Inférence par contraintes pour les GADTs

Olivier Martinot

INRIA

Sommaire

1 Inférence de types

Seulement le squelette

Inférence de types (Hindley-Milner, ...)
Grammaire de termes et types
Types τ de ML :

$$\tau ::= a \mid s\ \bar{\tau}$$

$$t ::= s\ \bar{t}$$

Inférence de type avec contraintes : génération / résolution / élaboration
Grammaire des contraintes

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

Pour des raisons que nous donnerons plus tard, 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 1.1. On définit $ftv(T)$ (resp. $ftv(C)$) l'ensemble des variables libres (flexibles et rigides) de T (resp. C).

Expliquer les liens avec la logique du premier ordre Solveur correspondant à la sémantique qui renvoie `SAT/UNSAT`
La contrainte let
Schémas polymorphes comme fragment d'une contrainte
Génération

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

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

Rajouter la génération de contrainte pour if-then-else ?
Explications de l'intuition (avec l'aide de l'exemple ?)
Faire un exemple de la traduction d'une annotation

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

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

Définition 1.2. L'instanciation d'un schéma $\forall \bar{Y}.T$ par un type ground t, notée $\forall \bar{Y}.T \preceq t$ est définie par $\exists \bar{t'}.T[\bar{t'}/\bar{Y}] = t$.

Définition σ des schémas
TODO : *Voir si on veut des types infinis ou pas et le cas échéant, rajouter une règle d'unification qui échoue quand U est cyclique dans la sous-section sur l'unification*
Règles de sémantiques

$$\frac{}{E;\gamma \models \text{true}} \qquad \frac{E;\gamma \models C_1 \qquad E;\gamma \models C_2}{E;\gamma \models C_1 \wedge C_2}$$

Expliquer que le $\exists$ et le $\forall$ dans les prémisses sont des quantificateurs de la méta-logique

$$\frac{\exists t, \quad E; \gamma[X \mapsto t] \models C}{E; \gamma \models \exists X.C} \qquad \frac{\begin{array}{l}\text{FORALL} \\ \forall t, \quad E; \gamma[X \mapsto t] \models C\end{array}}{E; \gamma \models \forall X.C} \qquad \frac{\gamma(X) = \gamma(T)}{E; \gamma \models X \text{ is } T}$$

Expliquer le lien entre t et T dans la règle du let

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

définir $\equiv$

Définition 1.3. *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.

Donner un exemple

Définition 1.4. On dit que Y est *dominé* 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 Prenons $U := X = Y \to Z$. Alors on a $Y \prec_U X$ et $Z \prec_U X$. ◊

2 Solveur

2.1 Système de réécriture

2.1.1 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 lequel 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 :

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

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

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

ϵ est une multi-équation de variables d'inférence, variables rigides et structures. Cette approche avec multi-équations nous permet de coller à l'implémentation qui se base sur une structure d'union-find.

Définition 2.1. Une multi-équation est standard, si elle contient au plus une variable rigide ou une structure. Elle est donc de la forme :

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

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

La grammaire de la pile des contextes S est :

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

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.

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.

2.1.2 Deux formes pour la contrainte let

On peut dés maintenant remarquer les deux différents contextes de contraintes let : un premier qui sera utilisé le temps de résoudre la contrainte dans la partie gauche, et un autre pour la contrainte dans la partie droite, avec la particularité qu'il faudra au préalable résoudre la contrainte de la partie gauche, c'est-à-dire la mettre sous forme U. 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 $\forall \bar{Y}.T$, 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.

Exemple 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 $\forall X.X \to X$. ◊

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.

2.2 Règles de réécriture

2.2.1 Règles d'unification

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)
$X = X = \epsilon$	$\to X = \epsilon$		
$s\ \bar{X} = s\ \bar{Y} = \epsilon$	$\to \bar{X} = \bar{Y} \wedge s\ \bar{X} = \epsilon$		
τ	$\to$ 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)
$\mathcal{U}$[false]	$\to$ false	où $\mathcal{U}$ est un contexte d'unification	

On peut remarquer la règle (FUSE) qui fusionne deux multi-équations, et les règles (CLASH-1), etc qui échouent quand on essaye de fusionner des structures et/ou des variables rigides ensemble.

