Université Paris Cité

École doctorale 386

INRIA Saclay - Équipe PARTOUT

Inférence par contraintes pour les GADTs

Par Olivier MARTINOT

Thèse de doctorat d'Informatique

Dirigée par Gabriel SCHERER

Et François POTTIER

Sommaire

Chapitre 1

Introduction

Afin de décrire les opérations que l'on souhaite faire effectuer à une machine aussi complexe qu'un ordinateur, il est fondamental de faire appel à des *langages de programmation*.

Parmi ces langages certains peuvent servir à parler d'opérations proches de ce qui se passe effectivement "sous le capot", telles que la gestion de la mémoire physique ou les opérations bit-à-bit reposant sur les portes logiques des circuits imprimés d'un CPU. On parle alors de langages *bas niveau*. D'autres langages sont conçus avec plus de recul par rapport à la représentation des données et la façon dont la machine opère sur ces données. En s'abstrayant de certains détails du fonctionnement interne d'un ordinateur, on peut s'épargner énormément de temps, car on n'a alors plus besoin d'écrire à la main la suite d'instructions à effectuer pour chaque calcul. On parle dans ce cas de langages de *haut niveau*.

Il n'existe pas un bon niveau d'abstraction dans l'absolu, mais plutôt tout un dégradé entre des langages de plus ou moins haut niveau.

Quoi qu'il en soit, les langages de programmation donnent une façon de décrire et d'agencer des *expressions*, elles-mêmes récursivement composées de plus petites expressions. De même que dans un langage naturel, une phrase regroupe un ensemble de mots en les liant les uns aux autres et en leur donnant un sens, un langage de programmation lie des expressions ensembles et leur donne un sens. On parle de *sémantique*.

Ainsi dans la phrase `"Un chien mouillé n'en sèche pas un autre."`, composée de différents éléments (nom, verbe, etc.), chaque mot et certains groupes de mots (`"Un chien mouillé"`, `"un autre"`, etc.) ont un sens séparément, mais le sens global est donné par la phrase entière.

On retrouve un principe similaire dans les langages de programmation. Ainsi l'expression `let f = fun n -> n + 1`, qui déclare une fonction `f` comme étant la fonction successeur sur les nombres entiers, contient une sous-expression `fun n -> n + 1`, qui est une fonction anonyme elle-même constituée d'une expression `n + 1`, qui est une addition elle-même constituée de deux expressions `n` et `1`.

On peut changer les mots d'une phrase pour qu'elle reste cohérente (quitte à en changer le sens), par exemple en les réordonnant (`"Un chien n'en sèche pas un autre mouillé."`), en modifiant des mots (`"Un tricératops mouillé n'en sèche pas un autre."`), etc. Mais on peut aussi produire une phrase qui n'a pas de sens (du moins si on s'exclut des libertés poétiques), telle que la phrase `"N'en sèche pas un chien mouillé un autre."`. De la même façon, dans les langages de programmation, toute expression n'a pas vocation à s'évaluer en un résultat. Par exemple l'application de la fonction `f` précédemment définie à une paire d'éléments ne pourra pas s'évaluer, car `f` attend comme argument un unique élément, qui de plus, doit être un nombre entier.

Pour classifier les expressions et s'assurer de pouvoir leur donner un sens, on attribue des *types* aux constructions des langages de programmation. Certains types peuvent être définis par le langage, comme c'est souvent le cas pour les entiers, les chaînes de caractères, les booléens, etc., mais également être déclarés par l'utilisateur ou encore exprimer des façons d'agencer des types ensemble : fonctions, types de données structurés, ... Prenons l'expression :

```
fun n -> (n - 1, n + 1)
```

Cette fonction prend en argument un entier `n` et renvoie la paire constituée du prédécesseur et du successeur de `n`. Le type de l'argument `n` est `int` (le type des entiers), le type de retour de la

fonction est la paire `int` × `int`, et le type de la fonction est `int` → (`int` × `int`). En donnant un type à cette fonction, on peut désormais s'assurer qu'elle est bien utilisée dans nos programmes. Ainsi, on s'aperçoit facilement que `f (0,0)` ou `(f 0) + 1` sont des expressions dont les types des sous-expressions sont incompatibles : on dit que ces expressions ne typent pas.

Il est important de rejeter de tels programmes, avant même de les exécuter, car on sait à l'avance que leur comportement n'est pas autorisé par la sémantique du langage. Analyser les types d'un programme et s'assurer que les types de ses expressions sont cohérents entre eux est le rôle d'un *typeur*. Le typeur peut effectuer ses vérifications en amont de l'exécution du programme, on parle alors de typage *statique*.

Le travail du typeur peut être simplifié par le programmeur si celui-ci annote certaines expressions de son code avec des types. Le typeur sait dans ce cas quel type il est censé donner aux expressions et n'a plus qu'à vérifier si les types explicités par le programmeur sont compatibles avec les expressions qu'ils annotent. Ainsi dans le code suivant

```
fun f (n : int) -> n + 1
```

le programmeur a annoté l'argument `n` de la fonction `f` avec le type des entiers naturels `int`. Le typeur, aidé par cette annotation, peut en déduire facilement que le type de retour de la fonction est également `int`. Il vérifie au passage, en typant l'addition `n + 1`, que le type `int` est bien compatible avec `n`. Annoter ses programmes permet ainsi de simplifier le typeur, à qui on fournit alors des types directement dans le corps du programme et qui n'a plus alors qu'à vérifier s'ils sont corrects. En outre, les types fournissent des informations utiles à des personnes qui voudraient relire le code. Il s'agit donc également d'une façon rudimentaire de documenter son code.

Une autre approche consiste à garder implicites les types dans le code, et à laisser le typeur les déduire lui-même. On parle alors d'*inférence des types*. Bien que cela complique l'écriture du typeur, cette approche a le mérite de simplifier le code et rendre son écriture plus rapide (du moins lorsque celui-ci n'introduit pas de bug de typage difficile à déceler à première vue !).

Un langage de programmation qui met un fort accent sur l'inférence de types est *ML*, pour *Meta Language* (voir par exemple Gordon, Milner, Morris, Newey, and Wadsworth (1978) pour une description du langage). Ce langage, apparu dans les années 1970, est à la base d'une famille de langages de programmations qui s'en inspirent. Parmi ceux-ci, on trouve OCaml (Leroy, Vouillon, Doligez, Rémy, Suárez, et al. (1996)), pour Objective Caml, un langage lui-même issu de Caml (Weis and Leroy (1993)).

Dans ML, OCaml et d'autres langages proches, les types sont implicites, et donc inférés par le typeur, bien que le programmeur a la possibilité d'annoter son code. Le système de type de ces langages permet du *polymorphisme paramétrique*, c'est-à-dire que le type d'une expression peut être paramétré par des *variables de type* en lieu et place de types habituels. Ces variables de types peuvent être instanciées par différents types à différents endroits, un peu à la manière d'une fonction dont les paramètres sont instanciés par différentes valeurs à différents endroits. Le polymorphisme s'avère fondamental pour écrire du code réutilisable. Prenons, par exemple, la fonction `pair` qui construit une paire à partir d'un élément :

```
let pair x = (x, x)
```

Cette fonction s'écrit de la même manière quelque soit le type de l'argument. On imagine bien à quel point il serait fastidieux et répétitif d'avoir à en écrire une version différente pour chaque type différent du paramètre (`int`, `bool`, `int` × `int`, etc.). Avec le polymorphisme, on ne donne pas à `pair` un type unique, mais plutôt un schéma de types : quelque soit le type α passé en paramètre, la fonction renvoie une paire de type $\alpha \times \alpha$. On note ce schéma $\forall\alpha.\ \alpha \to (\alpha \times \alpha)$. Dans le reste du code, on pourra appeler `pair` avec n'importe quel type t à la place de α et le typeur pourra vérifier que le type de retour de la fonction sera $t \times t$.

Afin de rattacher le typage d'un langage fonctionnel avec du polymorphisme, comme OCaml, à un cadre théorique plus large, on peut s'appuyer sur *Système F* (basé sur les travaux de Girard (1972) et Reynolds (1974)). Dans Système F, les variables de types sont introduites explicitement. Ainsi, la fonction identité décrite plus haut, s'écrit en Système F de la façon suivante :

$$\Lambda\alpha.\lambda(x : \alpha).x$$

La construction $\Lambda\alpha.u$ introduit une nouvelle variable de type α dans le reste de l'expression u. On remarque également qu'on a annoté le type de l'argument x de la fonction. En annotant ainsi les

expressions, il devient possible de déterminer facilement si elles sont bien typées ou non.

Pour passer d'une expression OCaml sans annotation à son pendant annoté (sous la forme d'une expression Système F, par exemple), dans laquelle les variables de types et les types des arguments de fonctions sont explicités, il nous faut une façon de faire l'inférence des types. Il existe pour cela plusieurs approches.

Une façon classique est décrite à travers l'algorithme $\mathscr{W}$, dû à Milner (voir Milner (1978)). Cet algorithme fait appel à la notion d'*unification*, que l'on retrouve dans d'autres domaines de l'informatique et de la logique. À partir d'un ensemble d'équations avec des variables, il s'agit de trouver une substitution pour ces variables qui permet d'unifier, c'est-à-dire de rendre égale, dans chaque équation, les expressions des deux côtés.

Reprenons l'exemple de la fonction identité `let id x = x` et essayons de générer des équations sur son type, appelons-le τ. Comme il s'agit d'une fonction on sait que

$$\tau \sim \alpha \to \beta$$

c'est-à-dire que τ doit pouvoir s'unifier avec un type flèche, dont le type de l'argument est α et le type de retour est β, pour un certain α et un certain β. Comme le corps de la fonction est simplement l'expression x, on peut en déduire que le type de retour de la fonction est égal au type de son argument. Cela se traduit par l'unification

$$\beta \sim \alpha$$

En unifiant les deux équations obtenues précédemment, on obtient une nouvelle unification sur le type de la fonction :

$$\tau \sim \alpha \to \alpha$$

En observant que l'on ne peut pas déduire de type plus précis, on trouve que τ est bien un type polymorphe $\forall\alpha.\alpha \to \alpha$.

Une autre approche, développée notamment dans Wand (1987), consiste à considérer l'inférence de types comme une résolution de contraintes d'unification entre différents types. Ces contraintes s'obtiennent directement à partir de la forme des expressions. Ainsi, par exemple, une fonction génère une contrainte qui s'assure que son type est un type flèche. Le typeur va générer un ensemble de contraintes qu'il cherchera à résoudre. L'idée étant que s'il parvient à les résoudre, le programme est censé être typable et la résolution des contraintes va lui permettre de trouver des types pour annoter le programme, s'il échoue à les résoudre le programme n'est pas censé être typable. Désigner un typeur qui satisfait ces propriétés n'est pas évident, et il peut être utile de décrire formellement son fonctionnement : définir un ensemble de contraintes approprié, décrire la génération et la résolution de contraintes, ... C'est ce travail qui est présenté dans Pottier and Rémy (2005).

Durant ma thèse, nous avons essayé de voir comment inférer les types d'un sous-ensemble d'OCaml avec une approche par résolution de contraintes. Nous nous sommes particulièrement intéressés à la façon d'inférer les GADTs, une fonctionnalité avancée d'OCaml, qui soulève des problèmes de typage intéressants.

1.1 Aperçu du travail de thèse

J'ai travaillé sur l'inférence de types par résolution de contraintes pour le langage OCaml, en étendant Inferno, une bibliothèque développée par François Pottier. Celle-ci fournit à l'utilisateur une interface pour générer des contraintes et les élaborer vers des termes annotés une fois résolues, ainsi qu'un solveur. Le but principal que l'on s'est fixé était de faire prendre en charge à Inferno une partie plus importante du langage OCaml. La bibliothèque est construite en deux parties : une partie "solveur", qui fournit à l'utilisateur des combinateurs et un solveur pour écrire un typeur, et une partie "client", qui est un exemple de typeur pour un langage fonctionnel jouet. Nous avons donc étendu la partie "client", afin de pouvoir typer des programmes plus riches que ceux qui pouvaient être exprimés avec le langage de base. Cela nous a également demandé de modifier de façon conséquente la partie "solveur", qui ne fournissait pas à l'utilisateur les outils suffisants pour traiter des fonctionnalités de typage avancées. Nous nous sommes concentrés durant ma thèse sur

l'ajout des GADTs dans Inferno. Pour cela, nous avons utilisé les règles de typage de Jacques Garrigue et Didier Rémy qui s'appuient sur la notion d'ambivalence..

Nous avons réussi à implémenter l'inférence des GADTs dans Inferno. L'efficacité était une des raisons de notre approche par contraintes, et nous donnons dans le manuscrit quelques éléments qui rendent notre implémentation efficace.

De plus, nous avons formalisé ce travail. Cela nous a demandé de réfléchir à un langage de contrainte adapté, ainsi qu'à une sémantique appropriée. Cette sémantique était notamment à comparer à celle décrite dans les travaux d'Alistair O'Brien.

1.2 Dans la suite

Dans les premiers chapitres de ce manuscrit, je rappelle l'état de l'art et définis les différentes notions à partir desquelles j'ai travaillé.

Le chapitre 2 revient sur l'inférence de types, et notamment l'inférence par résolution de contraintes. J'y définis notamment un ensemble de contraintes et leurs sémantiques, et donne une façon de générer des contraintes à partir d'un langage noyau ML. J'en profite également pour définir quelques notions qui seront utiles pour raisonner sur la sémantique des contraintes.

Le chapitre 3 décrit un système de réécriture défini dans Pottier and Rémy (2005) pour résoudre les contraintes définies précédemment. Il est important d'expliciter le fonctionnement de ce solveur, car une partie de mon travail a précisément consister à augmenter ce solveur avec de nouvelles règles de réécriture.

Dans le chapitre 4, je commence par rappeler certains aspects des GADTs, avant de parler plus en détail de la notion d'ambiguïté. J'expose une certaine approche pour détecter l'ambiguïté, qui a guidé notre travail sur le solveur.

Le chapitre 5 introduit une nouvelle contrainte : la contrainte d'hypothèse d'égalité. J'y explique deux choix possible de sémantique, et laquelle des deux nous a paru la plus pertinente.

Dans le chapitre 6, j'explique comment étendre le solveur pour prendre en charge cette nouvelle contrainte. Il s'agit de définir de nouvelles règles de réécriture et de comprendre comment le solveur peut détecter des erreurs de typage causées par une mauvaise utilisation des GADTs.

Le chapitre 7 développe notre conception des variables rigides locales. Traitées naïvement, les variables rigides créent des problèmes de partage lors de la résolution des contraintes. Pour y remédier, nous décrivons une notion de structures abstraites, qui a émergé de l'implémentation du solveur. Il a fallu introduire, en plus de cette notion de structure abstraite, une nouvelle contrainte `letr` que nous détaillons.

Quelques éléments notables de l'implémentation du solveur sont décrit dans le chapitre 8.

Le chapitre 9 positionne notre travail en le rapprochant des travaux liés.

Enfin, le chapitre 10 décrit certains aspects du travail d'implémentation qu'il n'était pas nécessaire de présenter plus tôt dans le manuscrit, mais qui peuvent intéresser la lectrice ou le lecteur curieux.

Chapitre 2

Inférence de types

2.1 Inférence de types à la Hindley-Milner

Quelques généralités

L'inférence de types regroupe les méthodes qui permettent de déterminer les types d'un programme dans lequel ceux-ci ne sont pas, ou seulement partiellement, spécifiés. Grâce à ces méthodes, on peut donc établir une discipline de types sans avoir à annoter entièrement un programme par ses types. Pour déduire les types d'un programme, on s'intéresse à la façon dont sont utilisées les expressions. En parcourant le programme, on apprend de plus en plus d'informations jusqu'à être capable de définir un type précis pour chaque élément.

Exemple 2.1.1.

```
let apply f x =
  (f x) + 1
```

`apply` est une fonction qui attend deux arguments : elle est donc de type

$$t_1 \to t_2 \to t_3$$

On sait également que son premier argument `f` est appliqué à `x` : il s'agit donc d'une fonction dont le type de l'argument est égal au type de `x`, à savoir t_2. On peut donc raffiner le type de `apply` pour obtenir

$$(t_2 \to t_4) \to t_2 \to t_3$$

On peut voir que l'expression `f x` est additionnée avec `1` : le type de retour de `f` est donc `int`. Cela permet de raffiner à nouveau le type de `apply` :

$$(t_2 \to \texttt{int}) \to t_2 \to t_3$$

Enfin, on sait que le type de retour d'une addition est `int`, donc que le type de `(f x) + 1` est `int`. On en déduit donc que le type de `apply` est

$$(t_2 \to \texttt{int}) \to t_2 \to \texttt{int}$$

pour un certain type t_2.

$\Diamond$

Unification

Inférer des types nécessite de pouvoir travailler temporairement sur des types incomplets (on peut penser au type $t_1 \to t_2 \to t_3$ dans l'exemple précédent). On peut exprimer ces types avec des inconnues $X, Y, \ldots$ à la place des types. Il faudra par la suite être en mesure d'*unifier* deux types ensemble afin de vérifier qu'ils sont bien compatibles et obtenir un type plus complet déduit des deux types unifiés.

Exemple 2.1.2.

```
let apply_bin f x y =
  (f x 0, f true y)
```

On peut déduire de `f x 0` que le type de `f` est de la forme

$$X \to \texttt{int} \to Z$$

et de `f true y` qu'il est de la forme

$$\texttt{bool} \to Y \to W$$

Ces deux types dénotent en fait un seul type : il faut donc les unifier ensemble, en remplaçant, si possible, les variables par des types. On s'aperçoit que X doit être égal à `bool` et Y égal à `int`. Enfin, il faut que les deux types de retours coïncident : Z doit être égal à W. ◊

Nous verrons que ces unifications peuvent s'écrire sous la forme de contraintes.

Polymorphisme et typage du let

Le système de type de Hindley-Milner, décrit dans Milner (1978) et Damas and Milner (1982), est la base du typage des langages de la famille ML, dont OCaml. Ce système permet notamment du polymorphisme, grâce aux expressions let.

Pour inférer une expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, on commence par inférer le type de e_1. On généralise ensuite ce type : on obtient un schéma de type qui est associé à x dans l'environnement de typage. Enfin, on infère le type de e_2 dans ce nouvel environnement, en instanciant le schéma à chaque occurrence de x.

Présentation d'un sous-ensemble d'OCaml

Dans la suite, nous travaillerons à partir d'un petit langage sous-ensemble d'OCaml. Voici la grammaire de ses termes :

$$u ::= x \mid \lambda x.u \mid u\ u \mid \mathsf{let}\ x = u\ \mathsf{in}\ u \mid \forall a.u \mid (u : \tau)$$

La grammaire des types est la suivante :

$$\begin{array}{lll} \tau & ::= & a \mid s\ \bar{\tau} \\ t & ::= & s\ \bar{t} \\ s & ::= & (\to) \mid (\times) \mid Constr \mid \ldots \end{array}$$

Les types peuvent contenir des variables polymorphes explicites, appelées variables rigides et notées a, b, etc. τ dénote les types utilisateurs, tandis que t dénote les types "ground" sans variables rigides. s dénote les structures des types, tels que la structure flèche pour les types de fonctions, ou la structure de paire. Un type peut être représenté comme un arbre dans lequel les noeuds sont des structures. On peut remarquer qu'un type utilisateur est soit une variable rigide, soit un type construit à partir d'une structure s dont les enfants sont un nombre arbitraire de types utilisateurs (notés $\bar{\tau}$).

Annotation avec Système F

Inférer des types de programmes écrit dans ce sous-ensemble d'OCaml consistera à annoter ces programmes avec des types Système F, dont voici la grammaire :

$$\tau_F ::= \alpha \mid s\ \bar{\tau_F} \mid \forall \alpha.\tau_F \mid \mu\alpha.\tau_F$$

On peut noter la possibilité d'exprimer des types récursifs avec la construction $\mu\alpha.\tau_F$. De tels types peuvent apparaître par unification durant le typage d'un programme, même si on ne souhaite pas forcément les autoriser. Cela permet également d'imiter l'option `-rectypes` d'OCaml.

On peut donner des règles de typage pour ce système. On commence par définir un contexte de typage

$$\Gamma ::= x_1 : \tau_F^1, \ldots, x_n : \tau_F^n$$

$$\frac{(x : \tau_F) \in \Gamma}{\Gamma \vdash x : \tau_F} \qquad \frac{\alpha \notin ftv(\Gamma) \qquad \Gamma \vdash t : \tau_F}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\tau_F} \qquad \frac{\Gamma \vdash t : \forall\alpha.\tau'_F}{\Gamma \vdash t\ \tau_F : \tau'_F[\alpha \backslash \tau_F]}$$

$$\frac{\Gamma \vdash t : \tau_F[\alpha \backslash \mu\alpha.\tau_F]}{\Gamma \vdash t : \mu\alpha.\tau_F} \qquad \frac{\Gamma \vdash t : \mu\alpha.\tau_F}{\Gamma \vdash t : \tau_F[\alpha \backslash \mu\alpha.\tau_F]}$$

On peut énoncer quelques règles de typage sur des structures s telles que $(\rightarrow)$ et $(\times)$:

$$\frac{\Gamma, x : \tau_F \vdash t : \tau'_F}{\Gamma \vdash \lambda(x : \tau_F).t : \tau_F \rightarrow \tau'_F} \qquad \frac{\Gamma \vdash t : \tau'_F \rightarrow \tau_F \qquad \Gamma \vdash t' : \tau'_F}{\Gamma \vdash t\ t' : \tau_F} \qquad \frac{\Gamma \vdash t : \tau_F \qquad \Gamma \vdash t' : \tau'_F}{\Gamma \vdash (t, t') : \tau_F \times \tau'_F}$$

2.2 Inférence de types avec contraintes

2.2.1 Principes généraux

Exprimer le problème de l'inférence de types à l'aide de contraintes est un sujet qui a été exploré dés les années 1990, et notamment pour l'inférence à la Hindley-Milner, à travers le modèle $HM(X)$ (Sulzmann, Odersky, and Wehr (1996)).

Dans cette approche, le typeur produit des contraintes à partir du programme à typer, et tente de les résoudre. Une des façons de décrire un tel typeur est de concevoir son fonctionnement en trois phases successives : la génération de contraintes, puis leur résolution et enfin l'élaboration vers un programme annoté. Le typeur se base alors sur un solveur de contraintes, dont la sémantique ne dépend que du langage de contrainte, mais pas des langages source et cible. Pour une contrainte close, le solveur renvoie `SAT`/`UNSAT` selon que la contrainte qui lui est fournie est satisfiable ou non. Pour une contrainte avec des inconnues, le solveur trouve un type valide pour les inconnues. Un des avantages de cette approche est que l'on est en mesure, à partir d'un ensemble relativement réduit de contraintes, d'exprimer un grand nombre de systèmes de typage différents. De plus, un même solveur peut servir à plusieurs typeurs différents et pour typer des langages différents.

Parmi les trois phases de l'inférence de types avec contraintes, seule la génération de contraintes et l'élaboration dépendent des langages source et cible : écrire un typeur consiste donc à décrire ces deux phases. Celles-ci n'ont pas lieu au même moment, mais peuvent en fait être décrites au même endroit dans le code du typeur. C'est ce principe que décrira François Pottier dans ses travaux (Pottier (2014)) et qui se retrouve dans sa bibliothèque Inferno, un framework pour écrire des typeurs avec une approche par résolution de contraintes.

2.2.2 Grammaire des contraintes

En ce qui concerne notre sous-ensemble d'OCaml, nous pouvons décrire un certain nombre de contraintes qui suffisent à inférer les types d'un programme. On dénote par T les types intermédiaires manipulés par le solveur durant l'inférence :

$$T ::= \ X \mid a \mid s\ \bar{X}$$

Ce type contient des variables rigides, des structures, ainsi que des variables d'inférence, ou *variables flexibles*, dénotées $X, Y, ...$ Ces variables représentent des inconnues dont le type est contraint par la forme du programme. Ainsi, si X est assignée au type d'une fonction, elle sera contrainte d'être égale à un type de la forme $Y \rightarrow Z$.

On définit également les schémas de types manipulés par le solveur :

$$\sigma ::= \forall\bar{X}.T$$

Pour exprimer des contraintes sur les types, on définit un langage des contraintes C :

$$C ::= \mathsf{true} \mid \mathsf{false} \mid C \wedge C \mid \exists X.C \mid X \text{ is } T \mid \mathsf{let}\ x = \lambda X.C \ \mathsf{in}\ C \mid x\ X \mid \forall a.C$$

Pour des raisons que nous expliciteront ultérieurement, nous choisissons d'avoir une contrainte X is T, que l'on peut considérer pour le moment comme l'égalité entre une variable d'inférence et un type.

Définition 2.1. On définit $ftv(T)$ (resp. $ftv(C)$) l'ensemble des variables libres (flexibles et rigides) de T (resp. C).

La contrainte let et le polymorphisme

Dans un programme OCaml, la syntaxe `let x = e1 in e2` introduit un identificateur `x` qui s'évalue en `e1` dans toutes ses occurrences dans `e2`. De la même façon, la contrainte let $x = \lambda X.C_1$ in C_2 du langage de contrainte introduit un identificateur x qui correspond à l'abstraction $\lambda X.C_1$ dans toutes les occurrences de x dans C_2. Comme le suggère la notation avec un λ, résoudre une contrainte $(\lambda X.\ C)\ C'$ consiste à résoudre $C'[X \backslash C]$. La sémantique de la contrainte let est équivalente à la sémantique de $\exists X.C_1 \wedge \mathsf{def}\ x : \lambda X.C_1$ in C_2 pour une certaine contrainte def qui définit x dans l'environnement qui évaluera C_2. Pour être satisfiable, il doit donc exister une variable X satisfaisant C_1.

