.\ (e_1,e_2) \in [\![t]\!]\}$$

De plus, cet opérateur est stable et deux prémodèles équivalents donnent le même opérateur.

Démonstration. On peut caractériser l'ensemble X à droite de l'égalité comme la plus petite solution de l'inéquation :

$$[\![t]\!] \cap \mathcal{D}_{\mathbf{product}} \subseteq X \times \mathcal{D}$$

Avec l'étude de la relation de sous-typage dans l'univers des types produits, on peut exprimer cette plus petite solution ; c'est :

$$\bigcup_{(P,N)\in t_{\mathbf{product}}} \left(\bigcap_{(t_i \times s_i)\in P} [\![t_i]\!] \backslash (\bigcap_{N'\subseteq N} \bigcup_{(t'_j \times s'_j)\in N'} [\![t'_j]\!]) \right)$$

(la réunion est prise sur les N' tels que : $\bigcap [\![s_i]\!] \not\subseteq \bigcup_{(t'_j \times s'_j) \in N \setminus N'} [\![s'_j]\!]$). On pose donc :

$$\pi_1(t) = \bigvee_{(P,N) \in t_{\mathbf{product}}} \left(\bigwedge_{(t_i \times s_i) \in P} t_i \wedge \neg (\bigwedge_{N' \subseteq N} \bigvee_{(t'_j \times s'_j) \in N'} t'_j) \right)$$

(la réunion est prise sur les N' tels que : $\bigwedge s_i \not\leq \bigvee_{(t'_j \times s'_j) \in N \setminus N'} s'_j$). On voit que $\pi_1(t)$ est construit à partir de types qui sont dans le socle de t, ce qui assure que π_1 est stable. □

B.3 Opérateur d'application

Fait B.3.0.4. *Dans tout modèle, pour $t_{\mathbf{arrow}}$ et t fixés, l'inéquation $t_{\mathbf{arrow}} \leq t \to s$ admet une plus petite solution s, que l'on note $t_{\mathbf{arrow}}.t$.*

L'opérateur $\_.\_ : \mathbf{T_{arrow}} \times \mathbf{T} \longrightarrow \mathbf{T}$ est croissant en chaque argument et il ne dépend que de la classe d'équivalence du modèle (en tant que prémodèle).

Démonstration. L'équivalence dans la définition des modèles permet d'étudier cette inéquation. On a successivement :

$$\begin{array}{ll} & t_{\mathbf{arrow}} \leq t \to s \\ \Leftrightarrow & \forall (P,N) \in t_{\mathbf{arrow}}.\ (\bigwedge_P t_i \to s_i) \wedge (\bigwedge_N \neg(t'_j \to s'_j)) \leq t \to s \\ \Leftrightarrow & \forall (P,N) \in t_{\mathbf{arrow}}.\ \bigwedge_P t_i \to s_i \leq \bigvee_N t'_j \to s'_j \vee t \to s \\ \Leftrightarrow & \forall (P,N) \in t_{\mathbf{arrow}}.\ (\exists (t'_j \to s'_j) \in N. \bigwedge_P t_i \to s_i \leq t'_j \to s'_j) \vee \\ & \qquad\qquad\qquad\qquad (\bigwedge_P t_i \to s_i \leq t \to s) \\ \Leftrightarrow & \forall (P,N) \in t_{\mathbf{arrow}}.\ (\bigwedge_P t_i \to s_i \leq \bigvee_N t'_j \to s'_j) \vee \\ & \qquad\qquad (\forall P' \subseteq P.\ (\bigwedge_{(t_i \to s_i) \in P'} s_i \leq s) \vee (t \leq \bigwedge_{(t_i \to s_i) \in P \setminus P'} t_i)) \end{array}$$

On voit que l'on obtient la plus petite solution pour s en prenant :

$$t_{\mathbf{arrow}}.t = \bigvee_{(P,N) \in t_{\mathbf{arrow}}} \left(\bigvee_{P' \subseteq P} \bigwedge_{(t_i \to s_i) \in P'} s_i \right)$$