2.2.2 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.

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

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

$S[[] \wedge C](x)$	$= S(x)$	
$S[\exists X[]](x)$	$= S(x)$	si $X \# ftv(S(x))$
$S[\forall X[]](x)$	$= S(x)$	si $X \# ftv(S(x))$
$S[\mathsf{let}\ y = \lambda X.C_1\ \mathsf{in}\ C_2](x)$	$= S(x)$	si $x \neq y \wedge X \# ftv(S(x))$
$S[\mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2](x)$	$= \lambda X.C_1$	si $X \# ftv(T)$

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

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

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 Prenons par exemple la contrainte let suivante :

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

En gardant cette forme, 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 en faisant remonter la quantification de Y, on obtiendrait la contrainte :

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

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 :

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

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

La règle (BUILD-SCHEME) est utilisée quand on a finit de résoudre la contrainte dans la partie gauche d'un let. On peut alors 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. 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. On construit alors un schéma à partir d'une contrainte d'unification U_2 et on tente de résoudre la partie droite C, avec une contrainte d'unification U_1 qui, elle, n'est pas liée à la construction d'un schéma de type pour x (car $X\bar{Y} \# ftv(U_1)$).

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 $\forall \bar{Y}.\exists \bar{X}.U_2 \equiv \mathsf{true}$ et on peut sortir du contexte $\forall \bar{Y} \exists \bar{X}.[]$, en faisant disparaître au passage U_2, puisqu'on sait la résoudre.

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.

$$S[\forall X.\exists \bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true} \quad \to \mathsf{false} \qquad \text{si } X \notin \bar{Y} \wedge X \prec_U^* Z \wedge Z \notin \bar{Y}$$
$$S[\forall X.\exists \bar{Y}[]] \ ; \ X = T = \epsilon \ ; \ \mathsf{true} \quad \to \mathsf{false}$$

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

Exemple Pour illustrer certaines des règles de réécriture, 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} \\
\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\ 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 tel que décrite dans le lemme ci-dessous. ◊

Exemple Prenons un cas où la contrainte se réécrit à false :

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

Dans cet exemple, Y sort de sa portée, puisqu'elle est quantifiée universellement mais égalisée avec une variable définie à l'extérieure (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 2.2. Une forme normale pour le système de réécriture $\rightarrow$ 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 2.3. Si S ; U ; $C \rightarrow S'$; U' ; C' alors $S[U \wedge C] \equiv S'[U' \wedge C']$

Les preuves de ces deux 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.

3 Élaboration

Seulement un squelette, mais quelques éléments dans l'"article jfla"
Voici la grammaire des types τ de Système F :

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \forall\alpha.\tau \mid \mu\alpha.\tau$$

4 Inferno

Seulement un squelette, mais quelques éléments dans l'"article jfla"
Principes : combinateurs, foncteurs
Grammaire des structures internes d'Inferno

$$s ::= (\rightarrow) \mid (\times) \mid Constr \mid \ldots$$

Traduction ML vers Système F

5 GADT, égalités de types

Seulement un squelette, mais quelques éléments dans l'"article jfla"
Rappel sur les GADTs
On fait une version simple avec des constructions refl/use ... in inspirée de l'article de Jacques et Didier
expliquer types ambivalents, localement = plusieurs types possibles mais à l'extérieur un seul, annotation peut lever l'ambivalence

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

$$\tau ::= \cdots \mid \mathsf{eq}\ (\tau, \tau)$$

Ambivalence, multi-équations
Syntaxe des multi-équations
types ambivalents ψ, qui sont des ensembles de types “ground”.