Cette contrainte let permet de représenter des schémas polymorphes dans le langage des contraintes. Ainsi, pour représenter le schéma $\forall\alpha.\alpha \to \alpha$, on peut écrire la contrainte let $x = \lambda X.\exists Y.X$ is $Y \to Y$ in C. Chaque occurrence de x dans C permet une instanciation différente du schéma. L'instanciation est représentée par la contrainte $x\ X$.

Définition 2.2. L'instanciation d'un schéma $\forall\bar{a}.T$ par un type ground t, notée $\forall\bar{a}.T \preceq t$ est définie par $\exists\bar{t'}.T[\bar{t'}/\bar{a}] = t$.

2.3 Génération

La génération de contraintes peut se décrire par une traduction de la forme $[\![t : X]\!]$, où t est un terme du langage source et X est la variable d'inférence qui représente le type que l'on contraint à être celui de t.

Nous générons des contraintes en "petit terme", c'est à dire que les feuilles types formulés dans les contraintes ont comme feuille des variables. La traduction d'un type utilisateur τ présent dans une annotation $(t : \tau)$ est donc une opération récursive. On décrit cette traduction via une fonction de traduction $(\!|X \sim \tau|\!)$ qui contraint X à être de type τ en définissant à la volée autant de variables fraîches que nécessaire.

$$\begin{array}{lcl}
[\![x : X]\!] & ::= & x\ X \\
[\![\lambda x.t : X]\!] & ::= & \exists X_1 X_2.\ X \text{ is } X_1 \to X_2 \wedge \mathsf{let}\ x = \lambda Y.(Y \text{ is } X_1)\ \mathsf{in}\ [\![t : X_2]\!] \\
[\![t\ u : X]\!] & ::= & \exists Y Z.\ Z \text{ is } Y \to X \wedge [\![t : Z]\!] \wedge [\![u : Y]\!] \\
[\![(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) : X]\!] & ::= & \mathsf{let}\ x = \lambda Y.[\![t_1 : Y]\!]\ \mathsf{in}\ [\![t_2 : X]\!] \\
[\![\forall a.t : X]\!] & ::= & \forall a.[\![t : X]\!] \\
[\![(t : \tau) : X]\!] & ::= & [\![t : X]\!] \wedge (\!|X \sim \tau|\!)
\end{array}$$

$$\begin{array}{lcl}
(\!|X \sim a|\!) & ::= & X \text{ is } a \\
(\!|X \sim s(\tau_i)_i|\!) & ::= & \exists (Y_i)_i.X \text{ is } s(Y_i)_i \wedge \bigwedge_i (\!|Y_i \sim \tau_i|\!)
\end{array}$$

Exemple 2.3.1.

On peut définir la fonction identité dans un let et observer la contrainte générée.

$$\begin{aligned}[\![\mathsf{let}\ f = \lambda x.\ x\ \mathsf{in}\ f : W]\!] &= \mathsf{let}\ f = \lambda X.[\![\lambda x.\ x : X]\!] \\ &\quad\ \mathsf{in}\ [\![f : W]\!] \\ &= \mathsf{let}\ f = \lambda X.\exists X_1 X_2.X\ \mathsf{is}\ X_1 \to X_2 \wedge \mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2 \\ &\quad\ \mathsf{in}\ f\ W\end{aligned}$$

On obtient une contrainte de la forme $\mathsf{let}\ f = C_{body}\ \mathsf{in}\ f\ W$. C_{body} correspond à la contrainte générée pour le corps de f et $f\ W$ est l'instanciation de cette contrainte avec la variable d'inférence W.

On peut prendre le même terme, appliquer la fonction f à une valeur (par exemple ()) et observer comment la contrainte générée évolue.

$$\begin{aligned}[\![\mathsf{let}\ f = \lambda x.\ x\ \mathsf{in}\ f\ ()\ : W]\!] &= \mathsf{let}\ f = \lambda X.[\![\lambda x.\ x : X]\!] \\ &\quad\ \mathsf{in}\ [\![f\ () : W]\!] \\ &= \mathsf{let}\ f = \lambda X.\exists X_1 X_2.X\ \mathsf{is}\ X_1 \to X_2 \wedge \mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2 \\ &\quad\ \mathsf{in}\ [\![f\ () : W]\!] \\ &= \mathsf{let}\ f = \lambda X.\exists X_1 X_2.X\ \mathsf{is}\ X_1 \to X_2 \wedge \mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2 \\ &\quad\ \mathsf{in}\ \exists Z Z'.\ Z\ \mathsf{is}\ Z' \to W \wedge f\ Z \wedge Z'\ \mathsf{is}\ \texttt{unit}\end{aligned}$$

◊

Par rapport à la contrainte précédente, on a contraint l'instanciation de f de façon à ce que son argument soit de type `unit` et son type de retour s'unifie à W, le type de toute l'expression.

2.4 Règles de sémantique

La génération de contraintes donne une intuition sur l'usage de chacune des contraintes. On peut aller plus loin en donnant une sémantique aux contraintes. Nous décrirons dans le chapitre 3 un solveur qui correspond à cette sémantique. Nous donnons des règles de sémantique qui font intervenir un jugement de la forme $E; \gamma \models C$, où E est un environnement qui associe un schéma de types aux variables du langage sources, γ est un environnement qui associe un type aux variables d'inférence, et C est une contrainte.

$$\frac{}{E;\gamma \models \mathsf{true}} \qquad \frac{E;\gamma \models C_1 \qquad E;\gamma \models C_2}{E;\gamma \models C_1 \wedge C_2} \qquad \text{EXISTS}\ \frac{\exists t, \quad E;\gamma[X \mapsto t] \models C}{E;\gamma \models \exists X.C} \qquad \text{FORALL}\ \frac{\forall t, \quad E;\gamma[a \mapsto t] \models C}{E;\gamma \models \forall a.C}$$

$$\frac{\gamma(X) = \gamma(T)}{E;\gamma \models X\ \mathsf{is}\ T} \qquad \frac{\forall t, \quad \sigma \preceq t \implies E;\gamma[X \mapsto t] \models C_1 \qquad E[f \mapsto \sigma];\gamma \models C_2}{E;\gamma \models \mathsf{let}\ f = \lambda X.C_1\ \mathsf{in}\ C_2} \qquad \frac{E(f) \preceq \gamma(X)}{E;\gamma \models f\ X}$$

Le quantificateur $\exists$ dans la prémisse de EXISTS ainsi que $\forall$ dans la prémisse de FORALL sont des quantificateurs de la méta-logique.

L'idée est d'avoir des contraintes faciles à exprimer. Il faut ensuite être capable de trouver une correspondance entre la sémantique décrivant les contraintes et le solveur : le solveur renvoie `SAT` pour la contrainte C si et seulement si $\emptyset \models C$.

Définition 2.3. Soit deux contraintes C_1 et C_2. Si, quel que soit E, γ, on a $E, \gamma \models C_1$ implique $E, \gamma \models C_2$ et $E, \gamma \models C_2$ implique $E, \gamma \models C_1$, alors on dit que C_1 et C_2 sont *équivalentes* et on note $C_1 \equiv C_2$.

Exemple 2.4.1.

On a, par exemple :

$$\exists X.\ X\ \mathsf{is}\ \texttt{int} \wedge \exists Y.\ Y\ \mathsf{is}\ \texttt{unit} \quad \equiv \quad \exists XY.\ X\ \mathsf{is}\ \texttt{int} \wedge Y\ \mathsf{is}\ \texttt{unit}$$

◊

Définition 2.4. *C détermine Y* si et seulement si pour tout environnement polymorphe E, et pour toutes assignations γ_1, γ_2 qui coïncident en dehors de Y et tels que $E; \gamma_1 \models C$ et $E; \gamma_2 \models C$, alors γ_1 et γ_2 doivent coïncider sur Y également.

Exemple 2.4.2.

La contrainte $\exists X.\ X \text{ is } Y \wedge X \text{ is } \texttt{bool}$ détermine Y. ◊

Définition 2.5. On dit que Y est *dominée* par X dans une contrainte d'unification U (et on note $Y \prec_U X$) si et seulement si U contient une clause $X = s\ \bar{Z} = \epsilon$ et $Y \in \bar{Z}$.

Exemple 2.4.3.

Prenons $U := X = Y \to Z$. Alors on a $Y \prec_U X$ et $Z \prec_U X$. ◊

Définition 2.6. Une contrainte d'unification U est *cyclique* ssi le graphe de $\prec_U$ est cyclique.

Exemple 2.4.4.

Prenons $U := X = Y \to \texttt{list}(X)$. Alors, comme X apparaît dans $Y \to \texttt{list}(X)$ on a que X est dominée par elle-même. ◊

On peut faire le lien entre la génération de contraintes et les solutions du jugement sémantique :

Conjecture 2.7. Soit t un terme clos, τ un type ground, E et γ des environnement et X une variable d'inférence.

$$E, \gamma \models [\![t : X]\!] \text{ et } \gamma(X) = \tau \text{ si et seulement si } \emptyset, \emptyset \vdash t : \tau$$

Chapitre 3

Solveur

3.1 Système de réécriture

3.1.1 Triplet d'état du solveur

Dans Pottier and Rémy (2005), un état du solveur est donné par un triplet S ; U ; C, où C est la contrainte en cours de résolution, S est une pile qui accumule des contextes dans lesquels résoudre cette contrainte et U est une contrainte d'unification, représentée par des multi-équations quantifiées existentiellement (il s'agit d'une partie de la contrainte déjà résolue). Les auteurs définissent un système de réécriture non déterministe de la forme :

$$S \ ; \ U \ ; \ C \to S' \ ; \ U' \ ; \ C'$$

On définit les multi-équations de variables d'inférence, variables rigides et structures.

$$\epsilon ::= X_1 = \cdots = X_n = a_1 = \cdots = a_m = s_1 \ \bar{Y_1} = \cdots = s_p \ \bar{Y_p}$$

Cette approche avec multi-équations nous permet de coller à l'implémentation qui se base sur une structure d'union-find.

Par soucis de simplification, on note ϵ sous forme d'une multi-équation (avec le symbole $=$), mais pour raisonner à nouveau en terme de contraintes, il s'agira de considérer cette multi-équation comme une conjonction de contraintes de la forme X is T.

Définition 3.1. Une multi-équation est standard, si elle contient au plus une variable rigide ou une structure. Elle est donc de la forme :

$$\hat{\epsilon} ::= X_1 = \cdots = X_n (= a \mid s \ \bar{Y})$$

La grammaire de U forme un sous-ensemble des contraintes :

$$U ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid U \wedge U \mid \exists X.U$$

La grammaire de la pile des contextes S est :

$$S ::= [] \mid S[\exists X.[]] \mid S[[] \wedge C] \mid S[\mathsf{let}\ x = \lambda X.[]\ \mathsf{in}\ C] \mid S[\mathsf{let}\ x = \lambda X.U\ \mathsf{in}\ []] \mid S[\forall a.[]]$$

La configuration S ; U ; C correspond à la contrainte $S[U \wedge C]$, c'est-à-dire la contrainte $U \wedge C$ replacée dans le contexte S. Le triplet initial pour que notre solveur travaille sur une contrainte C est donc [] ; true ; C.

Globalement, la réécriture avance en poussant des bouts de contraintes dans la pile de contextes, pour pouvoir travailler temporairement sur des sous-contraintes plus faciles à résoudre, en faisant des unifications au passage, puis en dépilant les contextes petit à petit.

Pour résoudre une contrainte $\mathsf{let}\ x = C_1\ \mathsf{in}\ C_2$, il faut résoudre d'abord C_1, puis C_2. Pour exprimer cela dans le système de réécriture, nous devons manipuler deux contextes différents pour la contrainte let : un premier qui sera utilisé le temps de résoudre la contrainte dans la partie gauche ($S[\mathsf{let}\ x = \lambda X.[]\ \mathsf{in}\ C]$), et un autre pour la contrainte dans la partie droite ($S[\mathsf{let}\ x = \lambda X.U\ \mathsf{in}\ []]$).

3.1.2 Un schéma polymorphe comme une contrainte

Une remarque importante concernant la contrainte let est qu'une abstraction de contrainte peut être vue comme un schéma polymorphe (et inversement). Ainsi l'abstraction $\lambda X.\exists \bar{Y}.(X = ... = T \wedge U')$ est équivalente à un certain schéma de la forme $\forall \bar{a}.T$.

Exemple 3.1.1.

Le schéma sous forme de λ-abstraction $\lambda X.\exists X_1 X_2.\ X$ is $X_1 \to X_2 \wedge X_1$ is X_2 peut-être vu comme le schéma $\forall a.a \to a$. ◊

La "forme contrainte" d'un schéma polymorphe a l'avantage que l'on peut facilement l'enrichir à mesure que l'on enrichit le système de type (et le langage des contraintes), contrairement à la "forme schéma" qui est spécifique à un système de type donné. On peut comprendre une contrainte let $x = \lambda X.\exists \bar{Y}.(X = ... = T \wedge U')$ in C comme une contrainte let qui associe à x un schéma $\forall \bar{a}.T$, où on a : (i) remplacé autant que possible les variables dans T pour la mettre sous forme de grand terme et (ii) extrudé autant que possible les variables existentielles pour les faire sortir à l'extérieur du lambda.

Le reste de la présentation étant en petit terme, nous faisons le choix de nous en tenir à la version de la contrainte let en petit terme.

3.2 Règles de réécriture

3.2.1 Règles d'unification

On peut définir un contexte d'unification pour manipuler la composante U en profondeur :

$$\mathcal{U} ::= [] \mid [] \wedge U \mid U \wedge [] \mid \exists X.[]$$

Pour procéder à des unifications dans la composante U, on spécifie des règles de réécriture, qui la simplifie (potentiellement jusqu'à true) ou se réduisent à false.

$(\exists \bar{X}.U_1) \wedge U_2$	$\to \exists \bar{X}.(U_1 \wedge U_2)$	si $\bar{X} \# ftv(U_2)$	
$X = \epsilon \wedge X = \epsilon'$	$\to X = \epsilon = \epsilon'$		(FUSE)
$X = X = \epsilon$	$\to X = \epsilon$		
$s\ \bar{X} = s\ \bar{Y} = \epsilon$	$\to \bar{X} = \bar{Y} \wedge s\ \bar{X} = \epsilon$		
T	$\to$ true		
$U \wedge$ true	$\to U$		
$s\ \bar{X} = s'\ \bar{Y} = \epsilon$	$\to$ false	si $s \neq s'$	(CLASH-1)
$s\bar{X} = a = \epsilon$	$\to$ false		(CLASH-2)
$a = b = \epsilon$	$\to$ false	si $a \neq b$	(CLASH-3)
U	$\to$ false	si U est cyclique (voir 2.6)	
$\mathcal{U}$[false]	$\to$ false	où $\mathcal{U}$ est un contexte d'unification	

L'opération $\bar{X} \# ftv(U_2)$ présente dans la première règle vérifie que les deux ensembles sont bien disjoints. On peut remarquer la règle (FUSE) qui fusionne deux multi-équations, et les règles (CLASH-1), etc qui échouent quand on essaye de fusionner ensemble des structures et/ou des variables rigides incompatibles.

3.2.2 Règles de réécriture du triplet $S\ ;\ U\ ;\ C$

Procéder à des unifications Une première règle de réécriture consiste simplement à faire avancer l'unification d'une étape :

$$S\ ;\ U\ ;\ C \to \quad S\ ;\ U'\ ;\ C \qquad \text{(UNIF)}$$
$$\text{si } U \to U'$$

Simplifier la contrainte courante On peut ensuite énumérer les règles qui, en fonction de la forme de la contrainte courante C, déplacent des bouts de cette contrainte vers les autres composantes (S ou U). Quand une contrainte courante C lie une variable qui est libre dans U, il faut faire un alpha-renommage de C, c'est-à-dire substituer dans C cette variable par une nouvelle variable fraîche.

$$
\begin{array}{lll}
S\ ;\ U\ ;\ X \text{ is } T & \to S\ ;\ U \wedge X = T\ ;\ \mathsf{true} & \\
S\ ;\ U\ ;\ C_1 \wedge C_2 & \to S[[] \wedge C_2]\ ;\ U\ ;\ C_1 & \\
S\ ;\ U\ ;\ \exists \bar{X}.C & \to S[\exists \bar{X}.[]]\ ;\ U\ ;\ C & \text{si } \bar{X} \# ftv(U) \\
S\ ;\ U\ ;\ \mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2 & \to S[\mathsf{let}\ x = \lambda X.[]\ \mathsf{in}\ C_2]\ ;\ U\ ;\ C_1 & \text{si } X \# ftv(U) \\
S\ ;\ U\ ;\ x\ X & \to S\ ;\ U\ ;\ C[X/Y] & \text{si } S(x) = \lambda Y.C \\
S\ ;\ U\ ;\ \forall a.C & \to S[\forall a.[]]\ ;\ U\ ;\ C & \text{si } a \# ftv(U)
\end{array}
$$

Pour le cas de l'instanciation, on définit $S(x)$ qui renvoie l'abstraction associée à x dans le plus proche contexte :

$$
\begin{array}{lll}
S[[] \wedge C](x) & = S(x) & \\
S[\exists X[]](x) & = S(x) & \text{si } X \# ftv(S(x)) \\
S[\forall a[]](x) & = S(x) & \text{si } a \# ftv(S(x)) \\
S[\mathsf{let}\ y = \lambda Y.C_1\ \mathsf{in}\ []](x) & = S(x) & \text{si } x \neq y \wedge Y \# ftv(S(x)) \\
S[\mathsf{let}\ x = \lambda Y.C_1\ \mathsf{in}\ []](x) & = \lambda Y.C_1 &
\end{array}
$$

Extruder des quantifications existentielles On définit des règles qui extrudent des quantifications existentielles :

$$
\begin{array}{lll}
S\ ;\ \exists \bar{X}.U\ ;\ C & \to S[\exists \bar{X}.[]]\ ;\ U\ ;\ C & \\
 & \quad \text{si } \bar{X} \# ftv(C) & \\
S[(\exists \bar{X}.[]) \wedge C] & \to S[\exists \bar{X}.([] \wedge C)] & \\
 & \quad \text{si } \bar{X} \# ftv(C) & \\
S[\mathsf{let}\ x = \lambda X.\exists \bar{Y}.[]\ \mathsf{in}\ C]\ ;\ U\ ;\ \mathsf{true} & \to S[\exists \bar{Y}.\mathsf{let}\ x = \lambda X.[]\ \mathsf{in}\ C]\ ;\ U\ ;\ \mathsf{true} & \text{(LET-ALL)} \\
 & \quad \text{si } \bar{Y} \# ftv(C) \wedge \exists X.U \text{ détermine } \bar{Y} & \\
S[\mathsf{let}\ x = \lambda X.C\ \mathsf{in}\ \exists \bar{Y}.[]] & \to S[\exists \bar{Y}.\mathsf{let}\ x = \lambda X.C\ \mathsf{in}\ []] & \\
 & \quad \text{si } \bar{Y} \# ftv(C) & \\
S[\forall \bar{a}.\exists \bar{Y}\bar{Z}.[]]\ ;\ U\ ;\ \mathsf{true} & \to S[\exists \bar{Y}.\forall \bar{a}.\exists \bar{Z}.[]]\ ;\ U\ ;\ \mathsf{true} & \\
 & \quad \text{si } \forall \bar{a} \exists \bar{Z}.U \text{ détermine } \bar{Y} &
\end{array}
$$

La règle (LET-ALL) permet de faire remonter une quantification existentielle en dehors d'un let. Pour cela, il faut s'assurer que la variable quantifiée est bien déterminée par la contrainte d'unification courante U. Si on retirait cette condition, on pourrait perdre le polymorphisme du let, puisque le schéma de type associé à x ne pourrait être instancié qu'une seule fois.

Exemple 3.2.1.

Prenons par exemple la contrainte let suivante :

$$\mathsf{let}\ f = \lambda X.\exists Y.X \text{ is } Y \to \mathtt{int}\ \mathsf{in}\ f\ (\mathtt{bool} \to \mathtt{int}) \wedge f\ (\mathtt{int} \to \mathtt{int})$$

En gardant cette forme, la contrainte est résoluble : on peut instancier l'argument de f avec n'importe quel type, puisqu'à chaque application de $\lambda X.\exists Y.X$ is $Y \to \mathtt{int}$ on pourra choisir une valeur arbitraire pour Y. Mais si on était autorisé à faire remonter la quantification de Y, on pourrait obtenir la contrainte :

$$\exists Y.\ \mathsf{let}\ f = \lambda X.X \text{ is } Y \to \mathtt{int}\ \mathsf{in}\ f\ (\mathtt{bool} \to \mathtt{int}) \wedge f\ (\mathtt{int} \to \mathtt{int})$$

Ici, Y est fixé avant la partie gauche du let, ce qui empêche Y d'être instanciée par différentes valeurs. La contrainte devient irrésoluble, car Y ne peut pas valoir à la fois `int` et `bool`. ◊

Simplifier le contexte Ci-dessous, des règles qui travaillent à partir de la forme du contexte courant, après avoir réussi à réécrire la contrainte courante en true. Ici, il n'est plus question d'extruder des quantifications existentielles, mais de dépiler le contexte ou de simplifier la composante d'unification :

$$S[] \wedge C] \ ; \ U \ ; \ \mathsf{true} \qquad \rightarrow S \ ; \ U \ ; \ C$$

(COMPRESS)
$$S[\mathsf{let}\ x = \lambda X.\exists \bar{Y} Z.[]\ \mathsf{in}\ C] \ ; \ Z = V = \epsilon \wedge U \ ; \ \mathsf{true} \quad \rightarrow S[\mathsf{let}\ x = \lambda X.\exists \bar{Y}.[]\ \mathsf{in}\ C] \ ; \ V = \theta(\epsilon) \wedge \theta(U) \ ; \ \mathsf{true} \quad \text{si } Z \neq V \wedge \theta = [Z \mapsto V]$$

La règle (COMPRESS) élimine une variable d'inférence superflue, en la remplaçant par une variable égale dans la composante d'unification.

Exemple 3.2.2.

Un exemple consiste en une contrainte simplifiée obtenue à partir du lambda-terme $\lambda x.\lambda y.x$, pour une variable d'inférence W, i.e. une simplification de $[\![\lambda x.\lambda y.x : W]\!]$:

$$\exists X_1 X_2.W \ \mathsf{is}\ X_1 \rightarrow X_2 \wedge \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2$$

Le triplet initial s'écrit donc :

$$[] \ ; \ \mathsf{true} \ ; \ \exists X_1 X_2.W \ \mathsf{is}\ X_1 \rightarrow X_2 \wedge \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2$$

En appliquant deux fois la règle de passage au contexte des contraintes existentielles, on obtient :

$$[][\exists X_1 X_2.[]] \ ; \ \mathsf{true} \ ; \ W \ \mathsf{is}\ X_1 \rightarrow X_2 \wedge \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2$$

On se retrouve avec une conjonction comme contrainte courante, que l'on passe au contexte. On peut à nouveau passer au contexte :

$$[][\exists X_1 X_2.[]][[] \wedge \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2] \ ; \ \mathsf{true} \ ; \ W \ \mathsf{is}\ X_1 \rightarrow X_2$$

Notre contrainte courante n'est plus qu'une unification, que l'on peut transférer dans la composante d'unification :

$$[][\exists X_1 X_2.[]][[] \wedge \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2] \ ; \ W = X_1 \rightarrow X_2 \ ; \ \mathsf{true}$$

On a simplifié la contrainte courante jusqu'à la réduire à true. On peut désormais simplifier le contexte. On commence par le dépiler :

$$[][\exists X_1 X_2.[]] \ ; \ W = X_1 \rightarrow X_2 \ ; \ \exists Y_1 Y_2.X_2 \ \mathsf{is}\ Y_1 \rightarrow Y_2 \wedge X_1 \ \mathsf{is}\ Y_2$$

En appliquant des réécritures similaires, on se retrouve avec le triplet :

$$[][\exists X_1 X_2 Y_1 Y_2.[]] \ ; \ W = X_1 \rightarrow X_2 \wedge X_2 = Y_1 \rightarrow Y_2 \wedge X_1 = Y_2 \ ; \ \mathsf{true}$$

La contrainte courante est true, la pile des contextes ne contient que des quantifications de variables flexibles, on peut s'arrêter là ! Le type de l'expression est assigné à la variable W : $X_1 \rightarrow Y_1 \rightarrow X_1$. ◊

Gestion du polymorphisme On peut définir les règles suivantes, qui concernent la gestion du polymorphisme :

$$\begin{array}{ll}
\text{(BUILD-SCHEME)} & \\
S[\mathsf{let}\ x = \lambda X.\exists \bar{Y}.[]\ \mathsf{in}\ C]\ ;\ U_1 \wedge U_2\ ;\ \mathsf{true} & \to S[\mathsf{let}\ x = \lambda X.\exists \bar{Y}.U_2\ \mathsf{in}\ []]\ ;\ U_1\ ;\ C \\
& \quad \text{si } X\bar{Y}\#ftv(U_1) \wedge \exists X\bar{Y}.U_2 \equiv \mathsf{true} \\
\\
\text{(POP-ENV)} & \\
S[\mathsf{let}\ x = \lambda X.\ U'\ \mathsf{in}\ []]\ ;\ U\ ;\ \mathsf{true} & \to S\ ;\ U\ ;\ \mathsf{true} \\
\\
\text{(POP-ALL)} & \\
S[\forall \bar{a}.\exists \bar{X}.[]]\ ;\ U_1 \wedge U_2\ ;\ \mathsf{true} & \to S\ ;\ U_1\ ;\ \mathsf{true} \\
& \quad \text{si } \bar{a}\bar{X}\#ftv(U_1) \wedge \exists \bar{X}.U_2 \equiv \mathsf{true}
\end{array}$$

Les règles (POP-...) dépilent un contexte S devenu inutile après simplification de la contrainte courante. La règle (POP-ENV) permet de sortir d'un contexte let qui lie une variable x, lorsque cette variable n'est plus référenciée dans le reste du solveur. La règle (POP-ALL) sépare U en deux parties : U_1 qui ne contient pas de variables jeunes ($\bar{Y}\bar{X}\#ftv(U_1)$) et U_2 qui contient des variables jeunes $\bar{X}$, qui déterminent les valeurs de $\bar{Y}$ (car $\exists \bar{X}.U_2 \equiv \mathsf{true}$). On sait alors, en particulier, que $\forall \bar{a}.\exists \bar{X}.U_2 \equiv \mathsf{true}$ et on peut sortir du contexte $\forall \bar{a}\exists \bar{X}.[]$, en faisant disparaître au passage U_2, puisqu'on sait la résoudre.