(la première réunion est prise sur les (P,N) tels que $(\bigwedge_P t_i \to s_i) \wedge \neg(\bigvee_N t'_j \to t'_j) \not\simeq 0$; la deuxième est prise sur les $P' \subseteq P$ tels que $t \not\leq \bigvee_{(t_i \to s_i) \in P \setminus P'} t_i$). Pour voir que l'opérateur $\_.\_ : \mathbf{T_{arrow}} \times \mathbf{T} \longrightarrow \mathbf{T}$ est croissant en chaque argument, on raisonne avec la caractérisation comme plus petit solution d'une inéquation. □

Remarque 41. *On constate que l'opérateur est stable ; on ne se servira pas de cette propriété.*

B.4 Filtrage des types

Nous prouvons dans cette section le théorème suivant :

Théorème B.4.0.5. *Pour tout couple compatible (t,p) et toute variable $x \in V(p)$, il existe un type $(t \Downarrow p)(x)$ tel que :*

$$[\![(t \Downarrow p)(x)]\!] = \{(e/p)(x) \mid e \in [\![t]\!]\}$$

On part des égalités :

$$\begin{array}{lcll}
(t \downarrow x)(x) & = & t & \\
(t \downarrow p_1 \wedge p_2)(x) & = & (t \downarrow p_i)(x) & \text{si } x \in V(p_i) \\
(t \downarrow p_1|p_2)(x) & = & (t \wedge \wr p_1 \int \downarrow p_1)(x) \cup (t \wedge \neg \wr p_1 \int \downarrow p_2)(x) & \\
(t \downarrow (p,_))(x) & = & (\pi_1(t) \downarrow p)(x) & \\
(t \downarrow (_,p))(x) & = & (\pi_2(t) \downarrow p)(x) &
\end{array}$$

On voit que si dans le membre gauche, le couple est compatible, alors c'est encore le cas dans le membre droit. Cela suggère d'introduire une réduction sur les types compatibles, définie par :

$$\begin{array}{lcl}
(t, p_1 \wedge p_2) & \rightsquigarrow & (t, p_1), (t, p_2) \\
(t, p_1|p_2) & \rightsquigarrow & (t \wedge \wr p_1 \int, p_1), (t \wedge \neg \wr p_1 \int, p_2) \\
(t, (p,_)) & \rightsquigarrow & (\pi_1(t), p) \\
(t, (_,p)) & \rightsquigarrow & (\pi_2(t), p)
\end{array}$$

On peut voir cette réduction comme une décomposition des contraintes en des disjonctions de contraintes plus simples. On regroupe les contraintes atomiques pour obtenir le résultat.

Fait B.4.0.6. *Posons :*

$$\langle t, p, x \rangle = \bigcup_{(t,p) \overset{*}{\rightsquigarrow} (t',x)} [\![t']\!]$$

Alors :

- *Si $x \in V(p)$:*
$$(t \downarrow p)(x) = \langle t, p, x \rangle$$
- *Si $x \notin V(p)$:*
$$\langle t, p, x \rangle = \varnothing$$
- *La réunion qui définit $\langle t, p, x \rangle$ est finie*

On pose donc :

$$(t \Downarrow p)(x) = \bigvee_{(t,p) \overset{*}{\rightsquigarrow} (t',x)} t'$$

et le théorème est prouvé. Il reste à prouver le fait.

Démonstration. Le second point est immédiat. Le troisième est facile : si t est $\mathcal{X}$-définissable, et $(t,p) \overset{*}{\rightsquigarrow} (t',p')$, alors t' est $\mathcal{X}$-définissable et p' est une sous-expression de p ; il n'y a qu'un nombre fini de tels couples (t',p').