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

Règles de typage, inférence

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

Principalité, expliquer pourquoi mettre l'annotation de type dans le terme dans le use

$$\frac{\Gamma \vdash t : \mathsf{eq}\ \tau_1\ \tau_2 \qquad \Gamma, \tau_1 = \tau_2 \vdash u : \tau}{\Gamma \vdash \mathsf{use}\ t\ \mathsf{in}\ u : \tau} \qquad \frac{\Gamma \vdash t : \tau' \qquad \Gamma \vdash \tau' = \tau}{\Gamma \vdash (t : \tau') : \tau}$$

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

Implémentation du typeur Système F, comparaison de type avec union-find

6 Contrainte d'hypothèse d'égalités

6.1 Retour sur la sémantique des variables rigides

6.1.1 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 $\forall X.C$ 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 toute les autres.

Exemple Dans l'égalité `X list = Y`, la plupart des choix pour `X,Y` rendent l'égalité fausse. ◊

Introduire une égalité fausse comme hypothèse rend l'environnement de typage incohérent, puisqu'elle peut permettre de déduire un typage inattendu. Si une égalité de type contient des variables rigides, quasiment toutes leurs instances rendront l'égalité fausse et donc le typage incohérent !

Exemple Prenons un programme qui ne type pas :

```
let wrong_coercion (type a) (x : a) = x + 1
```

Considérons un bout de la contrainte générée par ce programme : $\forall Y_a.\exists X.X\ \mathsf{is}\ Y_a \wedge X\ \mathsf{is}\ \texttt{int}$, dont la sémantique est $\exists X.X\ \mathsf{is}\ t \wedge X\ \mathsf{is}\ \texttt{int}$ pour tout t. Il n'y a qu'un seul choix pour t qui permet de résoudre cette contrainte : il faut choisir $t = \texttt{int}$. Pour tous les autres choix, la contrainte n'est pas résoluble, donc le programme ne type pas. ◊

6.1.2 Contourner les limites de la contrainte $\forall X.C$

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 variables 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 remplaçons, dans le langage de contrainte, la construction $\forall X.C$ par une contrainte $\forall a.C$, qui introduit localement un type abstrait a (et non plus une variable) :

$$C ::= \cdots \mid \forall a.C$$

La sémantique de $\forall a.C$, similaire à celle de $\forall X.C$, s'exprime par une quantification universelle sur les types ground dans la méta-logique :

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

Les règles de réécriture de cette contrainte sont également similaires à celle déjà introduites pour $\forall X.C$.

Comme nous l'expliquerons plus en détails, quand nous reparlerons des problèmes de partage de structure, ce changement est nécessaire pour traiter correctement les nouvelles constructions que nous introduisons plus loin dans le chapitre.

6.2 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.

6.3 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 $(a = \texttt{int}) \Rightarrow (a = \texttt{bool}) \Rightarrow C$.

Exemple

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

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

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

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

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

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

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.

6.4 Sémantique ambivalente

6.4.1 Un jugement ambivalent

Cette 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 qui utilisent 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). En fait, elle rejette les contraintes ambiguës, dans lesquelles des égalités de types s'échappent 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 :

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

6.4.2 Principales règles

Afin de tirer parti de l'ambivalence, il nous faut revisiter la sémantique de la construction X 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 Prenons $T = Y \to \texttt{bool}$ et $\gamma = [Y \mapsto \{t,\ \texttt{int}\}; X \mapsto \{t',\ t \to \texttt{bool},\ \texttt{int} \to \texttt{bool}\}]$. On a bien :

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

◊

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* :

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

Exemple Prenons $\gamma = [X \mapsto \{t',\ t \to \texttt{bool},\ \texttt{int} \to \texttt{bool}\}]$. On a bien, par exemple :

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

◊

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

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

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'est pas 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 :

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

- si κ est faux, alors l'environnement est incohérent, le témoin de X peut être ambivalent ($|\psi| \geq 1$).

Les règles pour la conjonction et la quantification universelle sont les mêmes que dans la sémantique naturelle :

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

Exemple 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 :

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

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 $t \neq \texttt{int}$.