La règle (BUILD-SCHEME) est utilisée quand on a finit de résoudre la contrainte dans la partie gauche d'un let. Pour la comprendre, il faut voir qu'on peut diviser la composante d'unification en deux : d'un côté une partie U_1 qui dépend de variables "vieilles" introduites antérieurement au let courant, de l'autre une partie U_2 qui ne contraint que des variables "jeunes" introduites dans le let courant. Quand on a fini de résoudre la contrainte de la partie gauche du let, on peut vérifier facilement la condition $\exists X\bar{Y}.U_2 \equiv \mathsf{true}$, qui n'est pas évidente à vérifier dans le cas d'une contrainte arbitraire $C \equiv \mathsf{true}$. Cette condition nous assure que U_2 ne contraint que des variables "jeunes", puisqu'il est suffisant de quantifier sur les variables locales $X, \bar{Y}$ pour prouver l'équivalence avec true. Cette contrainte d'unification U_2 nous sert à exprimer le schéma associé à x. On peut ensuite tenter de résoudre la partie droite du let, c'est-à-dire la contrainte C. Pour cela, on peut s'appuyer sur la contrainte d'unification U_1 qui, elle, n'est pas liée à la construction d'un schéma de type pour x (car $X\bar{Y}\#ftv(U_1)$). Un des exemples ci-dessous illustre l'utilisation de cette règle.

Exemple 3.2.3.

Reprenons un exemple déjà donné plus haut quand nous définissions la génération de contrainte :

$$[\![\mathsf{let}\ f = \lambda x.\ x\ \mathsf{in}\ f : W]\!]$$

$$= \mathsf{let}\ f = \lambda X.\exists X_1 X_2.X\ \mathsf{is}\ X_1 \to X_2 \wedge (\mathsf{let}\ f = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2)\ \mathsf{in}\ f\ W$$

Une réécriture de cette contrainte peut s'effectuer ainsi :

$$\begin{array}{ll}
 & []\ ;\ \mathsf{true}\ ;\ \mathsf{let}\ f = \lambda X.\exists X_1 X_2.X\ \mathsf{is}\ X_1 \to X_2 \wedge (\mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2)\ \mathsf{in}\ f\ W \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W]\ ;\ \mathsf{true}\ ;\ X\ \mathsf{is}\ X_1 \to X_2 \wedge \mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2 \\
\to^* & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W]\ ;\ X = X_1 \to X_2\ ;\ \mathsf{let}\ x = \lambda Y.\ Y\ \mathsf{is}\ X_1\ \mathsf{in}\ x\ X_2 \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W][\mathsf{let}\ x = \lambda Y.\ []\ \mathsf{in}\ x\ X_2]\ ;\ X = X_1 \to X_2\ ;\ Y\ \mathsf{is}\ X_1 \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W][\mathsf{let}\ x = \lambda Y.\ []\ \mathsf{in}\ x\ X_2]\ ;\ X = X_1 \to X_2 \wedge Y = X_1\ ;\ \mathsf{true}
\end{array}$$

Ici on a fini de résoudre la contrainte à l'intérieur du let. On cherche donc à appliquer la règle BUILD-SCHEME. On peut choisir pour cela de prendre $U_1 := X = X_1 \to X_2$ et $U_2 := Y = X_1$. On a alors bien $Y\#U_1$ et $\exists Y.U_2 \equiv \mathsf{true}$. Le triplet se réécrit donc en :

$$
\begin{array}{ll}
 & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W][\mathsf{let}\ x = \lambda Y.Y = X_1\ \mathsf{in}\ []]\ ;\ X = X_1 \to X_2\ ;\ x\ X_2 \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W][\mathsf{let}\ x = \lambda Y.Y = X_1\ \mathsf{in}\ []]\ ;\ X = X_1 \to X_2\ ;\ (\lambda Y.Y\ \mathsf{is}\ X_1)\ X_2 \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W][\mathsf{let}\ x = \lambda Y.Y = X_1\ \mathsf{in}\ []]\ ;\ X = X_1 \to X_2\ ;\ X_1\ \mathsf{is}\ X_2 \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1 X_2.[]\ \mathsf{in}\ f\ W]\ ;\ X = X_1 \to X_2 \wedge X_1 = X_2\ ;\ \mathsf{true} \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1.[]\ \mathsf{in}\ f\ W]\ ;\ X = X_1 \to X_1\ ;\ \mathsf{true} \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1.X = X_1 \to X_1\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ f\ W \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1.X = X_1 \to X_1\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ (\lambda X.\exists X_1.X = X_1 \to X_1)\ W \\
\to & [][\mathsf{let}\ f = \lambda X.\exists X_1.X = X_1 \to X_1\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ \exists X_1.\ W\ \mathsf{is}\ X_1 \to X_1 \\
\to^* & [][\mathsf{let}\ f = \lambda X.\exists X_1.X = X_1 \to X_1\ \mathsf{in}\ []]\ ;\ \exists X_1.\ W = X_1 \to X_1\ ;\ \mathsf{true} \\
\to & []\ ;\ \exists X_1.\ W = X_1 \to X_1\ ;\ \mathsf{true} \\
\to & [][\exists X_1.[]]\ ;\ \ W = X_1 \to X_1\ ;\ \mathsf{true}
\end{array}
$$

On voit donc que le terme $\mathsf{let}\ f = \lambda x.\ x\ \mathsf{in}\ f$ est typable et qu'il a pour type $X_1 \to X_1$ pour un certain X_1, et que le triplet final est bien une forme normale telle que décrite dans le lemme ci-dessous. ◊

Exemple 3.2.4.

Pour illustrer le fonctionnement de la règle BUILD-SCHEME, nous pouvons examiner la résolution de la contrainte générée par le terme $\lambda k.\ \mathsf{let}\ f = \lambda x.\lambda y.k\ y\ \mathsf{in}\ ()$. Une version simplifiée de cette contrainte, pour W la variable d'inférence du terme entier, est :

$$
\begin{array}{lll}
\exists K. & W = K \to & K' \\
\wedge & \mathsf{let}\ k = \lambda V. & V\ \mathsf{is}\ K\ \mathsf{in} \\
 & \mathsf{let}\ f = \lambda V'. & \exists X\ Y\ Z.V\ \mathsf{is}\ {}'X \to Y \to Z \\
 & & \wedge \mathsf{let}\ x = \lambda X'.X'\ \mathsf{is}\ X\ \mathsf{in} \\
 & & \quad \mathsf{let}\ y = \lambda Y'.Y'\ \mathsf{is}\ Y\ \mathsf{in} \\
 & & \quad \exists Z'\ Z''.\ Z'' = Z' \to Z \wedge k\ Z'' \wedge y\ Z' \\
 & \mathsf{in} & \\
 & K'\ \mathsf{is}\ \mathtt{unit} &
\end{array}
$$

Cette contrainte se réécrit, après quelques étapes, en la contrainte suivante :

$$
\begin{array}{l}
[\exists K.[]][\mathsf{let}\ k = \lambda V.V\ \mathsf{is}\ K\ \mathsf{in}\ []][\exists Y Z.[]][\mathsf{let}\ f = \lambda V'.\exists X.[]\ \mathsf{in}\ K'\ \mathsf{is}\ \mathtt{unit}]\ ; \\
W = K \to K' \wedge V' = X \to Y \to Z \wedge K = Y \to Z\ ; \\
\mathsf{true}
\end{array}
$$

On peut alors séparer la composante d'unification en deux composantes pour appliquer la règle BUILD-SCHEME, en posant :

$$U_1 := W = K \to K' \wedge K = Y \to Z \qquad\qquad U_2 := V' = X \to Y \to Z$$

U_1 vérifie bien $V', X \# ftv(U_1)$. Il nous reste à vérifier que $\exists V' X.U_2$ est bien équivalente à true. Dans la contrainte $\exists V' X.V' = X \to Y \to Z$ les variables Y et Z sont fixées, il nous faut donc trouver une façon d'assigner des types à V' et X. On peut prendre n'importe quelle valeur pour X et assigner à V' le type $X \to Y \to Z$. On aura donc bien, quelques soient les valeurs de Y et Z, $(\exists V' X.V' = X \to Y \to Z) \equiv \mathsf{true}$.

On peut donc appliquer la règle, et on obtient :

$$
\begin{array}{l}
[\exists K.[]][\mathsf{let}\ k = \lambda V.V\ \mathsf{is}\ K\ \mathsf{in}\ []][\exists Y Z.[]][\mathsf{let}\ f = \lambda V'.\exists X.V\ \mathsf{is}\ {}'X \to Y \to Z\ \mathsf{in}\ []]\ ; \\
W = K \to K' \wedge K = Y \to Z\ ; \\
K'\ \mathsf{is}\ \mathtt{unit}
\end{array}
$$

◊

Faire échouer des contraintes universelles Enfin, on définit des règles d'échec dans 2 cas de réécriture de contrainte universelle : lorsqu'une variable rigide s'échappe de sa portée et lorsque qu'on l'égalise avec un terme qui n'est pas une variable flexible.

$$\begin{array}{lll} S[\forall a.\exists \bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true} & \to \mathsf{false} & \text{si } a \prec_U^* Z \wedge Z \notin \bar{Y} \\ S[\forall a.\exists \bar{Y}[]] \ ; \ a = s \ \bar{Z} = \epsilon \ ; \ \mathsf{true} & \to \mathsf{false} & \end{array}$$

La première règle nous dit que si une variable rigide a est dominée (voir 2.5) par une variable vieille Z, alors la contrainte est invalide. En effet, a est quantifiée universellement, et une variable définie plus tôt ne peut donc pas dépendre de sa valeur.

Exemple 3.2.5.

Pour illustrer ces règles de réécriture, prenons un cas où la contrainte se réécrit à false :

$$\begin{array}{l} [] \ ; \ \mathsf{true} \ ; \ \exists X \forall a.X \ \mathsf{is} \ a \to a \\ \to^* [][\exists X.[]][\forall a.[]] \ ; \ X = a \to a \ ; \ \mathsf{true} \\ \to \mathsf{false} \end{array}$$

Dans cet exemple, a sort de sa portée, puisqu'elle est quantifiée universellement mais égalisée avec une variable définie à l'extérieur (qui a une portée plus étendue). Nous reviendrons sur l'échappement de portée plus loin, quand nous aborderons plus en détails le traitement des variables rigides et l'implémentation. ◊

Lemme 3.2. Le système de réécriture ($\to$) est fortement normalisant.

Lemme 3.3. Une forme normale pour le système de réécriture $\to$ est dans un des trois cas :

(i) S ; U ; $x \ T$ où x n'est définie par aucun contexte de let

(ii) S ; false ; true

(iii) χ ; U ; true où χ est un contexte existentiel et U est une conjonction de multi-équations satisfiable.

Lemme 3.4. Si S ; U ; $C \to S'$; U' ; C' alors $S[U \wedge C] \equiv S'[U' \wedge C']$

Les preuves de ces trois lemmes sont données dans Pottier and Rémy (2005). La présentation du système de réécriture donnée ici est légèrement différente, notamment en ce qui concerne la contrainte let et la traduction en petits termes. En particulier, quelques règles de manipulation des multi-équations ou des règles de réécriture diffèrent. Cependant les mécaniques décrites dans leur travail sont essentiellement les mêmes que celles présentées plus haut, et il est possible assez directement de faire correspondre les deux présentations.

Chapitre 4

GADTs, égalités de types

4.1 Rappels sur les GADTs

4.1.1 Des types de données algébriques plus expressifs

Les GADTs, pour *Generalized Algebraic Data Types*, sont des types de données algébriques dont les constructeurs peuvent fournir explicitement une instanciation (voir par exemple Xi, Chen, and Chen (2003)). Cela permet une approche plus fine du typage, puisque différents constructeurs d'un même type peuvent fournir des instanciations différentes dans sa déclaration.

Exemple 4.1.1.

Ce mécanisme nous permet, par exemple, d'exprimer les entiers naturels sous la forme suivante :

```
type _ nat =
| Zero : unit nat
| Succ : 'a nat -> (unit * 'a) nat

# Zero;;
- : unit nat = Zero

# Succ (Succ Zero);;
- : (unit * (unit * unit)) nat = Succ (Succ Zero)
```

Le type du constructeur `Zero` est spécifié comme étant `unit nat`, tandis que le type du constructeur `Succ` nous apprend qu'il attend un argument de type `'a nat` et construit un élément de type `(unit * 'a) nat`. Le type nous permet donc de distinguer précisément les éléments de `nat` : il suffit de dénombrer le nombre de types produits imbriqués dans l'argument de `nat` pour déterminer la valeur d'un élément juste par son type ! ◊

Cela permet en particulier une discipline de typage plus sécurisée, puisqu'on peut discriminer les cas directement grâce aux types des éléments que l'on manipule.

Exemple 4.1.2.

Ainsi pour écrire une fonction de soustraction par 1, on peut éliminer le cas problématique `Zero`, grâce au typage de l'argument :

```
let minus_one (n : (unit * 'a) nat) : 'a nat =
  match n with
  | Succ n' -> n'
```

Comme l'argument `n` de la fonction est de type `(unit * 'a) nat` on peut en déduire qu'il s'agit d'un successeur. De plus, en renvoyant un élément de type `'a nat`, on s'assure que l'on a bien effectué une soustraction par `1`. ◊

GADTs et variables existentielles

La déclaration d'un constructeur de GADT peut mentionner des variables dans ses arguments qui n'apparaissent pas dans le type de retour. On parle alors de variables existentielles.

Exemple 4.1.3.

Les GADTs sont notamment utiles pour déclarer des arbres de syntaxe abstraite :

```
type _ term =
| Var : string -> 'a term
| Lam : string * 'b term -> ('a -> 'b) term
| App : ('a -> 'b) term * 'a term -> 'b term

# let id = Lam ("x", Var "x");;
val id : ('a -> 'b) term = Lam ("x", Var "x")

# App (id, Var "y");;
- : 'a term = App (Lam ("x", Var "x"), Var "y")
```

Le type du constructeur `App` fait intervenir une variable existentielle `'a` dans sa déclaration, qui apparaît dans les arguments mais pas dans le type de retour. ♢

Écrire des fonctions plus expressives grâce aux GADTs

En annotant les types des arguments et de retour des fonctions, on peut écrire des fonctions plus expressives :

Exemple 4.1.4.

```
type 'a expr =
| Bool : bool -> bool expr
| Int : int -> int expr

let eval (type a) (e : a expr) : a =
  match e with
  | Bool b -> b
  | Int i -> i
```

On obtient une fonction `eval` qui renvoie un type différent selon qu'elle reçoit en argument une expression booléenne ou entière. Sans GADTs et sans la variable rigide `a` annotant le type de retour, on ne pourrait pas écrire une telle fonction. On peut noter qu'en OCaml on ne parle pas de variable rigide mais de "type abstrait local". ♢

Les GADTs permettent ainsi une plus grand latitude pour écrire des fonctions qui se comportent différemment selon le type de leurs arguments.

Exhaustivité plus fine du filtrage par motifs

De plus, en indiquant dans les déclarations de type comment instancier chaque constructeur, on peut typer des filtrages par motifs plus fins. En effet, on peut éliminer, grâce à la discipline de type, les cas qui ne peuvent pas se produire.

Exemple 4.1.5.

```
# fun (Int i : int expr) -> i
- : int expr -> int = <fun>
```

Dans cet exemple, l'annotation `int expr` permet de discriminer entre les différents constructeurs : le seul possible étant `Int`, l'argument de la fonction `Int i` est sous la bonne forme (la forme `Bool b` serait détectée comme une erreur). ♢

Cela se révèle particulièrement utile pour exprimer des filtrages par motifs plus concis :

Exemple 4.1.6.

```
let binop (type a) (e1 : a expr) (e2 : a expr) : a =
  match (e1, e2) with
  | (Bool b1, Bool b2) -> b1 && b2
  | (Int i1, Int i2) -> i1 + i2
```

Le filtrage par motifs de cette fonction est exhaustif puisque les types des deux expressions `e1` et `e2` sont contraints d'être égaux, et les seules instanciations possibles de `a expr` sont `bool expr` et `int expr`. En outre, rajouter des cas qui mélangent `Int` et `Bool` produirait une erreur de typage : cela signifierait que `int expr` et `bool expr` sont tous deux équivalents à `a expr` dans la même branche du filtrage par motifs. ◊

Alors comment raffiner le typage des types de données algébriques simples pour typer les GADTs, et comment détecter les erreurs ?

4.1.2 Les GADTs introduisent des égalités de types

À la différence des types de données algébriques simples, les GADTs peuvent apprendre au typeur des égalités entre types. Ainsi, dans l'exemple précédent, une égalité entre `a expr` et `bool expr` est introduite localement dans la branche dont le motifs est `(Bool b1, Bool b2)`. Le typeur peut en déduire que, dans cette branche, `a` est égale à `bool`, et donc que la conjonction booléenne `b1 && b2` est compatible avec le type `a`. Dans l'autre branche, c'est le type `int` qui est égalisé avec `a`, ce qui permet à l'addition `i1 + i2` d'être compatible localement avec `a`. Le type de retour de la fonction `a` est donc compatible tantôt avec `bool`, tantôt avec `int`.

En fait, sans l'annotation du type de retour de la fonction au type `a`, la fonction serait mal typée, car on s'attendrait à avoir le même type de retour dans les deux branches (soit `bool` soit `int`).

Ce mécanisme de typage, qui garde trace d'égalités entre types, permet donc d'exprimer des fonctions plus riches, en leur donnant des types contenant des variables rigides qui sont égalisées localement à différents types.

Grâce aux GADTs, on peut définir un type qui représente une égalité entre types :

```
type (_, _) eq = Refl : ('a, 'a) eq
```

On peut ainsi introduire une égalité dans le contexte :

```
let succ_and_discard (type a) (e : (a,int) eq) (n : a) =
  match e with
  | Refl -> (* introduce type equality a = int *)
      let _ = n + 1 in ()
```

Ici une égalité entre `a` et `int` est introduite dans le contexte, ce qui permet de voir `n` comme un entier et de lui ajouter `1`.

Ce motif de programmation sera récurrent dans les exemples que nous donnerons, et nous introduisons du sucre syntaxique : `use t : eq ty1 ty2 in u` introduisant le type `(ty1,ty2) eq` dans le contexte de typage de `u`. Pour ce faire, il faut également fournir un témoin `t` de cette égalité entre `ty1` et `ty2`. L'exemple précédent se réécrit avec cette construction :

```
let succ_and_discard (type a) (e : (a,int) eq) (n : a) =
  use e in (* introduce type equality a = int *)
  let _ = n + 1 in ()
```

4.1.3 Des GADTs avec les constructions Refl et use ... in ...

Pour formaliser ces nouvelles constructions, on étend la grammaire s des types de base :

$$s ::= \cdots \mid \mathsf{eq}$$

ainsi que la grammaire τ des types utilisateurs :

$$\tau ::= \cdots \mid \mathsf{eq}\ \tau\ \tau$$

Le constructeur Refl est de type $\mathsf{eq}\ \tau\ \tau$ pour un certain type τ. Notons que Refl représente une égalité entre des types qui ont éventuellement des variables rigides aux feuilles, mais pas des flexibles, puisqu'il s'agit de types utilisateurs. Ajouter la possibilité d'introduire des égalités avec des flexibles pourrait poser des problèmes de principalité.

Comme il est impossible de savoir à priori quel τ choisir, nous annotons les différentes occurrences de Refl par ce type et nous notons Refl_τ. La règle de typage de Refl_τ est donc simplement :

$$\frac{}{\Gamma \vdash \mathsf{Refl}_\tau : \mathsf{eq}\ \tau\ \tau}$$

Typer la construction use ... in ... revient essentiellement à rajouter une égalité dans le contexte de typage :

$$\frac{\Gamma \vdash t : \mathsf{eq}\ \tau_1\ \tau_2 \qquad \Gamma, \tau_1 = \tau_2 \vdash u : \tau}{\Gamma \vdash \mathsf{use}\ t : \mathsf{eq}\ \tau_1\ \tau_2\ \mathsf{in}\ u : \tau}$$

On vérifie que t est bien un témoin de $\mathsf{eq}\ \tau_1\ \tau_2$, et que u est typable avec un environnement dans lequel on rajoute l'égalité $\tau_1 = \tau_2$. Cette égalité pourra servir à justifier des jugements de typage, comme nous le verrons juste après. Par ailleurs, on précise un type $\mathsf{eq}\ \tau_1\ \tau_2$ qui annote t, car on ne peut pas deviner τ_1 et τ_2 avant de typer u sans perdre la principalité.

Pour utiliser les égalités de cet environnement de typage, on définit une règle de conversion entre deux types différents :

$$\frac{\Gamma \vdash t : \tau' \qquad \Gamma \vdash \tau' = \tau}{\Gamma \vdash (t : \tau') : \tau}$$

Cela nous demande de définir un jugement $\Gamma \vdash \tau = \tau'$. Pour prouver un tel jugement, on peut se servir d'égalités de types introduites auparavant, et des règles de symétrie, de transitivité, de congruence et de décomposition.

$$\frac{}{\Gamma \vdash \tau = \tau} \qquad \frac{(\tau = \tau') \in \Gamma}{\Gamma \vdash \tau = \tau'} \qquad \frac{\Gamma \vdash \tau' = \tau}{\Gamma \vdash \tau = \tau'} \qquad \frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash \tau' = \tau''}{\Gamma \vdash \tau = \tau''}$$

$$\frac{\Gamma \vdash \tau_1 = \tau'_1 \quad \dots \quad \Gamma \vdash \tau_n = \tau'_n}{\Gamma \vdash s(\tau_i)_i = s(\tau'_j)_j} \qquad \frac{\Gamma \vdash s(\tau_i)_i = s(\tau'_i)_i}{\Gamma \vdash \tau_i = \tau'_i}$$

Enfin, nous rajoutons un terme absurd_τ qui indique au typeur qu'une égalité incohérente a été rajoutée dans l'environnement :

$$\frac{\Gamma \vdash s\ \bar{\tau_1} = s'\ \bar{\tau_2} \qquad s \neq s'}{\Gamma \vdash \mathsf{absurd}_\tau : \tau}$$

Si Γ permet de prouver une égalité entre deux types incohérents alors absurd_τ est typable au type τ.

Nous venons de décrire une version simplifiée des GADTs, qui repose sur l'utilisation des constructions Refl et use ... in ..., mais les principes de typage sont suffisants pour implémenter une version plus étendue des GADTs.

La traduction d'un GADT du langage source fait apparaître une preuve d'égalité en argument.

Programme OCaml:

```
type _ expr =
| Int : int -> int expr
| Bool : bool -> bool expr
```

Notre langage cible (syntaxe imaginaire):

```
type α expr =
| Int of int * eq α int
| Bool of bool * eq α bool
```

Ainsi nous fournissons une preuve d'égalité grâce au constructeur Refl en argument du constructeur `Int` :

```
Int 42                          Int (42, Refl int)
```

Dans un filtrage par motifs, chaque branche introduit l'égalité correspondante :

```
let eval
  (type a) (e : a expr) : a
= match e with
  | Int n -> n
  | Bool b -> b
```

```
let eval α (e : α expr) =
  match e with
  | Int (n : int, w : eq α int) ->
      use w in (n : α)
  | Bool (b : bool, w : eq α bool) ->
      use w in (b : α)
```

La construction absurd peut-être utilisée dans le filtrage par motifs, pour signaler des branches inaccessibles :

```
let eq e1 e2 =
  match e1, e2 with
  | Int n1, Int n2 -> n1 = n2
  | Bool b1, Bool b2 -> b1 = b2
  | Int _, Bool _ -> .
  | Bool _, Int _ -> .
```

```
let eq e1 e2 =
  match e1, e2 with
  | Int (n1, _), Int (n2, _) -> n1 = n2
  | Bool (b1, _), Bool (b2, _) -> b1 = b2
  | Int _ w1, Bool _ w2 ->
      use w1 in (* Introduce (α = int) *)
      use w2 in (* Introduce (α = bool) *)
      absurd (* Now the context is inconsistent
                as we can derive (int = bool) *)
  | Bool (_, w1), Int (_, w2) ->
      use w1 in use w2 in absurd
```

4.2 Échappement d'égalité et ambiguïté

Nous avons vu comment rajouter les GADTs à Système F et comment exprimer ces GADTs à partir d'un noyau qui s'appuie sur un type égalité.