Il reste à prouver le premier point. On commence par voir que :

$$\begin{array}{lcll}
\langle t, x, x \rangle & = & t & \\
\langle t, p_1 \wedge p_2, x \rangle & = & \langle t, p_i, x \rangle & \text{si } x \in V(p_i) \\
\langle t, p_1|p_2, x \rangle & = & \langle t \wedge \wr p_1 \int, p_1, x \rangle \cup \langle t \wedge \neg \wr p_1 \int, p_2, x \rangle & \\
\langle t, (p,_), x \rangle & = & \langle \pi_1(t), p, x \rangle & \\
\langle t, (_,p), x \rangle & = & \langle \pi_2(t), p, x \rangle &
\end{array}$$

Pour établir $(t \downarrow p)(x) \subseteq \langle t, p, x \rangle$, on prouve par induction sur (e,p) (avec l'ordre lexicographique), et simultanément pour tous les types t que :

$$e \in [\![t]\!] \Rightarrow (e/p)(x) \in \langle t, p, x \rangle$$

On procède par disjonction de cas suivant la forme de p (on a $x \in V(p)$, ce qui limite le nombre de cas) :

- Cas $p = x$: on a $(e/x)(x) = e$ et $\langle t, x, x\rangle = [\![t]\!]$.
- Cas $p = p_1 \wedge p_2$: on a $(e/p)(x) = (e/p_i)(x)$ si $x \in V(p_i)$ et $\langle t, p, x\rangle = \langle t, p_i, x\rangle$ et on utilise l'hypothèse d'induction avec $p_i < p$.
- Cas $p = p_1|p_2$.
  - si $e \in \wr p_1 \int$, alors $e \in [\![t \wedge \wr p_1 \int]\!]$; on a $(e/p)(x) = (e/p_1)(x)$ et $\langle t \wedge \wr p_1 \int, p_1, x\rangle \subseteq \langle t, p, x\rangle$, ce qui permet de conclure en utilisant l'hypothèse d'induction avec $p_1 < p$.
  - si $e \notin \wr p_1 \int$, alors $e \in [\![t \wedge \neg \wr p_1 \int]\!]$; on a $(e/p)(x) = (e/p_2)(x)$ et $\langle t \wedge \neg \wr p_1 \int, p_2, x\rangle \subseteq \langle t, p, x\rangle$, ce qui permet de conclure en utilisant l'hypothèse d'induction avec $p_2 < p$.
- Cas $p = (p', \_)$: comme $e \in [\![t]\!] \subseteq [\![\wr p \int]\!] = [\![\wr p' \int]\!] \times \mathcal{D}$, on sait que $e = (e_1, e_2)$ avec $e_1 \in \pi_1(t)$. Alors $(e/p)(x) = (e_1/p')(x)$ et $\langle t, p, x\rangle = \langle \pi_1(t), p', x\rangle$; il ne reste plus qu'à utiliser l'hypothèse d'induction avec $e_1 < e$.
- Cas $p = (\_, p')$: comme le précédent.

Pour établir $\langle t, p, x\rangle \subseteq (t \downarrow p)(x)$, on commence par constater que si $(t, p) \rightsquigarrow (t', p')$, et si $x \in V(p')$, alors $(t' \downarrow p')(x) \subseteq (t \downarrow p)(x)$. On en déduit que si $(t, p) \overset{*}{\rightsquigarrow} (t', x)$, alors $t' = (t' \downarrow x)(x) \subseteq (t \downarrow p)(x)$.

Cela termine la preuve du fait. □

On dispose d'un algorithme effectif pour calculer $(t \Downarrow p)(x)$: il suffit de parcourir le graphe défini par $\rightsquigarrow$ à partir de (t, p), et de collecter t' lorsque l'on atteint un (t', x).