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

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 $\{\texttt{int}\}$:

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

- $t \neq \texttt{int}$

  Alors nécessairement ψ doit contenir `int` et t. Choisir directement le type ambivalent $\{\texttt{int}, t\}$ fonctionne :

  $$\dfrac{\dfrac{}{\mathsf{false}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}, t\}] \models^{\mathsf{amb}} X \text{ is } a} \quad \dfrac{}{\mathsf{false}; [a \mapsto \texttt{int}; X \mapsto \{\texttt{int}, t\}] \models^{\mathsf{amb}} X \text{ is } \texttt{int}}}{\exists \psi,\quad \mathsf{true}; [a \mapsto \texttt{int}; X \mapsto \psi] \models^{\mathsf{amb}} X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

$\Diamond$

6.4.3 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 :

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

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.

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

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

Exemple 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 :

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

On peut choisir $\psi_X = \{t, \mathtt{int}\}$, $\psi_W = \{t, \mathtt{int}\}$ et $\xi = \forall \emptyset.\{t, \mathtt{int}\}$ pour satisfaire ces jugements. En effet, on a bien $\forall \emptyset.a \preceq a$ et $\forall \emptyset.\mathtt{int} \preceq \mathtt{int}$.

$\Diamond$

6.5 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 $\psi \supseteq \{t, \texttt{int}\}$, on peut prendre $\psi = \{\texttt{int}\}$ ou $\psi = \{\ t\}$:

- 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 ensemble 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.

$$\texttt{Sing}(\gamma_n) = \gamma \text{ tel que } \forall a,\ \gamma(a) = \gamma_n(a) \text{ et } \forall X,\ \gamma(X) = \{\gamma_n(X)\}$$

Théoreme 6.1. 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

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

– $s(Y_i)_i$

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

- $C_1 \wedge C_2$

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

- $\exists X.C$

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

- $\forall a.C$

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

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

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

- $x\ X$

  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 :

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

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

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

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 6.2. Pour tout γ_n et pour toute contrainte C, si $\mathsf{true}; E; \mathtt{Sing}(\gamma_n) \models^{\mathsf{amb}} C$ alors $E; \gamma_n \models C$

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

Corollaire 6.3. Pour tout C, si $\mathsf{true}; E; \emptyset \models^{\mathsf{amb}} C$ 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 6.4. Pour tout γ_n et pour toute contrainte C qui ne contient pas de contrainte d'implication :

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

7 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 plus haut.

7.0.1 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 coller mieux 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 :

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

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

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

7.0.2 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, introduite à différents moments. 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

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

Sous les contexte 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). $\Diamond$

7.1 Unification

7.1.1 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 $\Phi \vdash \epsilon$.

7.1.2 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

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

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. La contrainte est donc équivalente à :

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

On a fait remonter la contrainte X is `int` et on peut maintenant simplifier à nouveau en :

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

$\Diamond$

7.1.3 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.

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

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.

Exemple

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

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

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

$\Diamond$

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 $\mathsf{Eqs}(S)$). 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 trois règles qui correspondent à trois cas d'unification de structures :

$$\begin{array}{ll}
\Phi \vdash s_1\ \bar{X} = s_2\ \bar{Y} = \epsilon & \rightarrow_{\mathsf{Eqs}(S)} (\Phi \cup \Phi' \vdash s_1\ \bar{X} = \epsilon) \wedge (\vdash (\!|\bar{X} \sim \bar{\tau}_1'|\!) \wedge (\!|\bar{Y} \sim \bar{\tau}_2'|\!)) \\
& \quad \text{si } s_1 \neq s_2 \wedge \Phi' \in \mathsf{Eqs}(S) \wedge \Phi \cup \Phi' \Rightarrow s_1\ \bar{\tau}_1 = s_2\ \bar{\tau}_2' \\
\Phi \vdash s\ \bar{X} = a = \epsilon & \rightarrow_{\mathsf{Eqs}(S)} (\Phi \cup \Phi' \vdash a = \epsilon) \wedge (\vdash (\!|\bar{X} \sim \bar{\tau}|\!)) \\
& \quad \text{si } \Phi' \in \mathsf{Eqs}(S) \wedge \Phi \cup \Phi' \Rightarrow s\ \bar{\tau} = a \\
\Phi \vdash \tau_1 = \tau_2 = \epsilon & \rightarrow_{\mathsf{Eqs}(S)} \Phi \cup \Phi' \vdash \tau_1 = \epsilon \\
& \quad \text{si } \Phi' \in \mathsf{Eqs}(S) \wedge \Phi \cup \Phi' \Rightarrow \tau_1 = \tau_2
\end{array}$$

pour un ensemble d'équations Φ' tel qu'il est un sous ensemble des hypothèses d'égalités dans $\mathsf{Eqs}(S)$ qui est un plus vieux possible et qui ne contient pas d'équations superflues.