Nous nous intéressons dans la suite à la façon d'inférer des types pour les GADTs. Le problème se pose, cependant, de s'assurer que les égalités de types ne sont utilisées que dans des endroits où elles sont définies. On a vu, par exemple, que pour les filtrages par motifs, chaque branche pouvait introduire ses propres égalités. Mais chacune de ces égalités ne sont valables que dans leurs branches respectives. Lorsqu'une égalité est utilisée par le typeur en dehors de la zone dans laquelle elle est définie, on dit qu'elle s'échappe de sa portée.

Exemple 4.2.1.

```
type _ ty = Int : int ty | Bool : bool ty

let default_or_val (type a) (ty : a ty) default (v : a) =
  match ty with
  | Int -> if default then 0 else v
  | Bool -> if default then false else v
(*
Error: This expression has type a = int
       but an expression was expected of type int
       This instance of int is ambiguous:
       it would escape the scope of its equation
*)
```

Dans la première branche du filtrage par motif, le type de `v` est déduit de son annotation : `a`. Mais l'instruction `if` force le type de `0` et `v` à être égaux. Mais on ne peut pas déduire que ces types sont égaux, on le voit bien avec la deuxième branche du filtrage par motif qui devrait forcer `a` à être égale à `bool`. Ainsi, comme le fait remarquer le message d'erreur du typeur, l'égalité entre `int` et `a` s'échappe.

◊

Pour faire apparaître plus visiblement les égalités de types manipulées, on peut écrire des exemples avec le type `eq` (4.1.2).

```
let succ (type a) (e : (a,int) eq) (n : a) =
  match e with
```

```
  | Refl -> (* introduce type equality a = int *)
      n + 1
```

Pour le type de retour, on a le choix entre `int` et `a` : l'opérateur `+` renvoie un entier, mais on pourrait choisir de voir cet entier comme une expression de type `a`. Dans l'exemple suivant c'est encore plus clair :

```
let f (type a) (x : (a,int) eq) (y : a) =
  match x with Refl -> if y > 0 then y else 0
(*
   Error: This expression has type int but an expression was expected of type
          a = int
        This instance of int is ambiguous:
        it would escape the scope of its equation
*)
```

Ici, la première branche du `if` renvoie une valeur de type `a` et la deuxième une valeur de type `int`. Les deux branches renvoient donc des valeurs de types différents pour le monde extérieur qui ne sait pas que $a =$ `int` : il y a une ambiguïté. Si on veut un système de types principal, il faut rejeter cet exemple pour ne pas avoir à faire un choix arbitraire entre `a` et `int` quand on type cette expression. Cela correspond à ne pas laisser l'égalité $a =$ `int` s'échapper de sa portée.

Pour passer outre cette restriction, il suffit d'annoter le type de retour de la fonction. Deux annotations sont possibles :

```
let f (type a) (x : (a,int) eq) (y : a) : int =
  match x with Refl -> if y > 0 then y else 0
```

ou

```
let f (type a) (x : (a,int) eq) (y : a) : a =
  match x with Refl -> if y > 0 then y else 0
```

On peut remarquer que ces deux types ne sont pas compatibles, ce qui montre bien que le programme (sans l'annotation) n'a pas de type principal.

Chapitre 5

Contrainte d'hypothèse d'égalités

5.1 Une nouvelle contrainte

Pour représenter l'introduction d'une égalité de types sous forme de contraintes, on étend simplement le langage des contraintes. On y rajoute une contrainte d'hypothèse d'égalité (ou contrainte d'implication) :

$$C ::= \cdots \mid (\tau_1 = \tau_2) \Rightarrow C$$

Cette contrainte introduit l'égalité $\tau_1 = \tau_2$ dans le contexte dans lequel sera résolue la contrainte C.

La contrainte générée pour la construction use ... in est essentiellement une contrainte d'hypothèse d'égalité :

$$[\![\mathsf{use}\ t\ :\ \mathsf{eq}\ \tau_1\ \tau_2\ \mathsf{in}\ u : X]\!] ::=$$

$$(\exists Y.(\!|Y \sim \mathsf{eq}\ \tau_1\ \tau_2|\!) \wedge [\![t : Y]\!]) \wedge (\tau_1 = \tau_2) \Rightarrow [\![u : X]\!]$$

On a imposé aux deux types de l'égalité de ne pas contenir de variable d'inférence. En effet, il semble difficile de définir un typage principal si on introduit des égalités sur des variables d'inférence qui peuvent dépendre de bouts de programme qui seront traités ultérieurement. Par ailleurs, cela ne pose pas de problème car une égalité de types introduite par un terme $\mathsf{use}\ \cdots : \mathsf{eq}\ \tau_1\ \tau_2\ \mathsf{in}\ \ldots$ nous fournit déjà des types τ_1, τ_2 introduits par l'utilisateur, sans variables d'inférence.

Quant à la contrainte générée pour Refl, elle s'assure simplement qu'on peut écrire son type sous la forme $\mathsf{eq}\ \tau\ \tau$:

$$[\![\mathsf{Refl} : X]\!] ::= \exists Y.X\ \mathsf{is}\ \mathsf{eq}\ Y\ Y$$

Dans la suite, nous discutons de comment choisir une sémantique pour la contrainte d'hypothèse d'égalité. Quant aux règles de résolution du solveur, elles n'ont rien d'évidentes, et seront discutées dans le prochain chapitre.

5.1.1 Arborescence de dérivation

L'introduction de variables rigides dans une contrainte crée une arborescence de dérivations sémantiques différentes : il y a une dérivation par instance de variable rigide. C'est ce qui est établi par la prémisse de la règle (FORALL) pour la contrainte $\forall a.C$ qui introduit une quantification universelle sur tous les types ground t.

Une même égalité de type peut donc prendre un sens différent selon l'instanciation de ses variables rigides. Elle peut être rendue vraie par une certaine instance mais fausse pour toutes les autres.

Exemple 5.1.1.

Dans la contrainte $\forall a.a = \texttt{list(int)}$, la plupart des choix d'instanciation pour X rendent l'égalité fausse. Quand on applique FORALL on obtient :

$$\frac{\forall t, \quad \emptyset, [a \mapsto t] \models a \text{ is } \texttt{list(int)}}{\emptyset, \emptyset \models \forall a.\ a \text{ is } \texttt{list(int)}}$$

que l'on peut voir comme une règle avec une infinité de prémisse, une par type de t :

$$\frac{\ldots \quad \emptyset, [a \mapsto \texttt{bool}] \models a \text{ is } \texttt{list(int)} \quad \ldots \quad \emptyset, [a \mapsto \texttt{list(int)}] \models a \text{ is } \texttt{list(int)} \quad \ldots}{\emptyset, \emptyset \models \forall a.\ a \text{ is } \texttt{list(int)}}$$

$\Diamond$

5.2 Sémantique naturelle

Une sémantique naturelle pour la contrainte d'introduction d'égalité est l'implication logique : si l'égalité est vraie, alors on doit résoudre la contrainte.

$$\frac{\gamma(\tau_1) = \gamma(\tau_2) \quad \Longrightarrow \quad E; \gamma \models C}{E; \gamma \models (\tau_1 = \tau_2) \Rightarrow C}$$

Si l'égalité introduit une incohérence (une égalité entre 2 types ground différents), la contrainte est vérifiée par l'absurde. L'incohérence peut se produire d'une suite d'égalités introduites (par transitivité), par exemple dans la contrainte $(a = \texttt{int}) \Rightarrow (a = \texttt{bool}) \Rightarrow C$. En fait, si une égalité de type contient des variables rigides, quasiment toutes leurs instances rendront l'égalité fausse et donc le typage incohérent !

Exemple 5.2.1.

1. On peut, par exemple, résoudre la contrainte $\forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}$.

   Après avoir introduit la variable universelle et la variable existentielle, on s'aperçoit qu'on doit trouver un type t', solution de X, qui satisfait $(a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}$.

   $$\frac{\dfrac{\forall t,\ \exists t', \quad [a \mapsto t, X \mapsto t'] \models (a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}}{\forall t, \quad [a \mapsto t] \models \exists X.(a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}}}{\emptyset \models \forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}}$$

   On a alors deux choix possibles pour t' qui permettent de résoudre X is int : soit int directement, soit t, l'instance de a, qui est égal à int d'après l'hypothèse de l'implication.

   $$\frac{\forall t, \quad t = \texttt{int} \quad \Longrightarrow \quad \overline{\emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models X \text{ is } \texttt{int}}}{\forall t, \quad \emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models (a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}}$$

   $$\frac{\forall t, \quad t = \texttt{int} \quad \Longrightarrow \quad \overline{\emptyset; [a \mapsto t, X \mapsto t] \models X \text{ is } \texttt{int}}}{\forall t, \quad \emptyset; [a \mapsto t, X \mapsto t] \models (a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}}$$

2. Comme prévu, on rejette la contrainte suivante :

   $$\forall a\ \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}$$

   L'arbre de dérivation pour cette contrainte est :

$$\dfrac{\dfrac{\forall t,\ \exists t', \quad [a \mapsto t, X \mapsto t'] \models X \text{ is } a \wedge X \text{ is } \texttt{int}}{\forall t, \quad [a \mapsto t] \models \exists X. X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\emptyset \models \forall a\ \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

Comme aucune égalité entre a et `int` n'a été introduite, on ne peut pas trouver de t' qui vaille à la fois l'un et l'autre, mis à part le cas particulier de la branche où t est `int`.

◊

Cependant, cette sémantique ne convient pas tout à fait pour identifier les programmes qui sont typés par notre solveur : elle permet de résoudre des contraintes générées par des programmes dont le typage est ambigu et que notre solveur rejette. De tels programmes sont rejetés aussi par OCaml, et mal typés dans le système décrit dans Garrigue and Rémy (2013). Prenons la contrainte $\forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}$. Il s'agit d'une partie de la contrainte obtenue à partir d'un exemple déjà donné plus haut :

```
let f (type a) (x : (a,int) eq) (y : a) =
  use x : eq a int in
  if y > 0 then y else 0
```

Quand on applique les règles de sémantique, on obtient :

$$\dfrac{\dfrac{\forall t,\ \exists t', \quad \emptyset; [a \mapsto t, X \mapsto t'] \models (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}{\forall t, \quad \emptyset; [a \mapsto t] \models \exists X.(a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\emptyset; \emptyset \models \forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

De là, le jugement de satisfiabilité a deux dérivations possibles, selon que l'on choisisse $X = t$ ou $X = \texttt{int}$.

- En choisissant $X = t$:

$$\dfrac{\dfrac{\forall t, \quad \dfrac{}{t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto t] \models X \text{ is } a} \quad \dfrac{}{t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto t] \models X \text{ is } \texttt{int}}}{\forall t, \quad t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto t] \models X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t, \quad \emptyset; [a \mapsto t, X \mapsto t] \models (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

- En choisissant $X = \texttt{int}$:

$$\dfrac{\dfrac{\forall t, \quad \dfrac{}{t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models X \text{ is } a} \quad \dfrac{}{t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models X \text{ is } \texttt{int}}}{\forall t, \quad t = \texttt{int} \implies \emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t, \quad \emptyset; [a \mapsto t, X \mapsto \texttt{int}] \models (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

Ces deux dérivations correspondent aux deux façons d'annoter le programme avec un type de retour pour la fonction `f`:

```
let f (type a) (x : (a,int) eq) (y : a) : a =
  use x : eq a int in
  if y > 0 then y else 0
```

ou

```
let f (type a) (x : (a,int) eq) (y : a) : int =
  use x : eq a int in
  if y > 0 then y else 0
```

Mais aucun des deux typages n'est principal, l'annotation est nécessaire. On est trop permissif, X ne devrait pas pouvoir être de deux types à la fois et cette contrainte devrait être rejetée.

Notre solveur va détecter cette ambiguïté et renvoyer `UNSAT`, il n'est plus complet par rapport à cette sémantique. Pour continuer de coller à notre solveur, on introduit une autre sémantique, qui repose sur une notion de types ambivalents et qui rejette bien cette contrainte.

5.3 Sémantique ambivalente

5.3.1 Types ambivalents

Après l'introduction d'une égalité $a = \texttt{int}$, on peut avoir envie de ne pas choisir entre les deux types, en voyant les différentes occurrences de ces types comme un type *ambivalent* : ni `int` ni `a`, mais un ensemble {`int`, `a`} qui contient ces deux types. Un tel type ambivalent n'est bien sûr correct que dans la zone du programme où l'égalité est disponible dans le contexte et pas dans le reste.

Il nous faut déterminer une façon d'utiliser l'ambivalence, mais aussi un critère pour la restreindre et empêcher le typage de programmes que l'on voudrait rejeter. On veut pouvoir typer localement des bouts de programme avec plusieurs types, mais qu'à l'extérieur le typage continue de bien se comporter et d'avoir de bonnes propriétés (typage principal). On cherche donc un système dans lequel des égalités entre types (ambivalents) peuvent permettre de typer une partie de programme, mais induisent une ambiguïté lorsqu'elles sortent de leurs portées. C'est une approche qui est développée par Garrigue and Rémy (2013), sur laquelle on s'est appuyé.

Les types ambivalents ψ, sont des ensembles de types "ground" que l'on note :

$$\psi ::= \emptyset \mid t \approx \psi$$

Pour résumer, un type ambivalent correspond localement à plusieurs types possibles, mais en sortant de la portée des égalités, l'ambivalence doit disparaître pour ne pas créer d'ambiguïté. Pour faire disparaître l'ambivalence d'un type, on peut annoter le programme.

5.3.2 Un jugement ambivalent

Notre sémantique *ambivalente* ($\models^{\mathsf{amb}}$) exprime la possibilité d'assigner localement un type ambivalent à une expression, tout en s'assurant qu'à l'extérieur son type reste unique. On veut retranscrire, dans une approche par contraintes, un système de typage qui nous permet d'accepter des programmes utilisant des égalités entre types, mais qui sont suffisamment annotés pour ne pas rompre avec la principalité.

Dans cette sémantique, les variables d'inférence $X, Y, ...$ ont des types ambivalents ψ comme témoins. À première vue, rajouter cette possibilité pourrait sembler produire une sémantique plus permissive que la précédente, alors que l'on cherche au contraire à rejeter plus de contraintes ! En fait, la façon dont nous gérons les contextes incohérents dans cette nouvelle sémantique rejette certaines contraintes qui étaient acceptées par la sémantique naturelle définie plus haut (sans en accepter de nouvelles). Elle rejette les contraintes ambiguës, qui comportent des égalités de types s'échappant des zones dans lesquelles elles sont définies.

La sémantique naturelle s'exprime par une implication dans la méta-logique des règles. Selon que l'égalité introduite était vraie ou fausse, on se retrouve avec un environnement de typage cohérent ou incohérent, avec lequel résoudre la partie droite de l'implication. Dans la sémantique ambivalente, on retranscrit cette idée en rajoutant de l'information dans le contexte de typage : on trace simplement la cohérence avec un bit κ dans les jugements.

$$\kappa ::= \mathsf{true} \mid \mathsf{false}$$

Les jugements prennent la forme :

$$\kappa; E; \gamma \models^{\mathsf{amb}} C$$

Ce jugement sémantique et les règles décrites ci-dessous sont assez simples, pourtant cela capture la notion d'ambivalence qui nous intéresse.

5.3.3 Principales règles

Afin de tirer parti de l'ambivalence, il nous faut revisiter la sémantique de la construction $X \ \mathsf{is}\ T$, qui était l'égalité jusqu'ici. En effet, les variables d'inférences X, Y, etc, qui peuvent être contenues dans T, peuvent désormais avoir des solutions ambivalentes.

Dans le cas où T est de la forme $s(Y_i)_i$, on aplatit les types ambivalents de chaque Y_i, et on teste l'*égalité* avec l'ensemble ambivalent des types de X qui ont s comme constructeur de tête.

$$\frac{\gamma(X)_{|s} = \{s(\tau_i)_i \mid \tau_i \in \gamma(Y_i)\}}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } s(Y_i)_i}$$

Exemple 5.3.1.

Prenons $T = Y \to \texttt{bool}$ et $\gamma = [Y \mapsto \{t,\ \texttt{int}\}; X \mapsto \{\texttt{unit},\ t \to \texttt{bool},\ \texttt{int} \to \texttt{bool}\}]$. On a bien :

$$\frac{\{t \to \texttt{bool},\ \texttt{int} \to \texttt{bool}\} = \gamma(X)_{|(\to)}}{\mathsf{true}; \emptyset; \gamma \models^{\mathsf{amb}} X \text{ is } T}$$

$\Diamond$

Dans le cas où T est une variable rigide (mais plus généralement dans le cas on il ne contient aucune variable d'inférence, et est donc de la forme τ), la sémantique de X is τ devient un test d'*appartenance à un ensemble* :

$$\frac{\gamma(\tau) \in \gamma(X)}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } \tau}$$

En effet, dans notre sémantique seules les variables d'inférences sont associées à des types ambivalents, mais ce n'est pas le cas des variables rigides, qui elles sont toujours associées à des types non-ambivalents : $\gamma(\tau)$ correspond donc à un unique type non-ambivalent.

Exemple 5.3.2.

Prenons $\gamma = [X \mapsto \{\texttt{unit},\ t \to \texttt{bool},\ \texttt{int} \to \texttt{bool}\}]$. On a bien, par exemple :

$$\frac{\gamma(t \to \texttt{bool}) = t \to \texttt{bool} \in \gamma(X)}{\mathsf{true}; \emptyset; \gamma \models^{\mathsf{amb}} X \text{ is } t \to \texttt{bool}}$$

$\Diamond$

Enfin, le sens d'une contrainte X is Y entre deux variables d'inférence est l'égalité ensembliste entre leurs témoins dans l'environnement de typage :

$$\frac{\gamma(X) = \gamma(Y)}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } Y}$$

Comme expliqué plus haut, le jugement de cette sémantique ambivalente prend en compte la cohérence (ou l'incohérence) de l'environnement de typage. Le seul moyen d'introduire une incohérence consiste à résoudre une contrainte d'hypothèse d'égalité entre deux types incompatibles. C'est ce qui est reflété par la sémantique de cette contrainte :

$$\frac{\kappa \wedge (\gamma(\tau_1) = \gamma(\tau_2)); E; \gamma \models^{\mathsf{amb}} C}{\kappa; E; \gamma \models^{\mathsf{amb}} (\tau_1 = \tau_2) \Rightarrow C}$$

Dans cette formulation, il n'y a pas besoin de stocker des égalités. Ici κ est un booléen qui indique si l'environnement est cohérent ou non. En faisant la conjonction avec $\gamma(\tau_1) = \gamma(\tau_2)$, on obtient une nouvelle valeur booléenne.

Ceci dit, des types universellement quantifiés peuvent rendre une égalité vraie ou fausse selon leurs valeurs. Il y a alors des versions de la dérivation dont la valeur de κ diffère.

Exemple 5.3.3.

Pour traiter la contrainte $\forall a.\ (a = \texttt{int}) \Rightarrow \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}$, on va quantifier sur tous les types ground t :

$$\frac{\forall t \quad t = \texttt{int}; \emptyset; [a \mapsto t] \models^{\mathsf{amb}} \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}}{\forall t \quad \mathsf{true}; \emptyset; [a \mapsto t] \models^{\mathsf{amb}} (a = \texttt{int}) \Rightarrow \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

On voit donc que la valeur de κ, ici $t = \texttt{int}$, n'est pas la même selon la valeur de t. ◊

La sémantique du $\exists X.C$ est modifiée, pour rendre compte de l'ambivalence qui peut être introduite par une hypothèse d'égalité. En effet, en résolvant une contrainte existentielle sous une hypothèse d'égalité, on peut être amené à attribuer un type ambivalent à une variable d'inférence. Le témoin pour X est choisi différemment selon que l'environnement de typage est cohérent ou pas. Si l'environnement est incohérent, on a introduit une égalité entre deux types t et t' incompatibles, donc on peut être amené à utiliser des types ambivalents ψ contenant des types différents qui mentionnent t et t'.

$$\frac{\exists \psi \quad \text{if } \kappa \text{ then } |\psi| = 1 \quad \kappa; E; \gamma[X \mapsto \psi] \models^{\mathsf{amb}} C}{\kappa; E; \gamma \models^{\mathsf{amb}} \exists X.C}$$

La valeur de κ nous indique quel type assigner à X :

- si κ est vrai, alors l'environnement est cohérent, le témoin de X n'a pas le droit d'être ambivalent. En d'autres termes, c'est un singleton : le cardinal de ψ, noté $|\psi|$, vaut 1. On peut donc simplifier la règle, dans ce cas, là en écrivant directement :

$$\frac{\exists t \quad \mathsf{true}; E; \gamma[X \mapsto \{t\}] \models^{\mathsf{amb}} C}{\mathsf{true}; E; \gamma \models^{\mathsf{amb}} \exists X.C}$$

- si κ est faux, alors l'environnement est incohérent, le témoin de X peut être ambivalent ($|\psi| \geq 1$). C'est la seule règle dans laquelle on peut utiliser $\kappa = \mathsf{false}$ pour introduire de l'ambivalence.

Les règles pour la conjonction et la quantification universelle sont les mêmes que dans la sémantique naturelle. Voici donc les règles de la sémantique ambivalente :

$$\frac{\gamma(X)_{|s} = \{s(\tau_i)_i \mid \tau_i \in \gamma(Y_i)\}}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } s(Y_i)_i} \qquad \frac{\gamma(\tau) \in \gamma(X)}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } \tau} \qquad \frac{\gamma(X) = \gamma(Y)}{\kappa; E; \gamma \models^{\mathsf{amb}} X \text{ is } Y}$$

$$\frac{\kappa; E; \gamma \models^{\mathsf{amb}} C_1 \quad \kappa; E; \gamma \models^{\mathsf{amb}} C_2}{\kappa; E; \gamma \models^{\mathsf{amb}} C_1 \wedge C_2} \qquad \frac{\forall t \quad \kappa; E; \gamma[a \mapsto t] \models^{\mathsf{amb}} C}{\kappa; E; \gamma \models^{\mathsf{amb}} \forall a.C}$$

$$\frac{\kappa \wedge (\gamma(\tau_1) = \gamma(\tau_2)); E; \gamma \models^{\mathsf{amb}} C}{\kappa; E; \gamma \models^{\mathsf{amb}} (\tau_1 = \tau_2) \Rightarrow C} \qquad \frac{\exists \psi \quad \text{if } \kappa \text{ then } |\psi| = 1 \quad \kappa; E; \gamma[X \mapsto \psi] \models^{\mathsf{amb}} C}{\kappa; E; \gamma \models^{\mathsf{amb}} \exists X.C}$$

Exemple 5.3.4.

Pour comparer avec la sémantique naturelle, on peut regarder quelle forme prendrait un arbre de dérivation de la sémantique ambivalente sur l'exemple donné plus haut :

$$\forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}$$

On s'aperçoit qu'on est bloqué : on ne peut pas choisir un type pour X. Par soucis de simplification de la lecture, on omet l'environnement polymorphe qui n'est pas utilisé dans cet exemple.

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall t\ \exists t', \quad (t = \texttt{int}); [a \mapsto t; X \mapsto \{t'\}] \models^{\mathsf{amb}} X \text{ is } a \quad (t = \texttt{int}); [a \mapsto t; X \mapsto \{t'\}] \models^{\mathsf{amb}} X \text{ is } \texttt{int}}{\forall t\ \exists t', \quad (t = \texttt{int}); [a \mapsto t; X \mapsto \{t'\}] \models^{\mathsf{amb}} X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t\ \exists t' \quad \mathsf{true}; [a \mapsto t; X \mapsto \{t'\}] \models^{\mathsf{amb}} (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t \quad \mathsf{true}; [a \mapsto t] \models^{\mathsf{amb}} \exists X.(a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\mathsf{true}; \emptyset \models^{\mathsf{amb}} \forall a\ \exists X.\ (a = \texttt{int}) \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

Dans le cas où t vaut `int`, il n'y a pas de problème, on peut choisir $t' = \texttt{int}$ et on pourra dériver directement :

$$\dfrac{}{\mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}\}] \models^{\mathsf{amb}} X \text{ is } a} \qquad \dfrac{}{\mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}\}] \models^{\mathsf{amb}} X \text{ is } \texttt{int}}$$

Mais pour toutes les autres valeurs de t, il nous faudrait à la place de $\{t'\}$ un type ambivalent $\psi \supseteq \{t, \texttt{int}\}$, alors que $\gamma(X)$ est contraint à être un singleton. La dérivation est bloquée, cette contrainte n'a donc pas de solution dans la sémantique ambivalente, ce qui est cohérent avec le comportement de notre solveur qui la rejette.