Remarque 42. *Pour typer le filtrage, on a besoin de tous les $(t \Downarrow p)(x)$ pour $x \in V(p)$. Le graphe à parcourir est le même pour tous les x, et donc un seul parcours suffit.*

B.5 Propriétés syntaxiques du système

On prouve ici les propriétés syntaxiques du système de types. Les preuves étant assez classiques, toujours par induction sur la structure des dérivations, on ne donne que les arguments importants. Notons l'importance de la croissance des opérateurs sur les types.

Lemme B.5.0.7 (Renforcement des hypothèses). *Soient Γ_1, Γ_2 deux environnements de typage de même domaine et tels que pour tout x, $\Gamma_2(x) \leq \Gamma_1(x)$. Alors tout jugement dérivable avec Γ_1 est dérivable avec Γ_2.*

Démonstration. La preuve se fait par induction sur la dérivation des jugements ; la seule règle à considérer est celle qui utilise l'environnement, autrement dit $\Gamma \vdash x : \Gamma(x)$; on ajoute simplement une application de la règle de subsomption. □

Théorème B.5.0.8 (Élimination de la subsomption). *Si $\Gamma \vdash \mathtt{e} : t$, alors il existe $t' \leq t$ tel que $\Gamma \Vdash \mathtt{e} : t'$.*

Démonstration. On procède par récurrence sur la structure de la dérivation de $\Gamma \vdash \mathtt{e} : t$. On regarde la dernière règle utilisée et l'on applique l'hypothèse d'induction aux jugements qui apparaissent dans les prémisses. Pour la règle de subsomption, le résultat est immédiat en utilisant la transitivité de $\leq$. Pour les autres règles, il faut essentiellement montrer que l'on peut « pousser » la subsomption dans la conclusion. Traitons par exemple le cas de la règle (op) :

$$\frac{\Gamma \vdash \mathtt{e} : t \leq t_{\mathtt{op}}}{\Gamma \vdash \mathtt{op}(\mathtt{e}) : \mathtt{op}(t)}\ (op)$$

L'hypothèse d'induction donne t' tel que : $\Gamma \Vdash \mathtt{e} : t' \leq t \leq t_{\mathtt{op}}$ On peut appliquer la règle (op) pour obtenir : $\Gamma \Vdash \mathtt{op}(\mathtt{e}) : \mathtt{op}(t')$; et la croissance de $\mathtt{op}(\_)$ donne $\mathtt{op}(t') \leq \mathtt{op}(t)$.

Pour les règles $(couple)$ et $(match)$, on procède de même, en utilisant la croissance de $\_ \times \_$ en ses deux arguments et de $(\_ \Downarrow p)(x)$ (on utilise également le théorème de renforcement des hypothèses).

Pour la règle $(abstr)$, c'est plus facile parce que la conclusion n'est pas changée. Pour les règles $(const)$,(var), il n'y a rien à faire. $\square$

Théorème B.5.0.9 (Conjonction). *La règle suivante est admissible dans le système :*

$$\frac{\Gamma \vdash \mathtt{e} : t_1 \quad \Gamma \vdash \mathtt{e} : t_2}{\Gamma \vdash \mathtt{e} : t_1 \wedge t_2}$$

Démonstration. Comme l'opérateur $\wedge$ est croissant en ses deux arguments, il suffit de prouver que la règle suivante est admissible :

$$\frac{\Gamma \Vdash \mathtt{e} : t_1 \quad \Gamma \Vdash \mathtt{e} : t_2}{\Gamma \vdash \mathtt{e} : t_1 \wedge t_2}$$

La structure de la dérivation d'un jugement $\Gamma \Vdash \mathtt{e} : t$ est alors entièrement déterminée par la forme de $\mathtt{e}$. On procède par induction (sur $\mathtt{e}$ ou sur la structure des dérivation, c'est pareil), en considérant deux dérivations $\Gamma \Vdash \mathtt{e} : t_1$ et $\Gamma \Vdash \mathtt{e} : t_2$. La dernière règle utilisée est la même pour ces deux dérivations. Pour les règles $(const)$,(var), il n'y a rien à faire, car le type obtenu ne dépend pas de l'instance de la règle. Pour les règles (op), $(couple)$ et $(match)$, on utilise encore une fois la croissance des opérateurs sur les types, pour montrer par exemple que $(t_1 \wedge t'_1) \times (t_2 \wedge t'_2) \leq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$ (en fait c'est une égalité ici).