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 :

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

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'ambiguités.

Exemple

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

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 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 :

$$\begin{aligned}&(\vdash X = a) \wedge (\vdash X = \texttt{int})\\ \to\ &\vdash X = a = \texttt{int}\\ \to\ &\ a = b,\ b = \texttt{int} \vdash X = a\end{aligned}$$

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

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

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

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

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 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. $\Diamond$

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.

$$\begin{array}{lll}
\Phi \vdash s_1\ \bar{X} = s_2\ \bar{Y} = \epsilon & \rightarrow \mathsf{false} & \text{(CLASH-AMB-1)} \\
& \quad \text{si } s_1 \neq s_2 \wedge \forall \bar{\tau_1}\bar{\tau_2}, \mathsf{Eqs}(S) \not\Rightarrow s_1\ \bar{\tau_1} = s_2\ \bar{\tau_2} & \\
\Phi \vdash s\ \bar{X} = a = \epsilon & \rightarrow \mathsf{false} & \text{(CLASH-AMB-2)} \\
& \quad \text{si } \mathsf{Eqs}(S) \not\Rightarrow s\ \bar{X} = a & \\
\Phi \vdash a = b = \epsilon & \rightarrow \mathsf{false} & \text{(CLASH-AMB-3)} \\
& \quad \text{si } a \neq b \wedge \mathsf{Eqs}(S) \not\Rightarrow a = b &
\end{array}$$

7.2 Nouvelles règles de réécriture

7.2.1 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

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

- $U\#X$ qui fait un test de non appartenance d'une variable X à un ensemble de multi-équations U.

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

7.2.2 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 :

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

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 qui ne dépendent pas de l'hypothèse d'égalité 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.

EX-IMPCST

$$\frac{V \in ftv(\exists X\bar{Y}.U) \qquad X \prec_U V}{S[\phi : (\tau_1 = \tau_2) \Rightarrow \exists X\bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true} \rightarrow S[\exists \bar{X}.\phi : (\tau_1 = \tau_2) \Rightarrow \exists \bar{Y}.[]] \ ; \ U \ ; \ \mathsf{true}}$$

Dans la règle, V est une variable liée quelques 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 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 terme 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 :

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

grâce à la règle EX-IMPCST. ◊

Une fois simplifiée, résoudre une contrainte d'hypothèse d'égalité 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.

POP-EQ

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

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

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

On ne peut pas faire remonter X car elle dépend de l'égalité entre a et `int`, mais on sait que la multi-équation $(a = \texttt{int} \vdash X = a)$ est triviale donc on peut l'oublier. De plus, on sait que $\vdash Y = \texttt{bool}$ 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 :

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

◊

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.

7.2.3 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 un égalité ϕ est nécessaire à prouver une multi-équation, mais que celle-ci n'est pas triviale.

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

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

Exemple

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

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

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

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

Exemple

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

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. ◊

7.2.4 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 7.1. (Correction) Si $S \ ; \ U \ ; \ C \to S' \ ; \ U' \ ; \ C'$ alors $S[U \wedge C] \equiv S'[U' \wedge C']$

On peut également conjecturer la complétude :

Conjecture 7.2. (Complétude) La configuration $S \ ; \ U \ ; \ C$ possède une chaîne de réécriture vers $[] \ ; \ U' \ ; \ \mathsf{true}$ si et seulement si $\mathsf{true}; \emptyset; \emptyset \models^{\mathsf{amb}} S[U \wedge C]$

TODO : *vérifier que c'est bien ça*

8 Nouvelle gestion des variables rigides

8.1 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 :

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

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 terme 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 $V_1 = \texttt{int}$.