L'introduction de X a lieu à un niveau de la dérivation où l'environnement de type est cohérent (κ vaut true), ce qui restreint son type à être un singleton. La contrainte d'hypothèse d'égalité, qui n'apparaît que plus tard dans la contrainte, et donc dans la dérivation, introduit une ambivalence entre a et `int` dans la suite de la contrainte. Cette ambivalence n'est donc pas reflétée dans l'environnement, ce qui rend la contrainte insatisfiable.

En quantifiant X après l'introduction de l'égalité $a = \texttt{int}$, on peut choisir un témoin ambivalent pour X dans les cas où $t \neq \texttt{int}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall t\ \exists \psi, \quad \text{if } t = \texttt{int} \text{ then } |\psi| = 1 \quad (t = \texttt{int}); [a \mapsto t; X \mapsto \psi] \models^{\mathsf{amb}} X \text{ is } a \wedge X \text{ is } \texttt{int}}{\forall t, \quad (t = \texttt{int}); [a \mapsto t] \models^{\mathsf{amb}} \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t, \quad \mathsf{true}; [a \mapsto t] \models^{\mathsf{amb}} (a = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\mathsf{true}; \emptyset \models^{\mathsf{amb}} \forall a.\ (a = \texttt{int}) \Rightarrow \exists X.\ X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

Pour fermer la dérivation, il faut raisonner par cas sur la valeur de t :

- $t = \texttt{int}$

  Alors ψ est en fait un singleton $\{t'\}$, que l'on peut choisir comme étant $\{\texttt{int}\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}\}] \models^{\mathsf{amb}} X \text{ is } a} \quad \dfrac{}{\mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}\}] \models^{\mathsf{amb}} X \text{ is } \texttt{int}}}{\exists t', \quad \mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \{t'\}] \models^{\mathsf{amb}} X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\mathsf{true}; [a \mapsto \texttt{int}] \models^{\mathsf{amb}} \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\mathsf{true}; [a \mapsto \texttt{int}] \models^{\mathsf{amb}} (a = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

- $t \neq \texttt{int}$

  Dans ce cas on a $\kappa = \mathsf{false}$ et on peut choisir ψ Nécessairement ψ doit contenir `int` et t. Choisir directement le type ambivalent $\{\texttt{int}, t\}$ fonctionne :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{false}; [a \mapsto t; X \mapsto \{\texttt{int}, t\}] \models^{\mathsf{amb}} X \text{ is } a} \quad \dfrac{}{\mathsf{false}; [a \mapsto t; X \mapsto \{\texttt{int}, t\}] \models^{\mathsf{amb}} X \text{ is } \texttt{int}}}{\exists \psi, \quad \mathsf{false}; [a \mapsto t; X \mapsto \psi] \models^{\mathsf{amb}} X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t \neq \texttt{int}, \quad \mathsf{false}; [a \mapsto t] \models^{\mathsf{amb}} \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\forall t \neq \texttt{int}, \quad \mathsf{true}; [a \mapsto t] \models^{\mathsf{amb}} (a = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

$\Diamond$

5.3.4 Polymorphisme

La règle pour la contrainte let et celle, en miroir, pour son instanciation, se basaient sur un environnement de typage non ambivalent. Il faut donc les modifier également. Pour commencer, nous introduisons une nouvelle catégorie syntaxique pour les schémas ambivalents, semblable à celle des types ambivalents :

$$\xi ::= \forall \bar{X}.\Psi \qquad\qquad \Psi ::= T \mid T \approx \Psi$$

Nous considérons, dans la sémantique, des schémas clos, dans lesquels les variables libres des Ψ sont toutes issues de la quantification $\forall \bar{X}$ en tête de schéma.

La notion d'instanciation s'étend naturellement aux schémas ambivalents et aux types ambivalents en la définissant comme l'instanciation de chaque schéma ambivalent par un des types ambivalents :

$$\forall \bar{X}.(T_0 \approx \cdots \approx T_n) \preceq (T_0 \approx \cdots \approx T_n)[\bar{X} \backslash \bar{t}]$$

La partie gauche d'une contrainte let peut désormais avoir un schéma ambivalent ξ et son instanciation peut résulter en un type ambivalent.

$$\frac{\exists \xi, \forall \psi \quad \text{if } \kappa \text{ then } |\xi| = 1 \qquad \xi \preceq \psi \implies \kappa; E; \gamma[X \mapsto \psi] \models^{\mathsf{amb}} C_1 \qquad \kappa; E[f \mapsto \xi]; \gamma \models^{\mathsf{amb}} C_2}{\kappa; E; \gamma \models^{\mathsf{amb}} \mathsf{let}\ f = \lambda X.C_1\ \mathsf{in}\ C_2}$$

$$\frac{E(f) \preceq \gamma(X)}{\kappa; E; \gamma \models^{\mathsf{amb}} f\ X}$$

Exemple 5.3.5.

On peut regarder un exemple qui illustre la gestion des schémas ambivalents :

$$\forall a.(a = \mathtt{int}) \Rightarrow \exists W.\mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W$$

En appliquant les règles de sémantique, on obtient :

$$\dfrac{\dfrac{\dfrac{\dfrac{\forall t \exists \psi_W, \quad \text{if } t = \mathtt{int} \text{ then } |\psi_W| = 1 \quad (t = \mathtt{int}); \emptyset; [a \mapsto t; W \mapsto \psi_W] \models^{\mathsf{amb}} \mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W}{\forall t, \quad (t = \mathtt{int}); \emptyset; [a \mapsto t] \models^{\mathsf{amb}} \exists W.\mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W}}{\forall t, \quad \mathsf{true}; \emptyset; [a \mapsto t] \models^{\mathsf{amb}} (a = \mathtt{int}) \Rightarrow \exists W.\mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W}}{\mathsf{true}; \emptyset; \emptyset \models^{\mathsf{amb}} \forall a.(a = \mathtt{int}) \Rightarrow \exists W.\mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W}$$

On peut s'intéresser au cas où $t \neq \mathtt{int}$. En posant $\gamma = [a \mapsto t; W \mapsto \psi_W]$, on obtient :

$$\dfrac{\exists \xi \forall \psi_X, \quad \dfrac{\psi_X \supseteq \{t, \mathtt{int}\}}{\xi \preceq \psi_X \implies \mathsf{false}; \emptyset; [a \mapsto t; W \mapsto \psi_W; X \mapsto \psi_X] \models^{\mathsf{amb}} X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}} \qquad \dfrac{\xi = E(f) \preceq \gamma(W) = \psi_W}{\mathsf{false}; [f \mapsto \xi]; [a \mapsto t; W \mapsto \psi_W] \models^{\mathsf{amb}} f\ W}}{\mathsf{false}; \emptyset; \gamma \models^{\mathsf{amb}} \mathsf{let}\ f = \lambda X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ f\ W}$$

On peut choisir $\xi = \forall \emptyset.\{t, \mathtt{int}\}$. Pour satisfaire $\xi \preceq \psi_X$, il faut $\psi_X = \{t, \mathtt{int}\}$. En effet, on a bien $\forall \emptyset.t \preceq t$ et $\forall \emptyset.\mathtt{int} \preceq \mathtt{int}$. Quant à ψ_W, on peut choisir $\{t, \mathtt{int}\}$ pour la même raison.

$\Diamond$

5.4 Correspondance entre les deux sémantiques

Pour se ramener à la sémantique naturelle, plus permissive, il suffit de rajouter une règle d'absurdité en plus de toutes les autres, qui permet de résoudre n'importe quelle contrainte dans un contexte incohérent. On obtient une sémantique ($\models^{\mathsf{amb}'}$).

$$\frac{}{\mathsf{false}; E; \gamma \models^{\mathsf{amb}'} C}$$

Cette règle permet d'obtenir deux dérivations sur l'exemple précédent, car il n'y a plus besoin d'exhiber un $\psi_X \supseteq \{t, \texttt{int}\}$, on peut prendre $\psi_X = \{\texttt{int}\}$ ou $\psi_X = \{t\}$:

- dans le cas $t \neq \texttt{int}$ le contexte est incohérent (κ est faux) et on utilise la règle d'absurdité
- dans le cas $t = \texttt{int}$, on pouvait déjà fermer la dérivation sans utiliser la règle d'absurdité.

Une solution γ en sémantique ambivalente associe aux variables d'inférence des ensembles de types ambivalents, là où la sémantique naturelle leur associe un unique type. Nous définissons une notion `Sing` pour la transformation d'une solution en sémantique naturelle vers une solution en sémantique ambivalente, en transformant chaque témoin en singleton.

$$\begin{aligned} \texttt{Sing}(\gamma_n) &= \gamma \text{ tel que } \forall a,\ \gamma(a) = \gamma_n(a) \text{ et } \forall X,\ \gamma(X) = \{\gamma_n(X)\} \\ \texttt{Sing}(E_n) &= E \text{ tel que } \forall x, \text{ si } E_n(x) = \forall X.T \text{ alors } E(x) = \forall X.\{T\} \end{aligned}$$

Théoreme 5.1. Soit γ_n et $\gamma_a = \texttt{Sing}(\gamma_n)$. Soit E un environnement polymorphe et $E_a = \texttt{Sing}(E)$

$$E; \gamma_n \models C \iff \mathsf{true}; E_a; \gamma_a \models^{\mathsf{amb}'} C$$

Preuve
Par induction sur C

- X is T

$$\begin{array}{rl} & \gamma_n \models X \text{ is } T \\ \iff & \gamma_n(X) = \gamma_n(T) \end{array}$$

On raisonne par cas sur la forme de T :

– τ

$$\begin{array}{rl} & \gamma_n(X) = \gamma_n(\tau) \\ \iff & \gamma_a(X) = \{\gamma_a(\tau)\} \\ \iff & \gamma_a(X) \ni \gamma_a(\tau) \text{ car } \gamma_a \text{ ne contient que des singletons} \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} X \text{ is } \tau \end{array}$$

– Y

$$\begin{array}{rl} & \gamma_n(X) = \gamma_n(Y) \\ \iff & \gamma_a(X) = \gamma_a(Y) \text{ car } \gamma_a \text{ ne contient que des singletons} \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} X \text{ is } Y \end{array}$$

– $s(Y_i)_i$

$$\begin{array}{rl} & \gamma_n(X) = \gamma_n(s(Y_i)_i) \\ \iff & \gamma_a(X) = \{\gamma_a(s(Y_i)_i)\} \\ \iff & \gamma_a(X) = \{s(\gamma_a(Y_i)_i)\} \\ \iff & \gamma_a(X)_{|s} = \{s(\gamma_a(Y_i)_i)\} \text{ car } \gamma_a \text{ ne contient que des singletons} \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} X \text{ is } s(Y_i)_i \end{array}$$

- $C_1 \wedge C_2$

$$\begin{array}{ll} & \gamma_n \models C_1 \wedge C_2 \\ \iff & \gamma_n \models C_1 \text{ et } \gamma_n \models C_2 \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} C_1 \text{ et } \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} C_2 \text{ par HI} \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} C_1 \wedge C_2 \end{array}$$

- $\exists X.C$

$$\begin{array}{ll} & \gamma_n \models \exists X.C \\ \iff & \exists t.\ \gamma_n[X \mapsto t] \models C \\ \iff & \exists t.\ \gamma_a[X \mapsto \{t\}] \models^{\mathsf{amb}'} C \text{ par HI} \\ \iff & \exists \psi.\quad |\psi| = 1 \quad \gamma_a[X \mapsto \psi] \models^{\mathsf{amb}'} C \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} \exists X.C \end{array}$$

- $\forall a.C$

$$\begin{array}{ll} & \gamma_n \models \forall a.C \\ \iff & \forall t. \gamma_n[a \mapsto t] \models C \\ \iff & \forall t. \mathsf{true}; \gamma_a[a \mapsto t] \models^{\mathsf{amb}'} C \text{ par HI} \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} \forall a.C \end{array}$$

- $\mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2$

$$\begin{array}{ll} & E; \gamma_n \models \mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2 \\ \iff & \exists \bar{Y} T, \forall t, \quad \forall \bar{Y}.T \preceq t \implies E; \gamma_n[X \mapsto t] \models C_1 \text{ et } E[x \mapsto \forall \bar{Y}.T]; \gamma_n \models C_2 \\ \iff & \exists \bar{Y} T, \forall t, \quad \forall \bar{Y}.T \preceq t \implies \mathsf{true}; E; \gamma_a[X \mapsto \{t\}] \models^{\mathsf{amb}'} C_1 \\ & \text{et } \mathsf{true}; E[x \mapsto \forall \bar{Y}.\{T\}]; \gamma_a \models^{\mathsf{amb}'} C_2 \text{ par HI} \\ \iff & \exists \bar{Y} T, \forall \psi, \qquad |\forall \bar{Y}.\{T\}| = 1, \quad \forall \bar{Y}.\{T\} \preceq \psi \implies \mathsf{true}; E_a; \gamma_a[X \mapsto \psi] \models^{\mathsf{amb}'} C_1 \\ & \text{et } \mathsf{true}; E_a[x \mapsto \forall \bar{Y}.\{T\}]; \gamma_a \models^{\mathsf{amb}'} C_2 \\ \iff & \mathsf{true}; E_a; \gamma_a \models^{\mathsf{amb}'} \mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2 \end{array}$$

- $x\ X$

  Si aucune contrainte let n'a introduit de variable x, alors la contrainte n'est pas résoluble. Sinon, x est associée, dans la sémantique naturelle, à une schéma $\forall \bar{Y}.T$, et on a :

$$\begin{array}{ll} & E; \gamma_n \models x\ X \\ \iff & \exists \bar{t}, \quad E(x) = \forall \bar{Y}.T \text{ et } \gamma_n(X) = T[\bar{t}/\bar{Y}] \\ \iff & \exists \bar{t}, \quad E_a(x) = \forall \bar{Y}.\{T\} \text{ et } \gamma_a(X) = \{T[\bar{t}/\bar{Y}]\} \\ \iff & \mathsf{true}; E_a; \gamma_a \models^{\mathsf{amb}'} x\ X \end{array}$$

- $(\tau_1 = \tau_2) \Rightarrow C$

$$\begin{array}{ll} & \gamma_n \models (\tau_1 = \tau_2) \Rightarrow C \\ \iff & (\gamma_n(\tau_1) = \gamma_n(\tau_2)) \implies \gamma_n \models C \end{array}$$

  On raisonne par cas sur la valeur de $\gamma_n(\tau_1) = \gamma_n(\tau_2)$:

– $\gamma_n(\tau_1) = \gamma_n(\tau_2)$

$$\begin{array}{ll} & (\gamma_n(\tau_1) = \gamma_n(\tau_2)) \Longrightarrow \gamma_n \models C \\ \iff & \gamma_n \models C \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} C \text{ par HI} \\ \iff & (\mathsf{true} \wedge \gamma_a(\tau_1) = \gamma_a(\tau_2)); \gamma_a \models^{\mathsf{amb}'} C \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} (\tau_1 = \tau_2) \Rightarrow C \end{array}$$

– $\gamma_n(\tau_1) \neq \gamma_n(\tau_2)$

$$\begin{array}{ll} & (\gamma_n(\tau_1) = \gamma_n(\tau_2)) \Longrightarrow \gamma_n \models C \\ \iff & \mathsf{false} \Longrightarrow \gamma_n \models C \text{ ce qui est toujours vrai (par la sémantique logique de l'implication)} \\ \iff & \mathsf{false}; \gamma_a \models^{\mathsf{amb}'} C \text{ ce qui est toujours vrai (par la règle d'absurdité)} \\ \iff & (\mathsf{true} \wedge \gamma_a(\tau_1) = \gamma_a(\tau_2)); \gamma_a \models^{\mathsf{amb}'} C \\ \iff & \mathsf{true}; \gamma_a \models^{\mathsf{amb}'} (\tau_1 = \tau_2) \Rightarrow C \end{array}$$

□

En retirant la règle d'absurdité, on obtient un système moins permissif. D'où le corollaire suivant, qui se déduit de la propriété de correspondance entre les sémantiques :

Corollaire 5.2. Pour tout γ_n, environnement E et pour toute contrainte C, si

$$\mathsf{true}; \mathtt{Sing}(E); \mathtt{Sing}(\gamma_n) \models^{\mathsf{amb}} C$$

alors

$$E; \gamma_n \models C$$

On peut spécialiser cet énoncé au cas d'une contrainte C close (sans variable d'inférence libre) :

Corollaire 5.3. Pour tout C, environnement E, si $\mathsf{true}; \mathtt{Sing}(E); \emptyset \models^{\mathsf{amb}} C$ alors $E; \emptyset \models C$

Si la contrainte d'implication n'apparaît pas dans une contrainte, alors κ reste true tout au long de la dérivation. Il n'y a donc pas de possibilité d'utiliser la règle d'absurdité. En utilisant cette remarque et la correspondance entre les deux sémantiques, on peut en déduire le corollaire suivant :

Corollaire 5.4. Pour tout γ_n, environnement E et pour toute contrainte C qui ne contient pas de contrainte d'implication :

$$\mathsf{true}; \mathtt{Sing}(E); \mathtt{Sing}(\gamma_n) \models^{\mathsf{amb}} C \iff E; \gamma_n \models C$$

Chapitre 6

Un solveur pour les contraintes d'hypothèse d'égalité

Une fois définies la syntaxe et la sémantique de cette nouvelle contrainte, nous expliquons comment s'effectue sa résolution dans le solveur, à travers de nouvelles règles de réécriture qui étendent le système de réécriture exposé dans le chapitre 3.

6.1 Préliminaires

6.1.1 Contexte d'hypothèses d'égalités

Pour commencer, nous rajoutons un nouveau contexte :

$$S ::= ... \mid S[\phi : (\tau_1 = \tau_2) \Rightarrow []]$$

On a annoté l'égalité $\tau_1 = \tau_2$ avec un nom ϕ pour pouvoir y faire référence dans les conditions de bord des règles de réécriture, mais aussi pour mieux coller à l'implémentation, car on a besoin de pouvoir manipuler efficacement les égalités. On aura également besoin de se référer à l'ensemble des égalités dans le contexte. Nous définissons donc une opération sur les contextes S :

$$\mathsf{Eqs}([]) = \emptyset \qquad \mathsf{Eqs}(S[[]\wedge]) = \mathsf{Eqs}(S) \qquad \mathsf{Eqs}(S[\exists X.[]]) = \mathsf{Eqs}(S) \qquad \mathsf{Eqs}(S[\forall a.[]] = \mathsf{Eqs}(S)$$

$$\mathsf{Eqs}(S[\mathsf{let}\ x = \lambda X.[]\ \mathsf{in}\ C]) = \mathsf{Eqs}(S) \qquad \mathsf{Eqs}(S[\mathsf{let}\ x = \lambda X.U\ \mathsf{in}\ []]) = \mathsf{Eqs}(S)$$

$$\mathsf{Eqs}(S[\phi : (\tau_1 = \tau_2) \Rightarrow []]) = \mathsf{Eqs}(S), \phi : \tau_1 = \tau_2$$

6.1.2 Ordre des équations

Notons que l'ordre dans lequel les égalités sont introduites est conservé par Eqs. On peut définir, pour les éléments de $\mathsf{Eqs}(S) = \phi_1, \ldots, \phi_n$, un ordre complet :

$$\text{Soient } \phi_i, \phi_j \in \mathsf{Eqs}(S) \text{ alors } \phi_i < \phi_j \iff i > j$$

Ainsi, plus une égalité a été introduite tôt, plus elle est grande (ou âgée). Un contexte d'hypothèses d'égalité peut contenir plusieurs fois la même égalité $\tau_1 = \tau_2$, introduite à différents moments avec des noms différents. De plus, des hypothèses d'égalités introduites peuvent, par transitivité, se combiner pour exprimer de nouvelles égalités. Il sera utile dans la suite de pouvoir différencier, selon leurs âges, les différentes façons de prouver une égalité à partir d'un contexte de typage. L'âge d'une partie de $\mathsf{Eqs}(S)$ est déterminé par l'âge de sa plus jeune équation.

Exemple 6.1.1.

$$\forall ab.(\phi_1 : a = b) \Rightarrow (\phi_2 : b = \texttt{int}) \Rightarrow (\phi_3 : a = \texttt{int}) \Rightarrow \ldots$$

Sous les contextes des typages successifs de cette contrainte, on peut prouver $a = \texttt{int}$ de deux façons : (i) en utilisant la dernière équation introduite ϕ_3, ou (ii) par transitivité avec les deux équations plus anciennes ϕ_1 et ϕ_2. Pour différencier ces deux façons de faire, on peut voir que (i) s'appuie sur un ensemble d'équations (en fait un singleton ici) plus jeune que (ii). ◊

6.1.3 Contourner les limites de la contrainte $\forall a.C$

L'introduction d'égalités entre types est triviale à traiter lorsqu'il s'agit de deux types ground différents : on obtient un contexte incohérent qui permet de tout prouver. Mais quand une égalité introduite implique des variables rigides, le problème du typage est plus intéressant. Il y a, en fait, une forte interaction entre la façon dont nous traitons les variables rigides et la façon dont nous traitons l'introduction d'égalité de types. Nous choisissons de présenter dans un premier temps l'introduction d'égalité de types, et de revenir ensuite plus en détails sur une présentation appropriée des variables rigides.

Cependant, afin de traiter l'introduction d'égalités de types, il nous faut faire une petite digression sur les variables rigides, qui sera d'avantage développée dans un chapitre dédié. L'approche que l'on a eu jusqu'ici pour les variables rigides se basait sur la possibilité, pour différentes occurrences d'une variable rigide, de partager la même structure. Mais le partage n'est plus possible avec l'introduction de types ambivalents : une variable rigide peut être rendue égale à une structure dans un bout du programme, sans l'être dans le reste. En présence d'hypothèse d'égalité, la construction que nous considérions jusqu'ici comme des variables rigides a, en fait, plutôt un comportement de structure. Pour prendre cela en compte, nous modifions le rôle de la contrainte $\forall a.C$, pour qu'elle introduise localement une structure abstraite a (et non plus une variable). Les éléments $a, b, \ldots$ ne dénote plus une catégorie syntaxique pour les variables rigides mais une catégorie de structure.

Nous introduisons une nouvelle grammaire s^a qui regroupe les structures s et les structures abstraites a :

$$s^a ::= s \mid a$$

La sémantique de $\forall a.C$ reste une quantification universelle sur les types ground dans la métalogique (règle FORALL). Les règles de réécriture de cette contrainte sont similaires à celles déjà introduites.

Génération de contrainte pour les annotations de types

Nous devons également modifier la façon dont nous générons des contraintes pour des termes annotés, afin de départager le type inféré à l'intérieur de l'annotation et celui à l'extérieur. Pour générer une contrainte d'annotation de types, on contraint le type de l'expression entière X à être celui de l'annotation τ. En ce qui concerne le terme à l'intérieur de l'annotation t, on le contraint à être d'un type Y, qui est égal à τ, mais qui n'est pas contraint à l'extérieur.

$$[\![(t : \tau) : X]\!] ::= (\!|X \sim \tau|\!) \wedge \exists Y.((\!|Y \sim \tau|\!) \wedge [\![t : Y]\!])$$

Comme nous l'expliquerons plus en détails, quand nous reparlerons des problèmes de partage de structure, ces changements sont nécessaires pour traiter correctement les nouvelles constructions que nous introduisons plus loin dans le chapitre 7.

6.2 Unification

6.2.1 Des multi-équations augmentées avec des ensembles d'égalités de types

Jusqu'ici, les unifications dans la composante U du solveur produisaient un ensemble de multi-équations dans lesquelles au plus un élément était une structure, le reste étant composé des variables d'inférence. Gérer des types ambivalents va nécessiter une approche un peu différente, car

désormais plusieurs structures peuvent être rendues égales. Dans un contexte cohérent, ce ne sont pas n'importe quelles structures qui peuvent être considérées comme égales, mais seulement celles qui sont rendues égales par des égalités introduites dans le contexte de typage. Nous choisissons donc d'adjoindre un ensemble ordonné d'égalités Φ à chaque multi-équations ϵ, et nous noterons $\Phi \vdash \epsilon$.

6.2.2 Choisir les bonnes équations pour unifier des multi-équations

Nous maintiendrons l'invariant que les équations contenues dans Φ apparaissent dans l'ordre dans lequel elles ont été introduites dans la contrainte de départ. Comme il est parfois possible de prouver une égalité de plusieurs façons, il faut déterminer quelle(s) équation(s) utiliser. En sortant de contextes dans lesquels ces équations sont introduites, on risquerait de les faire s'échapper de là où elles sont définies. Mais dans le cas d'une équation redondante, on doit pouvoir sortir du contexte et continuer à résoudre la contrainte : les équations introduites plus tôt suffisent. La bonne façon de choisir les équations à utiliser pour prouver une contrainte consiste à choisir systématiquement les égalités les plus anciennes, puisqu'on peut alors espérer se passer d'équations redondantes introduites plus récemment.