La seule source de multiplicité des typages pour une même expression est en fait la règle $(abstr)$ qui permet de rajouter des négations de types flèche. Deux instances de cette règle pour la même expression peuvent être fusionnées de manière à obtenir comme type l'intersection des types obtenus par chacune d'elles. Si $t' = \bigwedge_{j \in J_1} \neg(t'_j \rightarrow s'_j)$ et $t'' = \bigwedge_{j \in J_2} \neg(t'_j \rightarrow s'_j)$, on peut prendre $t''' = \bigwedge_{j \in J_1 \cup J_2} \neg(t'_j \rightarrow s'_j)$. Alors : $(t \wedge t') \wedge (t \wedge t'') \simeq (t \wedge t''')$. $\square$

Théorème B.5.0.10 (Substitution). *La règle suivante est admissible dans le système :*

$$\frac{\Gamma \vdash \mathtt{e}_1 : s \quad \Gamma + (x : s) \vdash \mathtt{e}_2 : t}{\Gamma \vdash \mathtt{e}_2[\mathtt{e}_1/x] : t}$$

Démonstration. La preuve se fait bien entendu par induction sur la structure de la dérivation de $\Gamma + (x : s) \vdash \mathtt{e}_2 : t$. On peut aussi la visualiser globalement, en « collant » à la place de chaque instance de la règle (var) pour la variable x dans cette dérivation une copie de la dérivation $\Gamma \vdash \mathtt{e}_1 : s$. $\square$

B.6 Le prémodèle des valeurs est compatible avec le système de types

Nous allons établir ici le théorème suivant :

Théorème B.6.0.11. *Soit t un type. Alors :*

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

Commençons par une inclusion :

Lemme B.6.0.12. *Soit t un type. Alors :*

$$v \in \mathbb{V}[\![t]\!] \Rightarrow \vdash v : t$$

Démonstration. On montre la propriété par induction sur v. Si $v \in \mathbb{V}[\![t]\!]$ et si $v \in \mathcal{V}_u$, alors $\exists (P, N) \in t_u.\ (\forall a \in P.\ v \in \mathbb{V}[\![a]\!]) \wedge (\forall a \in N.\ v \notin \mathbb{V}[\![a]\!])$.

Considérons le cas $u = \textbf{basic}, v = \mathsf{c}$. Alors $\forall b \in P.\ t_{\mathsf{c}} \leq b$ et $\forall b \in P.\ t_{\mathsf{c}} \not\leq b$. Les hypothèses sur t_{c} montrent que $t_{\mathsf{c}} \leq \neg b$ pour $b \in P$, et donc : $t_{\mathsf{c}} \leq \bigwedge_{b \in P} b \wedge \bigwedge_{b \in P} \neg b$, ce qui prouve $\vdash \mathsf{c} : \bigwedge_{b \in P} b \wedge \bigwedge_{b \in P} \neg b \leq t$.

Considérons le cas $u = \textbf{arrow}, v = \mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e}$. Posons $t_v = (t_1 \to s_1) \wedge \ldots \wedge (t_n \to s_n)$. Alors, en utilisant la définition de $\mathbb{V}[\![\_]\!]_{\mathbf{arrow}}$: $\forall (t \to s) \in P.\ t_v \leq t \to s$ et $\forall (t \to s) \in P.\ t_v \not\leq t \to s$. En instanciant la règle de typage de l'abstraction v avec comme types négatifs les $(t \to s) \in N$, on obtient : $\vdash v : t_v \wedge \bigwedge_{(t \to s) \in N} \neg (t \to s) \leq t$.