8.2 Structures abstraites

8.2.1 Départager les différentes occurrences

On est ainsi forcé de distinguer les différentes occurrences des variables rigides, celles-ci ne peuvent pas être traitées comme une seule variable d'inférence. Traiter une variable rigide comme une variable d'inférence (avec une contrainte de la forme $\forall X.C$) 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.

8.2.2 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.

8.2.3 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 let (voir le manuel oca **TODO** : *meilleur affichage pour la citation*) :

```
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.

8.2.4 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

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

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.

8.3 Généralisation et instanciation avec des structures abstraites

8.3.1 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 $\forall a.\lambda X.\exists(X_1 \text{ is } a)(X_2 \text{ is } a).X \text{ is } X_1 \to X_2$. Comment alors instancier un tel schéma, sachant qu'en plus celui-ci pourra 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 de 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 Dans la contrainte suivante :

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

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). ◊

8.3.2 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 :

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

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

BUILD-RIGID-SCHEME

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

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 $\forall\bar{a}\exists X\bar{Y} \equiv \text{true}$ 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

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

◊

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 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` $\to a$) pourrait être instancié à la fois par `unit` $\to$ `int` et `unit` $\to$ `bool`. ◊

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

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

Pour instancier x, W doit pouvoir être unifiée avec le schéma de type induit pas $\forall \bar{a}\lambda X.C$. 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

$$\begin{array}{rl} & S[\mathsf{letr}\ x\ =\ \forall a\lambda X.[]\ \mathsf{in}\ x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}]\ ;\ (a = \mathtt{int}) \Rightarrow X\ \mathsf{is}\ a \to a \wedge \exists Y.Y\ \mathsf{is} \\ \to & S[\mathsf{letr}\ x\ =\ \forall a\lambda X.[]\ \mathsf{in}\ x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool}]\ ;\ X\ \mathsf{is}\ a \to a\ ;\ \mathsf{true} \\ \to & S[\mathsf{letr}\ x\ =\ \forall a\lambda X.X\ \mathsf{is}\ a \to a\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ x\ W \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge x\ V \wedge V\ \mathsf{is}\ \mathtt{bool} \to \mathtt{bool} \\ \to^* & S[\mathsf{letr}\ x\ =\ \forall a\lambda X.X\ \mathsf{is}\ a \to a\ \mathsf{in}\ []]\ ;\ \mathsf{true}\ ;\ \exists W'.W\ \mathsf{is}\ W' \to W' \wedge W\ \mathsf{is}\ \mathtt{int} \to \mathtt{int} \wedge \exists V'V\ \mathsf{is}\ V' \to V' \wedge V\ \mathsf{is}\ \mathtt{b} \\ \to & \ldots \end{array}$$

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 `int` $\to$ `int` et `bool` $\to$ `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)
```

◊

8.4 Sémantique de la contrainte letr

8.4.1 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.

8.4.2 Décomposition de la contrainte letr

La sémantique de la contrainte let pouvait être obtenue par dé-sucrage et décomposition en deux contraintes plus simples : $\mathsf{let}\ x = \lambda X.C_1\ \mathsf{in}\ C_2 \equiv \exists X.C_1 \wedge \mathsf{def}\ x\ :\ \lambda X.C_1\ \mathsf{in}\ C_2$. 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 cela 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 donné ci-dessous :

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

Exemple

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

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

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

c'est-à-dire :

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

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.

$\Diamond$

Bien que cette sémantique pour $\exists a.C$ soit simple, cette contrainte n'a pas été implémentée, car il n'est pas évident de comprendre son sens. Il s'agit d'une des particularités de notre représentation à base de structures abstraites. Ceci dit, ne pas implémenter cette contrainte n'est pas forcément un problème. En effet, nous avons déjà décrit une implémentation de la contrainte letr qui n'est pas basée sur la décomposition en deux parties $\forall \bar{a} \ldots \exists \bar{a} \ldots$.