Exemple 6.2.1.

$$\forall ab.(\phi_1 : a = b) \Rightarrow (\phi_2 : b = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge (\phi_3 : a = \texttt{int}) \Rightarrow X \text{ is } \texttt{int}$$

Dans cette contrainte, il n'y a pas besoin de la dernière hypothèse d'égalité ϕ_3 pour prouver X is `int`, puisqu'on peut le déduire des deux premières hypothèses ϕ_1 et ϕ_2. En faisant remonter la contrainte X is `int`, on obtient :

$$\forall ab.(\phi_1 : a = b) \Rightarrow (\phi_2 : b = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int} \wedge (\phi_3 : a = \texttt{int}) \Rightarrow \mathsf{true}$$

On peut maintenant simplifier à nouveau en :

$$\forall ab.(\phi_1 : a = b) \Rightarrow (\phi_2 : b = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int}$$

$\Diamond$

6.2.3 Nouvelles règles pour manipuler les multi-équations

La règle d'introduction d'une contrainte X is τ dans une composante d'unification U reste similaire, car il n'y a pas besoin d'égalités de types pour déduire une équation entre une variable flexible et une structure. La composante Φ de la multi-équation créée est donc vide.

$$S \ ; \ U \ ; \ X \text{ is } \tau \qquad \rightarrow \qquad S \ ; \ U \wedge (\vdash X = \tau) \ ; \ \mathsf{true}$$

Cependant, l'unification des multi-équations est modifiée, puisque l'on peut désormais en unifier deux qui contiennent des structures à priori incompatibles, mais qui sont rendues compatibles par des équations de types. Lorsque l'on unifie deux multi-équations ensemble, il faut désormais produire un ensemble d'équations pour le résultat, qui justifie les égalités entre structures. Une approche en grand pas permettrait de construire directement une nouvelle multi-équation de la forme $\Phi \vdash \epsilon$ où ϵ ne contiendrait qu'au plus une structure. Nous choisissons cependant de garder l'approche en petits pas développée jusque-là, qui correspond ici à permettre aux multi-équations de contenir temporairement plusieurs structures. Ces structures seront par la suite comprimées, pendant que l'on rajoutera éventuellement en parallèle des équations nécessaires pour les prouver dans la composante Φ.

$$(\Phi_1 \vdash X = \epsilon_1) \wedge (\Phi_2 \vdash X = \epsilon_2) \qquad \rightarrow_{\mathsf{Eqs}(S)} \qquad \Phi_1 \cup \Phi_2 \vdash X = \epsilon_1 = \epsilon_2 \qquad \text{(FUSE-AMB)}$$

Il faut garantir que les équations de Φ_1 et Φ_2, qui prouvaient séparément les égalités contenues dans $X = \epsilon_1$ et $X = \epsilon_2$, soient bien contenues dans l'ensemble d'équations résultant. On obtient un ensemble $\Phi_1 \cup \Phi_2$, mais la notation peut-être trompeuse, car cet ensemble doit lui aussi être ordonné : il faut enchevêtrer les équations de façon à ce qu'elles conservent l'ordre d'introduction dans le programme. Cette réécriture dépend donc du contexte S, plus spécifiquement de l'ordre dans lequel des équations y sont introduites.

Exemple 6.2.2.

$$\forall ab.(a = b) \Rightarrow (a = \texttt{int}) \Rightarrow \exists X.(X \text{ is } a \wedge X \text{ is } \texttt{int}) \wedge (X \text{ is } a \wedge X \text{ is } b)$$

À partir de cette contrainte, on peut obtenir l'unification suivante :

$$(a = \texttt{int} \vdash X = a) \wedge (a = b \vdash X = b) \qquad \rightarrow_{a=b, a=\texttt{int}} \qquad a = b,\ a = \texttt{int} \vdash X = a = b$$

$\Diamond$

Il nous faut également des règles qui standardisent les multi-équations. Pour ne garder qu'un seule structure par multi-équation, nous supprimons les autres par réécritures successives. Mais ces structures effacées sont nécessaires à la résolution, et il faut en garder trace. Pour ce faire, nous insérons des égalités dans les Φ, qui justifient l'effacement de certaines structures de ϵ. Bien sûr, il doit s'agir d'égalités introduites dans la contrainte (dans $\mathsf{Eqs}(S)$). Nous choisissons également de prendre les égalités introduites le plus anciennement possible dans $\mathsf{Eqs}(S)$, de façon à ne pas être bloqué dans certaines réécritures quand une partie du contexte est evacué. Pour la même raison, nous écartons les équations superflues.

Dans notre représentation des multi-équations, l'information qu'il y a plusieurs structures égales passe ainsi de la composante ϵ à la composante Φ. Le choix de la structure à garder dans ϵ est donc arbitraire, puisqu'on peut le reconstituer depuis Φ.

Nous définissons $\mathsf{minEqs}(\mathsf{Eqs}(S), \Phi, \tau_1 = \tau_2)$ un sous-ensemble des Φ', égalités de $\mathsf{Eqs}(S)$, qui sont les plus vieilles possibles (il peut y en avoir plusieurs), qui ne contiennent pas d'équations superflues, et tels que $\Phi \cup \Phi' \Rightarrow \tau_1 = \tau_2$.

On écrit $\Phi \Rightarrow s_1^a\ \bar{\tau}_1 = s_2^a\ \bar{\tau}_2$ pour signifier que, dans un contexte d'égalité Φ, on peut prouver l'égalité $s_1^a\ \bar{\tau}_1 = s_2^a\ \bar{\tau}_2$. Pour cela, on peut raisonner par symétrie, réflexivité, transitivité, congruence, injectivité, absurde (si $s_1^a \cdots = s_2^a \ldots$ avec $s_1^a \neq s_2^a$, alors toutes les égalités sont vraies), sur les égalités dans Φ.

Exemple 6.2.3.

$$\begin{aligned}&\mathsf{minEqs}((\phi_1 : a = b,\ \phi_2 : b = \texttt{int},\ \phi_3 : a = \texttt{int}), (\phi_1 : a = b), a = \texttt{int})\\ &= \{(\phi_2 : b = \texttt{int})\}\end{aligned}$$

Pour prouver $a = \texttt{int}$ en sachant que $a = b$, on peut utiliser l'égalité $\phi_2 : b = \texttt{int}$ du contexte, qui est plus âgée que l'égalité $\phi_3 : a = \texttt{int}$. $\Diamond$

Exemple 6.2.4.

$$\begin{aligned}&\mathsf{minEqs}((\phi_1 : a = b,\ \phi_2 : b = \texttt{int},\ \phi_3 : a = \texttt{int}), (\phi_3 : a = \texttt{int}), a = \texttt{int})\\ &= \{\}\end{aligned}$$

Ici on peut prouver $a = \texttt{int}$ avec l'égalité $\phi_3 : a = \texttt{int}$ qui est déjà parmi les égalités de la composante Φ. Il n'y a donc pas besoin de rajouter d'égalités à Φ. Sans ϕ_3 dans la composante Φ, on aurait pu prouver $a = \texttt{int}$ avec $\phi_1 : a = b$ et $\phi_2 : b = \texttt{int}$, qui est un ensemble d'égalité plus vieux, mais on ne peut pas enlever d'égalités de Φ. $\Diamond$

Exemple 6.2.5.

$$\begin{aligned}&\mathsf{minEqs}((\phi_1 : a = b,\ \phi_2 : a = b,\ \phi_3 : b = \texttt{int},\ \phi_4 : a = \texttt{int}),\ \emptyset,\ a = \texttt{int})\\ &= \{(\phi_1 : a = b,\ \phi_3 : b = \texttt{int}), (\phi_2 : a = b,\ \phi_3 : b = \texttt{int})\}\end{aligned}$$

Dans cette exemple, on a deux façons de prouver $a = \texttt{int}$ en utilisant les égalités du contextes : $(\phi_1 : a = b,\ \phi_3 : b = \texttt{int})$ et $(\phi_2 : a = b,\ \phi_3 : b = \texttt{int})$. Ces deux ensembles ont le même âge (puisque leur plus jeune élément est ϕ_3). On peut donc choisir n'importe lequel des deux. ◊

Nous définissons deux règles d'unification de structures, selon qu'elles sont égales ou non :

$$\begin{array}{lll}\Phi \vdash s^a\ \bar{X} = s^a\ \bar{Y} = \epsilon & \rightarrow_{\mathsf{Eqs}(S)} & (\Phi \vdash s^a\ \bar{X} = \epsilon) \wedge (\bar{X}\ \mathsf{is}\ \bar{Y})\\ \Phi \vdash s^a\ \bar{X} = a = \epsilon & \rightarrow_{\mathsf{Eqs}(S)} & (\Phi \cup \Phi' \vdash s^a\ \bar{X} = \epsilon) \wedge (\bar{X}\ \mathsf{is}\ \bar{\tau})\\ & & \text{pour } \Phi' \in \mathsf{minEqs}(S, \Phi, s^a\ \bar{\tau} = a)\end{array}$$

Ces deux règles se recouvrent en partie et le choix de Φ' dans la deuxième règle est non-déterministe, mais ce n'est pas un problème car les différents choix possibles sont équivalents, et le reste du solveur est également non-déterministe.

Enfin, la règle de réécriture (UNIF), qui effectue un pas dans l'unification, doit désormais propager l'ensemble d'équation $\mathsf{Eqs}(S)$ aux règles d'unification :

$$\begin{array}{lll}S\ ;\ U\ ;\ C \rightarrow & S\ ;\ U'\ ;\ C & \text{(UNIF)}\\ & \text{si } U \rightarrow_{\mathsf{Eqs}(S)} U' & \end{array}$$

Dans la suite, pour ne pas alourdir la notation, nous omettrons de préciser l'ensemble $\mathsf{Eqs}(S)$ dans les règles lorsqu'il n'y a pas d'ambiguïtés.

Exemple 6.2.6.

$$a = b,\ a = \texttt{int} \vdash X = a = b \qquad \rightarrow \qquad a = b,\ a = \texttt{int} \vdash X = a$$

On n'a pas eu besoin de rajouter d'équation car celles déjà présentes permettaient de prouver $a = b$. On a supprimé b de la suite d'égalités : ce n'était plus nécessaire de garder cette information qui était déjà contenue dans l'ensemble d'équations. ◊

Exemple 6.2.7.

Prenons la contrainte suivante :

$$\forall ab.(a = b) \Rightarrow (b = \texttt{int}) \Rightarrow (a = \texttt{int}) \Rightarrow \exists X.X\ \mathsf{is}\ a \wedge X\ \mathsf{is}\ \texttt{int}$$

Un bout de la résolution de cette contrainte passe par l'unification suivante :

$$\begin{array}{ll}& (\vdash X = a) \wedge (\vdash X = \texttt{int})\\ \rightarrow & \vdash X = a = \texttt{int}\\ \rightarrow & a = b,\ b = \texttt{int} \vdash X = a\end{array}$$

On choisit les égalités $a = b, b = \texttt{int}$ pour justifier la multi-équation, plutôt que l'égalité $a = \texttt{int}$, qui justifie aussi la multi-équation, mais qui a une portée plus resserrée. ◊

Exemple 6.2.8.

Un autre exemple pour se familiariser avec l'unification entre deux structures :

$$\forall a.\exists XY.Y \text{ is } a \wedge Z \text{ is } \texttt{int} \wedge (\mathsf{list}(a) = \mathsf{option}(\texttt{int})) \Rightarrow \exists X.X \text{ is } \mathsf{list}(Y) \wedge X \text{ is } \mathsf{option}(Z)$$

Durant la résolution de la contrainte, on aura l'unification suivante :

$$\begin{array}{ll} & (\vdash X = \mathsf{list}(Y)) \wedge (\vdash X = \mathsf{option}(Z)) \\ \rightarrow & \vdash X = \mathsf{list}(Y) = \mathsf{option}(Z) \\ \rightarrow & (\mathsf{list}(a) = \mathsf{option}(\texttt{int}) \vdash X = \mathsf{list}(Y)) \wedge (\vdash Y = a) \wedge (\vdash Z = \texttt{int}) \end{array}$$

On a unifié deux structures différentes en rajoutant des unifications à vérifier sur leurs arguments. ◊

En plus de ces nouvelles règles d'unification, il faut également revenir sur les règles (CLASH-1), etc qui ne permettent pas d'ambivalence dans le solveur.

Exemple 6.2.9.

Une des règles d'unification (CLASH-...) nous permettait de conclure que $\vdash X = a = \texttt{int}$ est faux. Mais dans le contexte où a et `int` sont égales, cette multi-équation est cohérente. ◊

Ces règles doivent à présent prendre en compte les équations de types dans le contexte. Il s'agit de cas dans lesquels on ne peut pas appliquer les règles d'unification données un peu plus haut. Comme les variables rigides sont désormais traitées comme de la structure, on peut regrouper toutes ces règles en une seule :

$$\Phi \vdash s_1^a\ \bar{X} = s_2^a\ \bar{Y} = \epsilon \quad \rightarrow \quad \begin{array}{l} \mathsf{false} \\ \text{si } s_1^a \neq s_2^a \wedge \forall \bar{\tau_1}\bar{\tau_2}, \mathsf{Eqs}(S) \not\Rightarrow s_1^a\ \bar{\tau_1} = s_2^a\ \bar{\tau_2} \end{array} \qquad \text{(CLASH-AMB)}$$

6.3 Nouvelles règles de réécriture

6.3.1 Opérations sur les multi-équations

Pour spécifier des règles de réécriture, on a besoin de définir deux opérations :

- $U_{|X}$ qui filtre les multi-équations de U qui contiennent la variable X (on choisit $Y \neq X$) :

$$(\Phi \vdash \epsilon)_{|X} = \begin{cases} \epsilon \text{ si } X \in \epsilon \\ \emptyset \ sinon \end{cases} \qquad (\exists Y.U)_{|X} = U_{|X} \qquad (U_1 \wedge U_2)_{|X} = U_{1|X} \wedge U_{2|X}$$

- $U\#X$ qui fait un test de non appartenance d'une variable X à un ensemble de multi-équations U (on choisit $Y \neq X$) :

$$(\Phi \vdash \epsilon)\#X = X \notin \epsilon \qquad (\exists Y.U)\#X = U\#X \qquad (U_1 \wedge U_2)\#X = (U_1\#X) \wedge (U_2\#X)$$

6.3.2 Règles de réécriture de la contrainte d'hypothèse d'égalité

On peut écrire une première règle qui déplace simplement l'introduction de l'égalité dans le contexte :

PUSH-EQHYP

$$\frac{\mathsf{fresh}\ \phi}{S\ ;\ U\ ;\ (\tau_1 = \tau_2) \Rightarrow C \rightarrow S[\phi : (\tau_1 = \tau_2) \Rightarrow []]\ ;\ U\ ;\ C}$$

Les règles suivantes s'appliquent lorsque l'introduction d'une hypothèse d'égalité se trouve en haut de la pile et qu'on a suffisamment simplifié la contrainte courante.

Faire remonter les variables existentielles unifiée avec une variable plus vieille On définit une règle qui extrude des variables existentielles hors d'une hypothèse d'égalité (cela correspond à faire baisser leur niveau dans une implémentation efficace). Ces variables sont celles qui doivent remonter, car elles sont mises en équations avec des variables plus vieilles.

$$\text{EX-IMPCST}\quad \frac{V \in ftv(\exists X\bar{Y}.U) \qquad X \prec^*_U V}{S[\phi : (\tau_1 = \tau_2) \Rightarrow \exists X\bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true} \rightarrow S[\exists \bar{X}.\phi : (\tau_1 = \tau_2) \Rightarrow \exists \bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true}}$$

Dans la règle, V est une variable liée quelque part dans le contexte S, qui oblige X à remonter avant l'introduction de l'égalité ϕ. Il est possible que X s'échappe de sa portée, ce qui voudrait dire que la contrainte n'était pas satisfiable. Nous rattraperons ce cas grâce à une règle qui sera énoncée plus tard.

Exemple 6.3.1.

Prenons la contrainte :

$$\forall a.\exists V.(a = \mathtt{int}) \Rightarrow \exists XY.Y \ \mathsf{is}\ a \wedge Y \ \mathsf{is}\ \mathtt{int} \wedge X \ \mathsf{is}\ \mathtt{bool} \wedge V \ \mathsf{is}\ X$$

Y ne dépend pas de l'hypothèse d'égalité, on peut donc la vieillir en faisant remonter le lieu où elle est quantifiée. En termes de règle de réécriture, on peut passer de la configuration :

$$\forall a.\exists V.(a = \mathtt{int}) \Rightarrow \exists XY.[] \ ;$$

$$(a = \mathtt{int} \vdash Y = a) \wedge (\vdash X = \mathtt{bool}) \wedge (\vdash V = X) \ ;$$

$$\mathsf{true}$$

à la configuration :

$$\forall a.\exists VY.(a = \mathtt{int}) \Rightarrow \exists X.[] \ ;$$

$$(a = \mathtt{int} \vdash Y = a) \wedge (\vdash X = \mathtt{bool}) \wedge (\vdash V = X) \ ;$$

$$\mathsf{true}$$

grâce à la règle EX-IMPCST. ◊

Une fois simplifiée, résoudre une contrainte d'hypothèse d'égalité Pour résoudre une contrainte let, on simplifiait sa partie gauche, notamment par extrusions de quantifications existentielles (règle LET-ALL). Quand la partie gauche était suffisamment simplifiée, un bout de la contrainte d'unification qui ne faisait intervenir que des variables jeunes pouvait se retrouver à devenir trivial (règle BUILD-SCHEME).

On peut raisonner de façon analogue sur la contrainte d'unification pendant la résolution d'une contrainte d'hypothèse d'égalité. Après extrusions successives de variables, la partie droite de la contrainte devrait être triviale, du moins la sous contrainte qui ne mentionne que les variables introduites localement. On peut alors sortir du contexte d'hypothèse d'égalité.

$$\text{POP-EQ}\quad \frac{U_1 \# \bar{X}, \phi \qquad \big(\mathsf{Eqs}(S), \phi \Rightarrow \exists \bar{X}.U_2\big) \equiv \mathsf{true}}{S[\phi : (\tau_1 = \tau_2) \Rightarrow \exists \bar{X}.[]] \ ; \ U_1 \wedge U_2 \ ; \ \mathsf{true} \rightarrow S \ ; \ U_1 \ ; \ \mathsf{true}}$$

Quand on a appliqué la règle EX-IMPCST autant que nécessaire, il peut rester des variables $\bar{X}$ qui dépendent de l'égalité ϕ. On a une contrainte d'unification U de la forme $U_1 \wedge U_2$ où :

– U_1 est la partie qui dépend de variables extérieures mais pas des variables locales $\bar{X}$ (car $U_1 \# \bar{X}$) qui n'ont pas dû être extrudées. C'est donc un ensemble d'unification que l'on va tenter de résoudre plus tard.

– U_2 est la partie locale qui est vraie quelque soit la valeur des variables extérieures, et que l'on peut oublier ensuite.

On peut remarquer, dans la prémisse, que l'on a potentiellement besoin d'égalités introduites plus haut dans la pile ($\mathsf{Eqs}(S)$) pour résoudre U_2 (pas uniquement l'égalité ϕ). À l'instar d'autres règles de réécriture, le test d'équivalence avec true est facile à réaliser car U_2 est une contrainte d'unification et non pas une contrainte arbitraire.

Exemple 6.3.2.

On peut reprendre un version simplifiée de l'exemple précédent :

$$[][\forall a.[]][\exists Y.[]][(a = \mathtt{int}) \Rightarrow \exists X.[]] \ ; \ (a = \mathtt{int} \vdash X = a) \wedge (\vdash Y = \mathtt{bool}) \ ; \ \mathsf{true}$$

On ne peut pas faire remonter X car elle dépend de l'égalité entre a et $\mathtt{int}$, mais on sait que la multi-équation $(a = \mathtt{int} \vdash X = a)$ est triviale donc on peut l'oublier. De plus, on sait que $\vdash Y = \mathtt{bool}$ est indépendante de l'hypothèse d'égalité, et ne fait pas apparaître de jeune variable, on peut donc effacer le contexte courant de la pile en appliquant la règle POP-EQ :

$$[][\forall a.[]][\exists Y.[]] \ ; \ (\vdash Y = \mathtt{bool}) \ ; \ \mathsf{true}$$

◊

Les pré-conditions de la règle POP-EQ peuvent ne pas être réunies, même après extrusion d'un maximum de quantificateurs existentiels. Il s'agit de cas dans lesquels les contraintes font s'échapper une égalité, et qui ne sont donc pas correctes.

6.3.3 Règle qui détecte l'échappement d'hypothèses d'égalité

Pour ne pas rester bloqué quand les conditions pour appliquer POP-EQ ne sont pas réunies, nous définissons une autre règle de réécriture, qui se réécrit vers false.

Celle-ci s'applique quand une égalité ϕ est nécessaire à prouver une multi-équation, mais que celle-ci n'est pas triviale.

$$\text{SCOPE-ESCAPE}\ \frac{\phi \in \Phi \qquad \big(\mathsf{Eqs}(S), \phi \Rightarrow \exists \bar{X}.\epsilon\big) \not\equiv \mathsf{true}}{S[\phi : (\tau_1 = \tau_2) \Rightarrow \exists \bar{X}.[]] \ ; \ U \wedge (\Phi \vdash \epsilon) \ ; \ \mathsf{true} \rightarrow \mathsf{false}}$$

Ce cas se produit quand une variable vieille se sert d'une égalité introduite récemment.

Exemple 6.3.3.

$$\forall a \exists Y.(a = \mathtt{int}) \Rightarrow Y \ \mathsf{is} \ a \wedge Y \ \mathsf{is} \ \mathtt{int}$$

Cette contrainte n'est pas valide, puisque la variable Y est quantifiée avant l'introduction de l'égalité $a = \mathtt{int}$ qui permet de justifier que Y vaille à la fois a et $\mathtt{int}$. Après quelques réécritures, cette contrainte s'écrit dans notre solveur :

$$[][\forall a \exists Y.[]][\phi : (a = \mathtt{int}) \Rightarrow []] \ ; \ \phi : a = \mathtt{int} \vdash Y = a \ ; \ \mathsf{true}$$

On peut alors appliquer la règle SCOPE-ESCAPE, en prenant la multi-équation $\phi : a = \mathtt{int} \vdash Y = a$. ◊

Exemple 6.3.4.

$$\forall a \exists Y.(a = \texttt{int}) \Rightarrow \exists X.X \text{ is } a \wedge X \text{ is } \texttt{int} \wedge X \text{ is } Y$$

Dans cette contrainte, la variable Y est introduite avant l'hypothèse d'égalité $a = \texttt{int}$. Il n'est donc pas autorisé d'égaliser Y avec X, qui est introduite dans la portée de l'égalité et dont le type est ambivalent. Après réécriture, cette contrainte peut s'exprimer dans le solveur de la forme :

$$[][\forall a \exists Y.[]][\phi : (a = \texttt{int}) \Rightarrow \exists X.[]] \ ; \ \phi : a = \texttt{int} \vdash X = Y = a \ ; \ \mathsf{true}$$

On peut donc bien appliquer la règle SCOPE-ESCAPE. La contrainte $(a = \texttt{int}) \Rightarrow \exists X.X = a = \texttt{int}$ est vraie mais $(a = \texttt{int}) \Rightarrow \exists X.X = Y = a = \texttt{int}$ n'est pas vraie pour toutes les valeurs de Y. Pour résoudre cette contrainte, il faudrait donc pouvoir contraindre la valeur de Y en dehors de la portée de l'égalité, ce qui n'est pas permis. ◊

Pour que notre système de réécriture fonctionne, il faut que, si un contexte d'hypothèse d'égalité se trouve en haut de la pile des contextes, une des règles décrites s'applique. Nous n'avons pas prouvé cette propriété, mais nous n'avons pas trouver de contre-exemple.

6.3.4 Correspondance

On voudrait prouver la correspondance entre la résolution de la contrainte dans notre système et sa sémantique (à l'état de conjecture pour le moment).

Conjecture 6.1. (Correction) Si $S \ ; \ U \ ; \ C \rightarrow S' \ ; \ U' \ ; \ C'$ alors $S[U \wedge C] \equiv S'[U' \wedge C']$

On peut également conjecturer la complétude :

Conjecture 6.2. (Complétude) La configuration $S \ ; \ U \ ; \ C$ possède une chaîne de réécriture vers $[] \ ; \ U' \ ; \ \mathsf{true}$ si et seulement si $\mathsf{true}; \emptyset; \emptyset \models^{\mathsf{amb}} S[U \wedge C]$

Chapitre 7

Variables rigides locales

7.1 Variables rigides et types ambivalents : problème de partage

L'ajout de types ambivalents nous pousse à revoir notre gestion des variables rigides. Une variable rigide peut être rendue égale à une structure dans un bout du programme, sans que cela soit le cas à d'autres endroits. Prenons la fonction suivante :

```
fun (type a) x e ->
  ((x : a), use e : eq(a,int) in (x : a) + 1)
```

Pour le type de retour de cette fonction, on génère une contrainte, qui, une fois simplifiée, est équivalente à :

$$\begin{aligned}\forall a.\exists W V_x V_1 V_2.\quad & W \text{ is } V_1 \times V_2 \wedge \\ & V_1 \text{ is } a \wedge V_x \text{ is } a \wedge \\ & (a = \texttt{int}) \Rightarrow V_2 \text{ is } \texttt{int} \wedge \exists Z.Z \text{ is } a \wedge V_x \text{ is } a \wedge Z \text{ is } \texttt{int}\end{aligned}$$

W est le type de retour de la fonction. V_1 et V_2 sont les types associés aux deux éléments de la paire qui constitue le corps de la fonction. V_x est le type associé à `x`. V_1 et V_x sont de type a pour correspondre à l'annotation sur x dans la partie gauche de la paire. Le reste de la contrainte est défini sous une hypothèse d'égalité $a = \texttt{int}$. V_2 est de type `int` pour correspondre à l'addition dans la partie droite de la paire. La variable d'inférence locale Z nous est utile pour déduire le type de `(x : a)`. Ce type doit être égal à a en raison de l'annotation sur x, mais il doit également être égal à `int` puisqu'il est associé à une opérande d'une addition. Le type de x est contraint de correspondre à l'annotation `(x : a)` (d'où la contrainte V_x is a).