Considérons enfin le cas $u = \textbf{product}$. On a : $(v_1, v_2) \in (\bigcap_{(t_1 \times t_2) \in P} \mathbb{V}[\![t_1]\!] \times \mathbb{V}[\![t_2]\!]) \backslash (\bigcup_{(t_1 \times t_2) \in N} \mathbb{V}[\![t_1]\!] \times \mathbb{V}[\![t_2]\!])$. Le fait B.1.0.1 donne un $N' \subseteq N$ tel que $v_1 \in (\bigcap_{(t_1 \times t_2) \in P} \mathbb{V}[\![t_1]\!]) \backslash (\bigcup_{(t_1 \times t_2) \in N'} \mathbb{V}[\![t_1]\!]) = \mathbb{V}[\![s_1]\!]$ où $s_1 = (\bigwedge_{(t_1 \times t_2) \in P} t_1) \wedge \neg (\bigvee_{(t_1 \times t_2) \in N'} t_1)$ et $v_2 \in \mathbb{V}[\![s_2]\!]$ où $s_2 = (\bigwedge_{(t_1 \times t_2) \in P} t_2) \backslash (\bigvee_{(t_1 \times t_2) \in N \backslash N'} t_2)$. L'hypothèse d'induction donne alors : $\vdash v_1 : s_1$ et $\vdash v_2 : s_2$, et donc avec la règle de typage pour les couples : $\vdash (v_1, v_2) : s_1 \times s_2$. Enfin, le fait B.1.0.1 donne : $s_1 \times s_2 \leq (\bigwedge_{(t_1 \times t_2) \in P} t_1 \times t_2) \backslash (\bigvee_{(t_1 \times t_2) \in N} t_1 \times t_2) \leq t$. □

Corollaire B.6.0.13.

- $[\![t]\!] = \varnothing \Rightarrow \mathbb{V}[\![t]\!] = \varnothing$
- $t \leq s \Rightarrow \mathbb{V}[\![t]\!] \subseteq \mathbb{V}[\![s]\!]$

Démonstration. Prouvons le premier point en raisonnant par contraposition. Si l'on peut trouver $v \in \mathbb{V}[\![t]\!]$, alors le lemme précédent donne $\vdash v : t$ et l'on a déjà prouvé que cela implique $t \not\simeq 0$.

Le second point est conséquence du premier. □

Nous pouvons alors établir la réciproque du lemme :

Lemme B.6.0.14. *Si $\vdash v : t$, alors $v \in \mathbb{V}[\![t]\!]$.*

Démonstration. Avec le corollaire, il suffit de montrer que $\Vdash v : t \Rightarrow v \in \mathbb{V}[\![t]\!]$. On procède par induction sur v et on regarde la dernière règle utilisée dans la dérivation de $\Vdash v : t$. Tous les cas sont triviaux. □

B.7 Modèle des valeurs

Théorème B.7.0.15. *Soit t un type. Alors :*

$$\mathbb{V}[\![t]\!] = \varnothing \Rightarrow [\![t]\!] = \varnothing$$

Démonstration. Il s'agit de voir que : $[\![t]\!] \neq \varnothing \Rightarrow \mathbb{V}[\![t]\!] \neq \varnothing$, et l'on peut même se ramener à un univers donné et à une intersection de types atomiques ou de leurs négations dans cet univers.

On prouve par induction sur e (avec l'ordre bien fondé de la structure du modèle d'amorce) que si $e \in \mathcal{D}_u \wedge e \in [\![t]\!]$, alors on peut exhiber une valeur $v \in \mathcal{V}_u \wedge v \in \mathbb{V}[\![t]\!]$. Le cas $u = \mathbf{basic}$ est trivial ($\mathcal{D}_u = \mathcal{V}_u$ et l'interprétation des types de base est fixée). Le cas $u = \mathbf{product}$ utilise deux fois le fait $B.1.0.1$ et l'hypothèse d'induction, avec la même démarche que dans un lemme précédent. Enfin, traitons le cas $u = \mathbf{arrow}$. Supposons : $\bigwedge_{1 \leq i \leq n} t_i \to s_i \not\leq \bigvee_{1 \leq j \leq m} t'_j \to s'_j$. Définissons la valeur v par :