9 Implémentation du solveur

Quelques éléments, WIP

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

9.1.1 Une portée par égalité introduite

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. Quand on implémente un tel 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 noms d'égalités 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 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 0, et `e2` la portée 1. 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 1. ◊

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

Par ailleurs, les variables flexibles ont aussi un "niveau", qui correspond à la position de leur quantification existentielle dans le contexte (par exemple on peut choisir le niveau de De Bruijn de l'égalité la plus récemment ajoutée avant eux). Une variable de niveau n peut être définie par une multi-équation de portée au plus n.

Exemple 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 toute 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 7.1). 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 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.

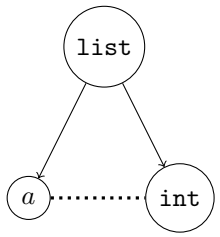
9.2 Gestion des égalités de types avec un graphe

9.2.1 Stocker des égalités de types dans un graphe

On l'a vu, la sémantique ambivalente ne nécessite pas de stocker les égalités de type introduites. En fait les égalités de types pouvaient se transposer dans les types ambivalents des variables flexibles. Pour des raisons de simplicité et d'efficacité, nous n'avons pas procéder de la même manière dans l'implémentation. En effet, la sémantique d'une contrainte existentielle consistait à deviner un ensemble de types ambivalents, ce qui n'est pas évident à implémenter. Contrairement au jugement sémantique, il nous faut une façon de stocker les égalités de types introduites, ainsi que 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.

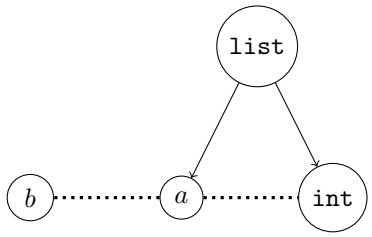
Exemple Si on veut représenter l'égalité `list` a = `list int`, il faut qu'une arête relie le noeud a avec le noeud `int`.



◊

Chaque feuille d'un noeud structurel est unique. Cela permet de tester facilement l'égalité entre différentes structures.

Exemple 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]
```

9.2.2 Garder trace de la portée des égalités

D'une certaine façons, les différents noeuds égaux dans le graphes sont interchangeables. Mais les hypothèses d'égalité étant introduites à différents endroits du programme, elles n'ont pas toute la même portée. Chaque arête doit donc fournir une information sur la portée de l'égalité qu'elle représente. Cette portée sera utilisé 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 à assurer la cohérences de ses équations. Lors des parcours de graphe, il faudra donc garder en mémoire la plus grande portée des arêtes traversées.

TODO : *exemple tikz*

On peut représenter le graphe avec le type suivant :

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

TODO : *Retirer les arêtes quand on sort d'un contexte d'hypothèse d'égalité.*

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

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

TODO : *Expliquer comment vérifier une égalité : si rien dans le graphe / si quelque chose dans le graphe.*

10 Travaux liés

References

Ocaml manual : locally abstract types.

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.

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.

11 TODOs, idées

11.1 TODOs

- Relire les TODOs dans les commentaires latex de l'ancien fichier
- Section 6 : Expliquer pourquoi on n'utilise pas de contraintes de la forme $(X = Y) \Rightarrow C$
- Section 6 : Vérifier le choix pour la grammaire de E dans le jugement sémantique
- Section 7 : Dire quelques mots sur la correspondance entre résolution de contrainte et sémantique, vérifier que c'est bien ce que l'on veut

11.2 Questions

- On autorise les égalités entre des structures différentes, qu'est-ce-que ça veut dire résoudre une contrainte dans un contexte incohérent ?

11.3 Idées

- Faire un exemple avec $eq(a, b) \to eq(b, a)$
- Quand ça s'y prête, écrire sous la forme d'un tuto Inferno : 1) comment utiliser Inferno, utiliser l'API 2) comment rajouter une fonctionnalité dans Inferno
- Section 9 : expliquer comment les règles de réécriture se traduisent dans l'implem, par exemple en présence d'hypothèse d'égalité, les multi-équations dans la composante U sont "normalisées" (même si la réécriture est en petits pas)
- Section 9 : expliquer comment l'exécution du solveur se complexifie, et comment le debugging est modifié