Ainsi, V_1 est de type a uniquement (ou plutôt $\{a\}$ si on raisonne en termes de types ambivalents). Cette partie de la paire, où l'égalité de type n'est pas introduite, n'a pas besoin d'être égalisée à `int`. Ce serait même une erreur, car l'égalité entre a et `int` sortirait de sa portée. Si a était traitée dans le solveur comme une variable, il pourrait être possible, si l'on ne fait pas attention, d'unifier V_1 et Z, en passant de $(V_1 = a) \wedge (Z = a) \wedge (Z = \texttt{int})$ à $(V_1 = Z = a) \wedge (Z = \texttt{int})$, ce qui permettrait d'en conclure que $V_1 = \texttt{int}$.

7.2 Structures abstraites

7.2.1 Départager les différentes occurrences

On est ainsi forcé de distinguer les différentes occurrences des variables rigides, celles-ci ne pouvant pas être traitées comme une seule variable d'inférence. Traiter une variable rigide comme une variable d'inférence ne permet pas de faire ces distinctions. Il faut donc assigner une variable d'inférence par occurrence de variable rigide. Comment signifier au solveur que les différentes variables flexibles pour les différentes occurrences d'une même variable rigide sont égales entre

elles, sans retomber sur le problème précédent d'unification entre variables qui font s'échapper une égalité ?

Nous choisissons de traiter les variables rigides non pas comme des variables mais comme des *structures abstraites*. En effet, les structures ne forcent pas l'unification entre les différentes variables qui leur sont égales.

7.2.2 Une variable flexible différente par occurrence de structure abstraite

Dans la génération de contraintes depuis un programme, on s'est assuré de transformer les types utilisateurs (dont les variables rigides) en petits termes, en générant à la volée de nouvelles variables d'inférence dés que nécessaire. On départage donc bien les différentes occurrences des structures abstraites, c'est-à-dire qu'on ne génère pas, par exemple, de contraintes de la forme :

$$\forall a.\exists X.X \text{ is } a \wedge (a = t) \Rightarrow X \text{ is } a \wedge \dots$$

Notre générateur la mettrait plutôt sous la forme :

$$\forall a.\exists X.X \text{ is } a \wedge (a = t) \Rightarrow \exists Y.Y \text{ is } a \wedge \dots$$

Dans la première forme, on aurait beau traiter les variables rigides comme de la structure, la génération de contrainte pourrait tout de même créer des échappements d'hypothèse d'égalité qui ne sont pas présentes dans le programme source.

7.2.3 Structures abstraites introduites localement, contrainte letr

Jusqu'ici, nous traitions les variables rigides introduites implicitement à top-level. En OCaml, les variables rigides s'introduisent localement avec un `(type a)` (voir le manuel Frisch and Garrigue (2010)) :

```
let f (type a) (foo : a list) = ...
```

Nous définissons une contrainte qui introduit localement de telles structures, et une façon de construire des schémas de types qui les prennent en compte. Nous étendons le langage des contraintes :

$$C ::= \cdots \mid \mathsf{letr}\ x = \forall \bar{a}\lambda X.C_1\ \mathsf{in}\ C_2$$

où $\bar{a}$ sont des structures abstraites introduites localement.

Intuitivement, il s'agit d'une contrainte let plus expressive, puisqu'elle permet de définir des structures abstraites locales en plus d'effectuer le traitement habituel d'une contrainte let. La construction $\forall a.C$ devient une forme particulière de cette contrainte, mêlée à une instanciation : on peut la réécrire $\mathsf{letr}\ x = \forall a\lambda\_.C\ \mathsf{in}\ x\ \_$.

De plus, letr nous permet d'instancier une même contrainte à plusieurs endroits, et nous évite donc une duplication de contraintes.

7.2.4 Génération de contrainte letr

Dans notre langage source, des variables rigides peuvent être introduites par la construction $\forall a.t$. Cela correspond, en OCaml, à introduire la variable `a` avec la construction `fun (type a) -> t`. Lorsque l'on introduit une variable rigide, elle est abstraite tant qu'on reste dans sa portée. C'est pour cela qu'il n'est pas correct d'écrire :

```
fun (type a) ->
  (fun x -> x : a -> a) true
```

Quand on sort de la portée, par contre, on peut instancier `a` avec n'importe quel type :

```
(fun (type a) ->
  (fun x -> x : a -> a))
  true
```

Pour typer un terme qui contient une variable rigide, il faut donc s'assurer qu'elle est utilisée de façon opaque dans sa portée, mais qu'on peut bien l'instancier à l'extérieur. Cela nous donne donc la génération de contrainte suivante :

$$[\![\forall a.t : W]\!] ::= \mathsf{letr}\ x = \forall a \lambda X.[\![t : X]\!]\ \mathsf{in}\ x\ W$$

Dans la partie gauche de la contrainte letr, a est une structure abstraite, qui ne peut pas être unifiée avec de la structure. Dans la partie droite, par contre, on instancie les structures abstraites. Ainsi, si le terme $\forall a.t$ a le type d'une variable W, alors on contraint W à instancier x.

Exemple 7.2.1.

$$\begin{array}{ll} & [\![(\forall a.\lambda x.(x : a))\ \mathsf{true} : W]\!] \\ = & \exists W' Z.Z\ \mathsf{is}\ W' \to W \wedge [\![\forall a.\lambda x.(x : a) : Z]\!] \wedge W'\ \mathsf{is}\ \mathtt{bool} \\ = & \exists W' Z.Z\ \mathsf{is}\ W' \to W \wedge \mathsf{letr}\ x = \forall a \lambda X.[\![\lambda x.(x : a) : X]\!]\ \mathsf{in}\ x\ Z \wedge W'\ \mathsf{is}\ \mathtt{bool} \end{array}$$

En simplifiant on obtient :

$$\mathsf{letr}\ x = \forall a \lambda X.[\![\lambda x.(x : a) : X]\!]\ \mathsf{in}\ x\ (\mathtt{bool} \to W)$$

◊

Reste à expliquer la résolution d'une telle contrainte. Elle repose, à l'instar de la contrainte let classique, sur un mécanisme de généralisation et d'instanciation.

7.3 Généralisation et instanciation avec des structures abstraites

7.3.1 Polymorphisme en présence de structures abstraites

La contrainte letr lie localement des structures abstraites $\bar{a}$. Une fois la partie gauche de la contrainte résolue, il faut donc que le schéma de type correspondant mentionne ces structures et puisse les instancier dans la partie droite.

Comme les types sont traduits en format petits termes, le schéma $\forall a.\ a \to a$ est représenté par un schéma de la forme $\forall a.\lambda X.\exists(X_1\ \mathsf{is}\ a)(X_2\ \mathsf{is}\ a).X\ \mathsf{is}\ X_1 \to X_2$. Comment alors instancier un tel schéma, sachant que celui-ci pourra, en plus, être instancié à plusieurs endroits de façon indépendante ?

À l'instanciation, il faut un mécanisme liant ensemble les différentes variables flexibles qui ont une structure abstraite locale. Le schéma de types est copié, et chaque variable de structure abstraite est remplacée par une même variable d'inférence fraîche. Pour se rappeler quelles structures abstraites sont définies localement et doivent être instanciées, il faut les lister et en garder trace durant la généralisation.

Ainsi, on obtient bien le comportement attendu : les structures sont abstraites à l'intérieur, mais généralisables à l'extérieur.

Exemple 7.3.1.

Dans la contrainte suivante :

$$\mathsf{letr}\ x = \forall a \lambda X.X\ \mathsf{is}\ a \wedge (a = \mathtt{int}) \Rightarrow \exists Y.Y\ \mathsf{is}\ a \wedge Y\ \mathsf{is}\ \mathtt{int}\ \mathsf{in}\ x\ W$$

a est abstraite tant que l'on cherche à résoudre la contrainte dans la partie gauche, mais devient généralisable à l'extérieur, et on peut instancier le schéma complet (avec $x\ W$ par exemple, pour une variable d'inférence W). ◊

7.3.2 Solveur pour les structures abstraites

Les règles de réécriture des contraintes letr sont similaires à celles des contraintes let. Il faut cependant se pencher sur la façon dont sont traitées les structures abstraites locales.

Nous commençons par définir des nouveaux contextes de réécriture :

$$S ::= \cdots \mid S[\mathsf{letr}\ x = \forall \bar{a}\lambda X.[]\ \mathsf{in}\ C] \mid S[\mathsf{letr}\ x = \forall \bar{a}\lambda X.C\ \mathsf{in}\ []]$$

On ne s'attardera pas sur les règles d'extrusion de quantificateurs existentiels qui correspondent à celles s'appliquant aux contraintes let. Nous définissons une règle qui construit un schéma de types avec des structures abstraites locales :

BUILD-RIGID-SCHEME

$$\frac{X\bar{Y}\#ftv(U_1) \wedge a\#U_1 \wedge \forall \bar{a}\exists X\bar{Y}.U_2 \equiv \mathsf{true}}{S[\mathsf{letr}\ x = \forall \bar{a}\lambda X.\exists \bar{Y}.[]\ \mathsf{in}\ C]\ ;\ U_1 \wedge U_2\ ;\ \mathsf{true} \rightarrow S[\mathsf{letr}\ x = \forall \bar{a}\lambda X.\exists \bar{Y}.U_2\ \mathsf{in}\ []]\ ;\ U_1\ ;\ C}$$

On remarque que les structures abstraites locales sont présentes dans la partie gauche de la contrainte une fois résolue (letr $x = \forall \bar{a}\lambda X.\exists \bar{Y}.U_2$ in []).

La condition de bord $\forall \bar{a}\exists X\bar{Y} \equiv \mathsf{true}$ assure que l'on peut bien garder $\bar{a}$ abstraites pour résoudre U_2. Notons que la quantification universelle $\forall \bar{a}$ se trouve au début de cette contrainte. En effet, les variables $X, \bar{Y}$ doivent pouvoir être unifiées avec des structures qui contiennent $\bar{a}$.

Exemple 7.3.2.

$$\begin{array}{ll} & S[\mathsf{letr}\ x = \forall a\lambda X.[]\ \mathsf{in}\ \mathsf{true}]\ ;\ (a = \mathtt{int}) \Rightarrow X\ \mathsf{is}\ a \rightarrow a \wedge \exists Y.Y\ \mathsf{is}\ a \wedge Y\ \mathsf{is}\ \mathtt{int}\ ;\ \mathsf{true} \\ \rightarrow & S[\mathsf{letr}\ x = \forall a\lambda X.[]\ \mathsf{in}\ \mathsf{true}]\ ;\ X\ \mathsf{is}\ a \rightarrow a\ ;\ \mathsf{true} \\ \rightarrow & S[\mathsf{letr}\ x = \forall a\lambda X.X\ \mathsf{is}\ a \rightarrow a\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ \mathsf{true} \end{array}$$

◊

Au moment de l'instanciation du schéma de types, il faut être en mesure de trouver des témoins pour les structures abstraites locales. Les éventuelles structures abstraites plus anciennes contenues dans le schéma, elles, doivent bien rester abstraites.

Exemple 7.3.3.

Prenons le programme suivant, qui contient une expression `let` imbriquée dans une autre :

```
let f (type a) x =
  let g () = (x : a) in
  (g () : int, g () : bool)
```

Si on instanciait toutes les structures abstraites, et pas juste les structures locales au letr, alors le type de `g` (c'est-à-dire `unit` $\rightarrow a$) pourrait être instancié à la fois par `unit` $\rightarrow$ `int` et `unit` $\rightarrow$ `bool`. ◊

Au moment de l'instanciation, on substitue, dans le schéma, chaque structure abstraite localement par une variable flexible fraîche :

$$\begin{array}{rl} S\ ;\ U\ ;\ x\ W & \rightarrow S\ ;\ U\ ;\ \exists \bar{Y}.C[\bar{Y}/\bar{a}][W/X] \\ & \text{si } S(x) = \forall \bar{a}\lambda X.C \end{array}$$

Pour instancier x, W doit pouvoir être unifiée avec le schéma de type induit par $\forall \bar{a}\lambda X.C$. Cela correspond à essayer de résoudre la contrainte obtenue après substitution de $\bar{a}$ par des variables fraîches $\bar{Y}$, d'une part, et substitution de X par W d'autre part.

Exemple 7.3.4.

$S[\mathsf{letr}\ x = \forall a\lambda X.[]\ \mathsf{in}\ x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}]$;
$(a = \mathtt{int}) \Rightarrow X\ \mathsf{is}\ a \to a \wedge \exists Y.Y\ \mathsf{is}\ a \wedge Y\ \mathsf{is}\ \mathtt{int}$;
true

$\to$ $S[\mathsf{letr}\ x = \forall a\lambda X.[]\ \mathsf{in}\ x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}]$;
$X\ \mathsf{is}\ a \to a$;
true

$\to$ $S[\mathsf{letr}\ x = \forall a\lambda X.X\ \mathsf{is}\ a \to a\ \mathsf{in}\ []]$;
true ;
$x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}$

$\to^*$ $S[\mathsf{letr}\ x = \forall a\lambda X.X\ \mathsf{is}\ a \to a\ \mathsf{in}\ []]$;
true ;
$\exists W'.W\ \mathsf{is}\ W' \to W' \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge \exists V'V\ \mathsf{is}\ V' \to V' \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}$

$\to$ ...

Ici, on voit que les deux instanciations sont faites avec des variables d'inférence différentes (et indépendantes entre elles). Cela permet d'instancier le schéma $\forall a.a \to a$ avec les types $\mathtt{int} \to \mathtt{int}$ et $\mathtt{bool} \to \mathtt{bool}$ et de donc de typer des programmes comme celui-ci :

```
type 'a val =
| Nat of int * ('a,int) eq
| Bool of bool * ('a,bool) eq

let eval (type a) (t : a val) =
  match t with
  | Nat (v, e) -> let v = (v : int) in use e : eq a int in (v : a)
  | Bool (v, e) -> let v = (v : bool) in use e : eq a bool in (v : a)
```

$\Diamond$

7.4 Sémantique de la contrainte letr

7.4.1 Sémantique comme contrainte à part entière

Nous pouvons définir une sémantique, pour letr et l'instanciation, qui sont assez déclaratives, puisqu'on ne donne pas explicitement une façon d'obtenir de schéma de type, mais plutôt une contrainte qui induit un schéma. La version ambivalente est similaire.

$$\frac{\kappa, E, \gamma \models \forall \bar{a} \exists X.C_1 \qquad \kappa, E[x \mapsto \forall \bar{a}\lambda X.C_1], \gamma \models C_2}{\kappa, E, \gamma \models \mathsf{letr}\ x = \forall \bar{a}\lambda X.C_1\ \mathsf{in}\ C_2}$$

Si on peut résoudre C_1, dans un environnement de typage contenant des témoins pour $\bar{a}$ et X, on a vérifié qu'il existait une solution pour la partie gauche du letr. Il reste alors à trouver une solution pour la partie droite. C'est ce qui est fait dans la dernière prémisse : on essaye de prouver C_2 avec un environnement polymorphe dans lequel on attribue un schéma à x.

Pour trouver une solution pour la contrainte d'instanciation $x\ W$, il faut, en plus de l'instanciation classique d'un schéma par un type, trouver comment instancier les structures abstraites dans le schéma associé à x.

$$\frac{E(x) = \forall \bar{a}\lambda X.C \qquad \kappa, E, \gamma \models \exists \bar{Y}.C[W/X][\bar{Y}/\bar{a}]}{\kappa, E, \gamma \models x\ W}$$

On remplace les occurrences de chaque structure abstraite a dans C par une variable fraîche Y.

7.4.2 Décomposition de la contrainte letr

La sémantique de la contrainte let pouvait être obtenue par dé-sucrage et décomposition en deux contraintes plus simples :

$$\mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2 \quad \equiv \quad \exists X.C_1 \wedge \mathsf{def}\ x : \lambda X.C_1\ \mathsf{in}\ C_2$$

On peut également définir la contrainte letr par expansion, même si l'implémentation du solveur n'en tient pas compte.

Notons que traiter directement letr comme une contrainte à part entière, et non comme du sucre syntaxique, est non seulement mieux compris, mais épargne en outre une duplication de contraintes.

Une façon de comprendre la contrainte letr est de la décomposer en deux composantes sous la forme :

$$\mathsf{letr}\ x = \forall \bar{a}\lambda X.C_1\ \mathsf{in}\ C_2 \quad \equiv \quad \forall \bar{a}.\exists X.C_1 \wedge C_2[x\ Y \backslash \exists \bar{a}.C_1[X \backslash Y]]$$

L'idée est de garder les $\bar{a}$ comme des structures abstraites dans C_1 et de les instancier dans C_2. Il s'agit d'une décomposition similaire à celle d'un let classique, mais avec des structures abstraites en plus. De même, les contraintes $\forall a.C$ et $\exists a.C$ sont similaires aux contraintes $\forall X.C$ et $\exists X.C$. La sémantique de $\exists a.C$ est donnée ci-dessous :

$$\frac{\exists t, \quad \kappa; E; \gamma[a \mapsto t] \models C}{\kappa; E; \gamma \models \exists a.C}$$

Exemple 7.4.1.

$$\begin{array}{l}[\![\forall a.\lambda xy.(x : a, y : a) : W]\!] \\ = \forall a \exists Z.[\![\lambda xy.(x : a, y : a) : Z]\!] \wedge \exists a[\![\lambda xy.(x : a, y : a) : W]\!]\end{array}$$

La partie gauche de la contrainte peut s'écrire en forme grand terme, après simplification :

$$\exists a \exists XY X'Y'.W\ \mathsf{is}\ X \to Y \to X' \times Y' \wedge X\ \mathsf{is}\ X' \wedge Y\ \mathsf{is}\ Y' \wedge X'\ \mathsf{is}\ a \wedge Y'\ \mathsf{is}\ a$$

c'est-à-dire :

$$\exists a \exists XY.W\ \mathsf{is}\ X \to Y \to X \times Y \wedge X\ \mathsf{is}\ a \wedge Y\ \mathsf{is}\ a$$

Les deux occurrences de a dans l'annotation $(x : a, y : a)$, qui correspondent à X is a et Y is a dans la contrainte, sont bien indépendantes.

$\Diamond$

Cette sémantique du let avec $\exists a.C$ est plutôt simple et se prête bien à une spécification, mais entraîne de la duplication. De plus, on ne sait pas implémenter facilement un solveur pour la contrainte $\exists a.C$. La contrainte $\forall a.C$ introduisait une rigide qui pouvait ensuite être égalisée à un type lors de l'inférence du type d'un GADT. Une fois cette égalité utilisée, on savait qu'il fallait augmenter la portée de la variable a. Mais dans le cas de la contrainte $\exists a.C$, on ne sait pas, à priori, si unifier a avec un type nécessite de recourir à un égalité du contexte (qui impose d'augmenter la portée) ou s'il s'agit juste d'une unification pour établir la vraie valeur de a.

Notre implémentation de la contrainte letr ne se base donc pas sur la décomposition en deux parties $\forall \bar{a} \ldots \exists \bar{a} \ldots$ mais plutôt sur les aspects décrits plus tôt dans ce chapitre.

Chapitre 8

Implémentation du solveur

8.1 Passer du système de réécriture au solveur Inferno

Le système de réécriture présenté plus haut donne une spécification de haut niveau de la résolution de contraintes, qui a vocation à être implémentée dans un programme "solveur" de contraintes. Il faut notamment retranscrire les mécanismes de passage au contexte de la composante S et implémenté la composante d'unification U avec une structure de données adaptée, ce a quoi s'est attelé François Pottier lorsqu'il a designé Inferno.

Dans Inferno, chaque variable existentielle introduite se voit assigner un *niveau* qui correspond au placement de son quantificateur dans S. On peut choisir un niveau de De Bruijn qui s'incrémente à chaque fois qu'on rentre dans un let, chaque égalité introduite et à chaque quantification universelle. La composante U, quant à elle, est implémentée par une structure d'union-find (voir Tarjan (1975)), qui est une structure de données souvent plébiscitée pour les problèmes d'unification.

Il peut être intéressant de regarder comment cela se traduit sur un exemple. Prenons le terme $\lambda k.\ \mathsf{let}\ f = \lambda x.\lambda y.\ k\ y\ \mathsf{in}\ ()$. Si la variable W est la variable d'inférence du terme entier, la contrainte générée (simplifiée par soucis de compréhension) est :

$$
\begin{array}{l}
\exists K K'.\ W\ \mathsf{is}\ K \to K' \wedge \\
\mathsf{let}\ k = \lambda V.\ V\ \mathsf{is}\ K\ \mathsf{in} \\
\mathsf{let}\ f = \\
\quad \lambda U. \exists X Y Z.\ U\ \mathsf{is}\ X \to Y \to Z\ \wedge \\
\quad \mathsf{let}\ x = \lambda X'.\ X'\ \mathsf{is}\ X\ \mathsf{in} \\
\quad \mathsf{let}\ y = \lambda Y'.\ Y'\ \mathsf{is}\ Y\ \mathsf{in} \\
\quad \exists Z'\ Z''.\ Z''\ \mathsf{is}\ Z' \to Z \wedge k\ Z'' \wedge y\ Z' \\
\mathsf{in} \\
K'\ \mathsf{is}\ \mathtt{unit}
\end{array}
$$

Dans le solveur Inferno, les variables d'inférences se voient assigner un niveau et une structure (qui peut être inconnue). Par convention W a le niveau 0 et pour le reste des variables, le compte commence à 1. Ainsi K, K' ont le niveau 1, X, Y, Z le niveau 3 et Z', Z'' le niveau 5.

Dans la contrainte, on remarque que y est instanciée avec Z' tandis que x n'est jamais instanciée.

Intéressons nous aux unifications qui se produisent lors de la résolution de la contrainte $Z''\ \mathsf{is}\ Z' \to Z \wedge k\ Z'' \wedge y\ Z'$.

À ce moment de la résolution, l'état du solveur Inferno peut être représenté ainsi :

Variable	Niveau	Structure
W	0	$K \to K'$
K	1	None
K'	1	None
U	3	$X \to Y \to Z$
X	3	None
Y	3	None
Z	3	None
Z'	5	None
Z"	5	None

Dans le solveur par réécriture décrit plus haut, on traite un triplet de la forme :

$$S \ ; \ U \ ; \ Z'' \ \mathsf{is} \ Z' \to Z \wedge k \ Z'' \wedge y \ Z'$$

pour un certain S et un certain U, issus des réécritures successives de la contrainte initiale. On réécrit ce triplet en

$$S[[] \wedge k \ Z'' \wedge y \ Z'] \ ; \ U \ ; \ Z'' \ \mathsf{is} \ Z' \to Z$$

puis en

$$S[[] \wedge y \ Z'] \ ; \ U \wedge Z'' = Z' \to Z \ ; \ k \ Z''$$

On procède à l'unification de Z'' avec $Z' \to Z$. L'état du solveur Inferno devient :

Variable	Niveau	Structure
W	0	$K \to K'$
K	1	None
K'	1	None
U	3	$X \to Y \to Z$
X	3	None
Y	3	None
Z	3	None
Z'	5	None
Z"	5	$\mathbf{Z'} \to \mathbf{Z}$

Puis on procède à l'instanciation de k par Z'' :

$$S \ ; \ U \wedge Z'' = Z' \to Z \wedge K = Z'' \ ; \ y \ Z'$$

Cela se traduit dans l'état du solveur Inferno par l'unification des variables K et Z''. Le niveau de la classe d'équivalence est le niveau minimum des deux variables :

Variable	Niveau	Structure
W	0	$K \to K'$
K, **Z"**	1	$\mathbf{Z'} \to \mathbf{Z}$
K'	1	None
U	3	$X \to Y \to Z$
X	3	None
Y	3	None
Z	3	None
Z'	5	None

Vient ensuite la résolution de l'instanciation de y par Z'. Dans le solveur par réécriture, on ajoute l'unification de Y avec Z' :

$$S \; ; \; U \wedge Z'' = Z' \to Z \wedge K = Z'' \wedge Y = Z' \; ; \; \mathsf{true}$$

ce qui se traduit dans le solveur Inferno par le changement suivant :

Variable	Niveau	Structure
W	0	$K \to K'$
K, Z"	1	$Z' \to Z$
K'	1	None
U	3	$X \to Y \to Z$
X	3	None
Y, **Z'**	3	None
Z	3	None

Le solveur Inferno entame alors une phase de *généralisation* des variables du let : il propage les niveaux des variables vers les feuilles de leurs structures, puis crée un schéma polymorphe. Les variables dont le niveau n'a pas baissé sont dites *génériques*. Ici, Z' est de niveau 3, mais elle est la feuille de la structure $Z' \to Z$ de la variable Z'', qui est de niveau 1. Idem pour Z. On met donc à jour leur niveau :

Variable	Niveau	Structure
W	0	$K \to K'$
K, Z"	1	$Z' \to Z$
K'	1	None
Y, Z'	**1**	**None**
Z	**1**	**None**
U	3	$X \to Y \to Z$
X	3	None

8.2 Garder trace de l'introduction d'équations de types avec des niveaux et portées

8.2.1 Une portée par égalité introduite

Quand on implémente un solveur, on a besoin de garantir qu'une égalité ϕ ne s'échappe pas de la zone dans laquelle elle est définie – dans le cas contraire on peut appliquer la règle SCOPE-ESCAPE qui se réécrit vers false. Cette gestion des noms d'égalités n'est pas évidente à implémenter efficacement.