$$\mu f : (t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.\mathsf{e}$$

où e est une expression arbitraire qui rend v bien typée (il y en a !). On peut appliquer la règle de typage pour les abstractions en prenant comme types négatifs les $t'_j \to s'_j$ (la condition dans les prémisses sur ces types est vérifiée grâce à la propriété de disjonction forte), et on obtient ce que l'on veut. Ceci termine la preuve du théorème. □

B.8 Fidélité de l'application

Théorème B.8.0.16. *La notion d'application dans le modèle des valeurs est fidèle, c'est-à-dire :*

$$\mathbb{V}[\![t_{\mathbf{arrow}}.t]\!] = t_{\mathbf{arrow}} \bullet t$$

Démonstration. Le résultat de *subject reduction* montre immédiatement que :

$$t_{\mathbf{arrow}} \bullet t \subseteq \mathbb{V}[\![t_{\mathbf{arrow}}.t]\!]$$

Pour prouver l'inclusion réciproque, on commence par se ramener au cas où $t_{\mathbf{arrow}} = (\bigwedge t_i \to s_i) \wedge \neg(\bigvee t'_j \to s'_j) \not\simeq 0$. En effet, les deux membres de l'égalité à prouver commutent avec la réunion sur $t_{\mathbf{arrow}}$ (et sur t aussi d'ailleurs) et sont vides lorsque $t_{\mathbf{arrow}} = 0$.

Prenons donc $t_{\mathbf{arrow}} = (\bigwedge_I t_i \to s_i) \wedge \neg(\bigvee_J t'_j \to s'_j) \not\simeq 0$. En regardant la formule donnant l'opérateur d'application, on voit qu'il faut prouver : si $I' \subseteq I$ est tel que $t \not\leq \bigvee_{i \notin I} t_i$, alors $\bigcap_{i \in I'} [\![s_i]\!] \subseteq t_{\mathbf{arrow}} \bullet t$.

Prenons donc une valeur v de type $\bigwedge_{i \in I'} s_i$; posons $t' = t \wedge \neg \bigvee_{i \notin I'} t_i$ et considérons l'expression :

$$\mathsf{e} = \mu f : (t_1 \to s_1; \ldots; t_n \to s_n; t' \to 1).\lambda x.\text{match } x \text{ with } t' \to v \mid 1 \to \mathsf{e}_0$$

où $\vdash \mathsf{e}_0 : 0$ (il existe une telle expression) On voit facilement que $\vdash \mathsf{e} : t_{\mathbf{arrow}}$. Enfin, comme $t' \not\simeq 0$, on peut trouver une valeur v_0 de type t'.

On constate que $\mathsf{e}v_0 \longrightarrow v$ et donc $v \in t_{\mathbf{arrow}} \bullet t$, ce qu'il fallait prouver.

□

B.9 Algorithme de sous-typage

Lemme B.9.0.17. *Prenons un modèle quelconque. Posons $R = \{t \in \mathbf{T} \mid [\![t]\!] = \varnothing\}$. Il s'agit d'une simulation.*

Démonstration. Donnons une idée de la preuve. Prenons un type t tel que $[\![t]\!] = \varnothing$, donc tel que $[\![t_u]\!] = \varnothing$ et considérons un couple $(P, N) \in t_u$. L'étude de la relation de sous-typage dans le cas général des prémodèles (pour les types de base et les types produit) et la condition pour les modèles (pour les types flèche) montre que les types qui doivent se trouver dans R pour satisfaire la condition de simulation sont effectivement vides. $\square$

Lemme B.9.0.18. *Soit R une simulation. Si $t \in R$, alors $[\![t]\!] = \varnothing$ dans le modèle réduit.*