Une idée inspirée d'OCaml est de représenter les lieux où les noms d'égalités sont définis par des entiers, que l'on appelle des portées. Il s'agit en fait de niveaux de De Bruijn: l'égalité ϕ la plus ancienne dans le contexte a la portée 0, la suivante a la portée 1, etc. La portée détermine la partie de la contrainte dans laquelle on est autorisé à utiliser cette égalité. On peut ainsi définir la portée d'une multi-équation comme le maximum des portées des égalités utilisées pour justifier sa cohérence.

Exemple 8.2.1.

Prenons le programme suivant :

```
fun (type a b) e1 e2 ->
  use e1 : eq a b in
  use e2 : eq b int in
  fun x -> (x : a, x + 1)
```

L'égalité `e1` a la portée 1, et `e2` la portée 2. Ici tout se passe bien, car l'expression `fun x -> (x : a, x + 1)`, qui utilise les deux égalités, se trouve dans les deux portées à la fois. Mais si on modifie le programme en :

```
fun (type a b) e1 e2 ->
  use e1 : eq a b in
  fun x ->
    use e2 : eq b int in
    (x : a, x + 1)
```

le programme ne type plus. L'égalité `e2` s'échappe de sa portée, puisque `x` est définie avant son introduction, hors de la zone de portée 2. ◊

8.2.2 Quand faut-il rejeter une contrainte ? Lien entre niveau et portée

Pour détecter les échappements d'égalité, on impose une discipline sur les niveaux et les portées. Une variable de niveau n peut être définie par une multi-équation de portée au plus n.

Exemple 8.2.2.

Prenons un bout de la contrainte générée par le programme donnée plus haut :

$$\forall ab.(\phi_1 : \ a \ = \ b) \Rightarrow \exists X_f XY.\ X_f \ \mathsf{is}\ X \to Y \wedge (\phi_2 : \ b \ = \ \texttt{int}) \Rightarrow \exists Z. \ldots$$

Ici les variables X_f, X, Y sont de niveau 0 et Y de niveau 1. Seule Y peut utiliser l'égalité ϕ_2 dont la portée est 1, mais toutes peuvent utiliser l'égalité ϕ_1 de portée 0. ◊

Quand on unifie des variables d'inférence, on fait l'union de leurs multi-équations, et la cohérence du résultat peut dépendre d'égalités $\phi : a = \tau$ qui n'étaient pas nécessaires pour les multi-équations avant unification (cf 6.2). On met alors à jour une nouvelle portée maximum pour la multi-équation résultante. Cela peut donner une portée maximum plus grande que les portées maximum précédentes.

Exemple 8.2.3.

En reprenant un exemple simple, on peut dérouler l'unification des multi-équations et s'intéresser à leurs portées.

$$\forall a.(\phi : \ a \ = \ \texttt{int}) \Rightarrow X \ \mathsf{is}\ a \wedge X \ \mathsf{is}\ \texttt{int}$$

On obtient dans un premier temps deux multi-équations indépendantes, $\vdash X = a$ et $\vdash X = \texttt{int}$. Ces deux multi-équations ont une portée de 0 : elles ne reposent sur aucune équation de types et sont cohérentes dans toute la zone de la contrainte dans laquelle a a été introduite. Pour unifier ces deux multi-équations, il faut utiliser l'égalité ϕ, qui réduit la zone dans laquelle la multi-équation est cohérente : $\phi \vdash X = a$. La portée de la multi-équation devient 1. ◊

Autrement dit, l'unification de multi-équations peut faire augmenter leur portée (jamais la diminuer), ce qui restreint la zone de code dans laquelle le type est bien formé. Si cette nouvelle portée est strictement supérieure au niveau des variables flexibles de la multi-équation, le solveur échoue : on essaye d'unifier une variable vieille avec une variable jeune dont la validité repose sur un équation qui va sortir du contexte. Cela correspond exactement à la condition d'échappement des égalités dans la règle SCOPE-ESCAPE, mais retranscrite avec des niveaux et des portées.

8.3 Gestion des égalités de types avec un graphe

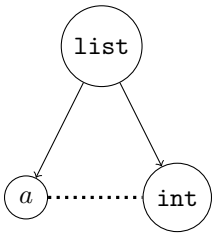
8.3.1 Stocker des égalités de types dans un graphe

On l'a vu, le solveur garde trace des égalités de type introduites dans sa composante S et dans les multi-équations manipulées dans sa composante U. Il a donc fallu trouver une manière de représenter efficacement les hypothèses d'égalité dans le contexte et savoir lesquelles sont nécessaires pour unifier des multi-équations. De plus, nous devons stocker leur portée, afin d'évaluer si un type ambivalent est bien formé (i.e. qu'il est cohérent avec des égalités introduites auparavant).

Pour cela, nous utilisons une structure de graphe dont les noeuds sont des structures, et les arêtes représentent des égalités. Les noeuds en eux-mêmes peuvent être arborescents puisqu'ils représentent des types arborescents. De plus, cette arborescence peut contenir des structures qui sont égales à d'autres structures : il faut propager les égalités aux feuilles. De plus, chaque noeud structurel est unique. Cela permet de tester facilement l'égalité entre différentes structures.

Exemple 8.3.1.

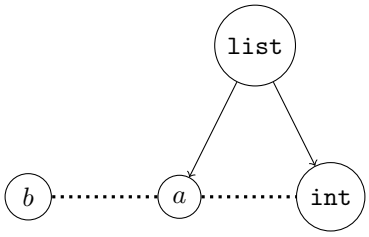
Si on veut représenter l'égalité `list` a = `list int`, il faut qu'une arête relie le noeud a avec le noeud `int`.



◊

Exemple 8.3.2.

Si on rajoute à l'exemple précédent l'égalité $a = b$, il faut ajouter une arête entre a et b, et non pas dupliquer un noeud a. De cette façon, on pourra tester rapidement l'égalité entre b et `int` en explorant le graphe depuis le noeud b. On peut également tester rapidement l'égalité entre `list` b et `list int` par un parcours depuis le noeud `int`.



◊

```
type vertex =
  structure

and structure =
  S of vertex Structure.structure [@@unboxed]
```

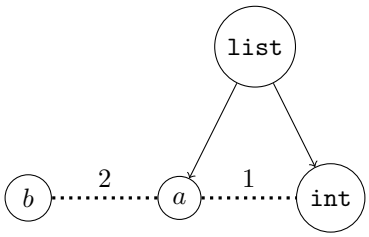
On définit ici un constructeur `S` avec l'annotation `@@unboxed` car OCaml ne permet pas d'écrire des abréviations de type cycliques.

8.3.2 Garder trace de la portée des égalités

D'une certaine façon, les différents noeuds égaux dans le graphe sont interchangeables. Mais les hypothèses d'égalité étant introduites à différents endroits du programme, elles n'ont pas toutes la même portée. Chaque arête doit donc fournir une information sur la portée de l'égalité qu'elle représente. Cette portée sera utilisée lors de l'unification de multi-équations. En effet, il faudra mettre à jour la portée de la multi-équation résultante, en tenant compte des portées des équations nécessaires pour assurer la cohérence de ses équations. Lors des parcours de graphe, il faudra donc garder en mémoire la plus grande portée des arêtes traversées.

Exemple 8.3.3.

Si l'égalité `list` a = `list int` est de portée 1 et $a = b$ de portée 2, on peut représenter ces égalités par le graphe :



Pour savoir, à partir de ce graphe, si on peut supposer une égalité entre b et `int`, il suffit de parcourir les noeuds en partant de b, en gardant trace des portée parcourues. Ici, on trouvera qu'il faut emprunter un chemin sur lequel la plus grande portée est 2. ◊

Dans l'implémentation, on peut représenter le graphe des égalités avec le type suivant :

```
type equalities_graph = (vertex, (vertex * scope) option) HashTabl.t
```

Ainsi, dès qu'une égalité est introduite, on rajoute au graphe les noeuds qui n'existaient pas, en stockant une arête entre eux dans la table de hachage, en spécifiant la portée courante. On stocke en fait deux arêtes entre chaque noeud, afin de faciliter les parcours.

Choisir un graphe plutôt qu'une structure d'union-find

À première vue, le graphe n'apparaît pas comme la solution la plus efficace, puisqu'une comparaison avec un type nécessite de se comparer à toute sa classe d'équivalence, ce qui implique de parcourir toute sa composante connexe, en temps linéaire, là où une structure d'union-find permet une complexité amortie constante. Cependant, nous devons garder trace de la portée des égalités, ce qui n'est pas du tout évident avec une structure d'union-find, puisque l'algorithme d'unification sous-jacent n'est pas prévu pour prendre en compte des informations sur des arêtes entre deux noeuds. Par ailleurs, nous trouvons raisonnable de supposer que le nombre d'égalités introduites, et donc la taille des composantes connexes, reste petit et qu'un algorithme en temps linéaire sur le graphe s'effectue dans les faits en temps constant.

8.3.3 Modification du graphe à la sortie d'un contexte d'hypothèse d'égalité

Pour que le graphe d'égalités ne stocke que les égalités qui sont dans le contexte courant de typage, il faut être en mesure de retirer les arêtes qui représentent des égalités hors de portée. Le bon moment pour retirer de telles arêtes est la sortie d'un contexte d'hypothèse d'égalité (matérialisé par la règle POP-EQ dans le système de réécriture). Il s'agit donc de retirer toutes les arêtes d'une portée donnée. Afin de ne pas avoir à parcourir l'ensemble du graphe pour les enlever, on garde en mémoire les arêtes introduites dans le graphe en les stockant dans une pile, et en les poussant au fur et à mesure qu'on les ajoute au graphe. Retirer toutes les arêtes de la portée courante revient donc à retirer une par une les arêtes de la pile, jusqu'à ce qu'on rencontre une arête de portée plus petite (qui aura été introduite avant que l'on rentre dans le contexte duquel on sort).

On représente ainsi la pile d'arêtes :

```
type edges_stack = (scope * vertex * vertex) Stack.t
```

L'environnement des égalités de types est composé du graphe des égalités ainsi que de la pile :

```
type eqenv = { table: equalities_graph ; added_edges : edges_stack}
```

8.4 Comparaison avec OCaml

Chapitre 9

Travaux liés

Travail de François Pottier et Didier Rémy sur l'inférence de type pour ML

L'approche décrite dans *The Essence of ML Type Inference* (Pottier and Rémy (2005)) nous a fourni une base pour le système de réécriture avec ses trois composantes (S, U et C) décrite dans le chapitre 3. Ils y décrivent des règles de réécriture, que nous avons repris, ainsi que des façons de raisonner sur ces règles, qui nous ont été bien utiles pour étendre ce système. Nous nous sommes initialement inspirés de la façon dont sont traitées les variables rigides dans ce travail, avant de changer pour la présentation décrite dans le chapitre 7.

Travail de François Pottier sur la bibliothèque Inferno

L'article de François Pottier qui introduit les principes de fonctionnement d'Inferno (Pottier (2014)) nous a bien sûr beaucoup aider pour la prise en main de la bibliothèque. En outre, cet article détaille un langage de contraintes ainsi que la génération de contraintes pour un noyau ML, qui nous a fourni une bonne base pour la formalisation décrite dans ce manuscrit. L'approche décrite en annexe, d'une sémantique avec valeur pour les contraintes, nous a aussi aiguillé lorsqu'il a fallut décrire notre sémantique, même si nous avons laissé de côté, par manque de temps, les valeurs sémantiques présentes dans ses jugements, qui servent à l'élaboration.

Travail de Jacques Garrigue et Didier Rémy sur l'inférence des GADTs

Comme indiqué dans la subsection 5.3.1, notre approche pour l'inférence suit celle développée dans Garrigue and Rémy (2013), basée sur la notion de types ambivalents. Notre but était de rejeter et d'accepter les mêmes programmes que leur travail.

Thèse de Master d'Alistair O'Brien

Le travail d'Alistair O'Brien, qu'il a effectué en parallèle du mien, porte sur l'inférence de types par résolution de contraintes, pour un sous-ensemble d'OCaml (O'Brien (2022)). Plutôt que de partir d'une bibliothèque comme Inferno, il a décidé de programmer son propre moteur d'inférence, et a rapidement réussit à prendre en charge une partie importante d'OCaml. Sa thèse de Master est donc fortement liée à ce que nous avons entrepris de notre côté.

Il y développe en annexe une sémantique ambivalente avec une contrainte d'implication. Après discussions avec lui, nous avons abouti à une simplification de notre sémantique.

Notre langage de contraintes est similaire au sien, mais comporte quelques différences. Il définit notamment deux contraintes existentielles différentes, une classique et une ambivalente, là où notre langage n'en comporte qu'une seule. Son langage comporte une contrainte d'égalité ainsi qu'une contrainte de "sous-ensemble", là où le notre englobe ces deux contraintes en une seule contrainte is. Notre choix est motivé par la preuve de correspondance entre les deux sémantiques que nous avons décrite dans le manuscrit. Une autre différence entre nos sémantique vient de la gestion des égalités introduites. Dans son travail, Alistair O'Brien prend en compte dans son jugement sémantique un environnement pour les égalités introduite, là où notre jugement ne comporte qu'un

bit d'information (la composante κ) qui renseigne sur la cohérence ou l'incohérence du contexte de typage.

Nous avons, de notre côté, pu pousser un peu plus la formalisation d'un solveur en donnant un système de réécriture, qui n'est pas donné dans sa thèse de Master.

Travaux de Thomas Réfis

Pour améliorer le typeur OCaml, Thomas Réfis y a intégré la notion de portée pour détecter l'ambivalence (Refis (2018)). Le système que l'on a décrit plus haut et que l'on a rajouté à Inferno est donc inspiré de ce qu'il a implémenté. Dans son travail figure aussi l'idée de traiter les variables rigides comme de la structure abstraite.

Travail de Vytiniotis, Peyton Jones, Schrijvers et Sulzmann sur l'inférence des GADTs

Les GADTs sont également une fonctionnalité importante du langage Haskell, dans lequel les types sont inférés. Des chercheurs ont développé un système nommé OUTSIDEIN(X) (Vytiniotis, Jones, Schrijvers, and Sulzmann (2011)) qui décrit l'inférence de type avec des contraintes pour ce langage. Ils ont pu appliquer ce système à des fonctionnalités avancées comme les GADTs. Une différence notable entre cette approche et celle de Pottier et Remy réside dans la gestion des `let` internes, puisque ceux-ci ne sont pas implicitement généralisés dans OUTSIDEIN(X). Le système de contrainte y est plus riche, pour implémenter des fonctionnalités de Haskell comme les *types classes* et les *types families*. Nous avons cependant fait le choix de partir du travail de Garrigue et Rémy sur les GADTs et les types ambivalents pour coller mieux au typage de OCaml.

Travail de Zhao, Oliveira et Schrijvers

La mécanisation de la méta-théorie de l'inférence de type à l'aide d'assistant de preuve est également une approche intéressante. Dans leur article Zhao, Oliveira, and Schrijvers (2019), les auteurs présentent une formalisation du système de type de Dunfield et Krishnaswami, avec l'assistant de preuve Abella. Ce système permet du polymorphisme de rang plus élevé que le système de Hindley-Milner, bien qu'il nécessite d'avantage d'annotation. Les preuves de correction et de complétude pour l'inférence de type de ce système avait déjà été faite manuellement, mais en les formalisant à l'aide d'Abella, ces chercheurs se sont rendus compte d'un certains nombres d'erreurs. Ils les ont rectifié et ont travaillé sur un algorithme d'inférence légèrement différent de l'orginal, afin de mener à bien la mécanisation des preuves nécessaires.

Chapitre 10

Ce dont on n'a pas parlé

Pour être un peu plus exhaustif quant au travail effectué durant la thèse, on peut également citer quelques éléments qu'il n'était pas forcément utile de mentionner plus haut.

10.0.1 Élaboration

Nous nous sommes concentrés, jusqu'ici, sur la génération et la résolution de contraintes, mais n'avons pas parlé de comment effectuer l'élaboration. Inferno permet de décrire l'élaboration des contraintes vers des termes annotés, de façon assez pratique, au même endroit dans le code que la génération de contraintes. Faire en sorte de faciliter l'élaboration nous a conduit à certains choix de design, tel que dans les règles de sémantique, qui sont conçues de façon à pouvoir raisonner relativement simplement sur les types à produire pour annoter les termes. Un des aspects de l'élaboration qui nous a intéressé est la façon de représenter les schémas polymorphes. Nous avons expliqué plus haut qu'il était important de départager les différentes occurrences des variables rigides qui apparaissent dans les annotations de type. C'est d'ailleurs pourquoi nous les traitons comme des structures abstraites. Au moment de la généralisation, et de la formation d'un schéma polymorphe, il faut tout de même exprimer le fait que ces différentes occurrences représentent une même variable. Nous avons décidé de garder les structures abstraites locales dans le schéma polymorphe au lieu de les remplacer directement par des variables génériques. Ce n'est que durant l'instanciation que cette substitution aura lieu.

10.0.2 Implémentation des GADTs dans Inferno

Nous avons implémenté la version avec le "noyau GADT" décrit plus haut. Le code du solveur est construit autour d'un filtrage par motif sur la contrainte en cours de résolution (composante C du système de réécriture). Les positions des quantifications existentielles (qui sont passés au contexte dans la composante S du système de réécriture) sont représentées par des niveaux. Les unifications en cours (composante U du système de réécriture) sont représentées par une structure d'union-find. On peut retrouver le code ici.

De même que dans la formalisation du système de réécriture, nous nous sommes appuyés, dans l'implémentation, sur un traitement des variables rigides comme des structures abstraites et sur une contrainte d'introduction d'égalité. Cela a demandé un changement dans la représentation des schémas polymorphes, et donc des modifications dans la mécanique de généralisation et d'instanciation.

Inferno fonctionnait déjà avec des niveaux, mais nous avons ajouter les portées et la détection d'échappement d'égalités.

Pour représenter les égalités dans le contexte, nous avons implémenté le graphe décrit dans la section 8.3

Il y a un vrai saut de difficulté entre l'implémentation des types de données algébriques (voir 10.0.3) et celles des GADTs. En effet, implémenter les GADTs nécessite d'introduire les structures abstraites et la gestion des égalités de types, et donc de modifier de façon non négligeable le solveur.

Afin de mesurer la fiabilité de notre implémentation, nous avons écrit un certain nombre de tests unitaires qui sont accessibles ici. Ces tests ont révélé certains bugs, que nous avons corrigé, et ils

passent maintenant tous. Cependant, notre implémentation est encore dans un état expérimental et n'est pas intégrée à Inferno pour le moment, il reste notamment à nettoyer l'historique git.

10.0.3 Autres améliorations d'Inferno

Quantified applicatives

La bibliothèque Inferno s'appuie sur un foncteur applicatif `'a co` pour représenter les contraintes qui s'évalueront en un terme annoté de type `'a`. En écrivant un typeur, on se retrouve à manipuler ce type, notamment à travers sa fonction `map`, ainsi qu'une fonction de création de variable existentielle `exist`. Nous n'étions pas satisfait de la façon d'écrire avec l'interface proposée, et notamment sur la façon de définir un nombre dynamique de variables existentielle (pour typer des tuples par exemple). Nous avons donc modifié cette interface, en utilisant les "binding operators" d'OCaml, et avec l'idée de transformer des valeur de sortie en des valeurs d'entrée modale.

L'implémentation de ce changement d'interface correspond à la merge request 8. Ce travail est décrit plus en détail dans une publication pour le ML Workshop de l'ICFP 2020 que l'on peut retrouver ici.

Types algébriques et pattern-matching

Nous avons rajouté les types algébriques et les pattern-matching au langage source d'Inferno. Cela n'a pas demandé de modification dans le solveur. Ces ajouts sont visibles dans les merge requests 9, 11, 12 et 13.

Ingénierie support

Étendre Inferno a demandé du travail d'ingénierie support, afin notamment de tester les nouvelles fonctionnalités de typage rajoutées au fur et à mesure.

- Parser pour le langage source. Afin de pouvoir écrire des tests un peu plus fournis et de ne pas se restreindre à devoir écrire des termes directement sous la forme d'AST, il était important d'implémenter un parser, dont le code est visible dans la merge request 21.

- Améliorations du printer. Nous avons modifié l'approche du printer d'Inferno, car il nous fallait pouvoir affiché à la fois des termes du langage source et ceux du langage cible. Or ces deux langages sont très similaires, et nous voulions éviter de dupliquer le code du printer. Nous avons donc défini un langage générique P, dont le seul but est d'être affiché, et qui subsume à la fois le langage source et le langage cible. Pour afficher des termes de ces deux langages, il suffit alors de les traduire vers P. L'implémentation se trouve dans la merge request 23.

- Nouveau système de test automatisé, accompagné d'un shrinker et d'une suite de tests. Le shrinking est une technique qui consiste à réduire un gros contre-exemple (typiquement un terme qui devrait être typable mais que notre solveur n'arrive pas à typer) en un contre-exemple plus petit, et donc plus facile à comprendre et déboguer. Notre approche pour le shrinking consiste à énumérer les sous-termes du contre-exemple (en faisant attention à garder un terme clos) et à voir si le comportement reste le même ou est corrigé. Si le comportement reste le même on peut continuer à essayer de "shrinker" les sous-termes, sinon on a trouvé un contre-exemple dont on ne sait pas réduire facilement la taille.

  Pour un aperçu de notre système de test avec shrinker, voir le code des merge requests 25, 31 et 33.

- Trous typés. Gabriel Scherer a implémenté des trous typés, qui peuvent avoir n'importe quel type et permettent de shrinker plus efficacement les termes, c'est-à-dire de réduire davantage la taille d'un contre-exemple. Son code est disponible dans la merge request 27.

Conclusion

Nous nous sommes penchés durant cette thèse sur l'inférence de type d'un sous-ensemble d'OCaml, avec une approche par résolution de contraintes. Nous nous demandions si cette approche, qui a certaines bonnes propriétés, pouvaient s'appliquer de façon élégante et efficace à ce langage. Nous avons fait le choix de nous concentrer en priorité sur l'inférence des GADTs, qui est une fonctionnalité cruciale d'OCaml et qui n'était pas, à priori, évidente à formuler avec des contraintes. Pour ce faire, nous sommes partis des travaux existants sur l'inférence de type par contrainte pour ML et sur la gestion des types ambivalents dans les GADTs.

En utilisant la bibliothèque Inferno, nous avons implémenté l'inférence de types pour les GADTs. Pour cela nous avons déterminé une façon particulière de gérer les variables rigides, ce qui nous a demandé d'effectuer des modifications importante dans le solveur de Inferno. Cette façon d'implémenter les GADTs, avec une notion de types ambivalents, nous semble plus claire et efficace que le typeur actuel. Nous avons également été amené à modifier l'interface de la bibliothèque et à rajouter d'autres fonctionnalités que les GADTs, ainsi qu'un système de tests. Nous avons augmenté le solveur défini dans les travaux de François Pottier et Didier Rémy, pour lui faire prendre en charge la gestion des égalités de types introduite par les GADTs. Cela nous a notamment demandé d'y rajouter une nouvelle contrainte et des règles de réécriture en conséquence, en faisant attention à la façon dont nous traitions les variables rigides introduites dans les types de GADTs.

Donner suite à ce travail de thèse pourrait se faire dans différentes directions. Tout d'abord, malgré le travail théorique, nous n'avons pas fourni certaines preuves qui mériteraient d'être faites. Seul un "noyau GADT" a été intégré dans Inferno, et on pourrait vouloir rajouter un langage "surface", plus commode à utiliser. Il serait également intéressant de continuer à rajouter, de façon itérative, des fonctionnalités dans Inferno pour enrichir le système de type traité. On ferait grossir petit à petit le sous-ensemble d'OCaml pris en charge par la bibliothèque, ce qui soulèverait probablement de nouvelles questions intéressantes.

Bibliographie

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

Alain Frisch and Jacques Garrigue. Ocaml manual : locally abstract types, 2010.

Jacques Garrigue and Didier Rémy. Ambivalent Types for Principal Type Inference with GADTs. In *11th Asian Symposium on Programming Languages and Systems*, Melbourne, Australia, December 2013.

Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

Mike Gordon, Robin Milner, Lockwood Morris, Malcolm Newey, and Christopher Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130, 1978.

Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy, Ascánder Suárez, et al. Ocaml, 1996.

Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

Alistair O'Brien. Typing ocaml in ocaml : A constraint-based approach. Master's thesis, Queens' College, 2022.

François Pottier. Hindley-Milner elaboration in applicative style. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2014.

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. A draft extended version is also available.

Thomas Refis. Changes to ambivalence scope tracking, 2018.

John C Reynolds. Towards a theory of type structure. In *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, pages 408–425. Springer, 1974.

Martin Sulzmann, Martin Odersky, and Martin Wehr. *Type inference with constrained types*. Univ., Fak. für Informatik, 1996.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5): 333–412, 2011.

Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10 (2):115–121, 1987.

Pierre Weis and Xavier Leroy. *Le langage CAML*. InterEditions Paris, 1993.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, 2003.

Jinxu Zhao, Bruno C d S Oliveira, and Tom Schrijvers. A mechanical formalization of higher-ranked polymorphic type inference. *Proceedings of the ACM on Programming Languages*, 3 (ICFP):1–29, 2019.