Démonstration. On va prouver par induction sur le terme e et disjonction de cas suivant sa forme que $e \in [\![t^0]\!] \wedge t^0 \in R$ donne une contradiction. Soit u de sorte que $e \in \mathcal{D}_u$. Donc $e \in [\![t^0_u]\!]$ et on peut choisir $(P, N) \in t^0_u$ tel que $e \in [\![a]\!]$ pour tout $a \in P$ et $e \notin [\![a]\!]$ pour tout $a \in N$. La condition de simulation pour ce couple (P, N) va donner la contradiction. On obtient que d'autres types sont dans R. Par induction, il suffit donc d'exhiber des termes plus petits qui sont dans ces types.

Si $u =$ **basic**, la contradiction est immédiate. Sinon, il faut utiliser les faits B.1.0.1, B.1.0.2. Regardons par le cas du produit : $e = (e_1, e_2)$. Le fait B.1.0.1 donne un $N' \subseteq N$ tel que $e_1 \in \bigcap_{(t \times s) \in P} [\![t]\!] \setminus \bigcup_{(t' \times s') \in N'} [\![t']\!]$ et de même pour e_2. Posons $t_1 = \bigwedge_{(t \times s) \in P} t \wedge \bigwedge_{(t' \times s') \in N'} \neg t'$ et $t_2 = \bigwedge_{(t \times s) \in P} s \wedge \bigwedge_{(t' \times s') \in N'} \neg s'$. La condition de simulation pour ce N' dit que $t_1 \in R$ ou $t_2 \in R$. Mais on a : $e_1 \in [\![t_1]\!]$, $e_2 \in [\![t_2]\!]$ et par induction, il n'est donc pas possible d'avoir $t_1 \in R$, ni $t_2 \in R$. Le cas des flèches se traite de même. $\square$

Bibliographie

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4) :575–631, September 1993.

[ACP+92] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, and D. Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, 1992.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2) :237–268, April 1991.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 93.

[BDCd95] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types : syntax and semantics. *Information and Computation*, 119 :202–230, 1995.

[Cas95] G. Castagna. Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping. *Formal Aspects of Computing : International Journal of Formal Methods*, 8(3) :247–293, 1995.

[CGL95] G. Castagna, G. Ghelli, , and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1) :115–135, February 1995.

[Cou97] P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.

[Dam94a] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 687–706. Springer-Verlag, 1994.

[Dam94b] Flemming M. Damm. Subtyping with union types, intersection types and recursive types II. Technical Report 2259, INRIA, may 1994.

[Dam94c] Flemming M. Damm. Type inference with set theoretic type operators. Technical Report 838, IRISA, july 1994.

[DOM] Document Object Model (DOM) ; `http://www.w3.org/DOM/`.

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[HP00] Haruo Hosoya and Benjamin C. Pierce. Xduce : A typed xml processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[Ler01] Xavier Leroy. The Objective Caml system release 3.02 ; Documentation and user's manual, 2001.

[LM91] Xavier Leroy and Michael Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523, pages 406–426. Springer-Verlag, Berlin, Heidelberg, New York, 1991.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Symposium on Principles of Database Systems*, pages 11–22, 2000.

[Pie89] Benjamin C. Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-169, 1989.

[Pie90] Benjamin C. Pierce. Intersection and union types. Technical report, Carnegie Mellon University, 1990.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1) :131–165, July 1994.

[Rém93a] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[Rém93b] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[SCH] XML Schema Part 1 : Structures ; `http://www.w3.org/TR/xmlschema-1`.

[XML] Extensible Markup Language Recommendation (XML) 1.0 ; `http://www.w3.org/TR/REC-xml`.

[XPA] XML Path Language (XPath) ; `http://www.w3.org/TR/xpath`.

[XSL] XSL Transformations (XSLT) ; `http://www.w3.org/TR/xslt`.

Index des notations