Inférence par contraintes pour les GADTs
WIP (Work in Progress)

Olivier Martinot and Gabriel Scherer

INRIA

Abstract

Inferno est une bibliothèque, développée par François Pottier, pour l'inférence de types par résolution de contraintes. Elle fournit à l'utilisateur une interface pour générer des contraintes et les élaborer vers des termes annotés une fois résolues, ainsi qu'un solveur. Nous présentons notre travail en cours : l'ajout des GADTs dans Inferno. Nous utilisons les règles de typage de Garrigue et Rémy qui s'appuient sur la notion d'ambivalence. Nous expliquons les modifications au système de type, aux contraintes d'inférence, et décrivons les éléments d'une implémentation efficace.

1 Contexte

1.1 La bibliothèque Inferno

Inferno est une bibliothèque, développée par François Pottier, pour l'inférence de types par résolution de contraintes (Pottier, 2014). Elle fournit à l'utilisateur une interface pour générer des contraintes et les élaborer vers des termes annotés une fois résolues, ainsi qu'un solveur de ces contraintes. Si tout se passe bien, l'utilisateur n'a qu'à écrire son typeur avec les contraintes d'inferno. Mais le langage de contraintes ne prend pas en charge certains systèmes de types complexes. Il faut donc l'étendre avec de nouvelles formes de contraintes et implémenter leur support dans le solveur.

Dans notre travail, on s'intéresse à l'inférence de type d'un sous-ensemble d'OCaml, qu'on élabore en l'annotant vers des termes Système F. En étendant progressivement le langage source pris en charge pour y ajouter des fonctionnalités d'OCaml, on est contraint d'étendre dans le même temps le backend d'Inferno pour répondre à nos besoins.

1.2 Rajouter les GADTs dans Inferno

On se penche sur l'ajout du support pour les GADTs (Guarded Algebraic Datatypes) dans Inferno.[1] Nous les rajoutons donc au langage source (un sous-ensemble de OCaml), ainsi que dans le langage cible (Système F). Le système de contrainte d'Inferno n'est pas assez riches pour traiter les GADTs, il nous faut donc l'étendre : rajouter un nouveau type de contrainte, expliquer comment elle est censée être résolue, et implémenter ces changements dans le solveur.

Ce travail est encore en cours au moment de la rédaction de cet article, certaines parties ont déjà été traitées, d'autres de façon partielle, et d'autres pas du tout. Nous signalons notre avancement dans les différentes parties.

[1] Nous supposons que le lecteur ou la lectrice connaît les GADTs en OCaml. Pour une découverte ou un rappel, voir par exemple le chapitre correspondant du manuel OCaml: https://v2.ocaml.org/manual/gadts-tutorial.html.

2 Une approche pour les GADTs

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

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

On peut ainsi introduire une égalité dans le contexte :

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

Ici l'égalité est introduite dans le contexte, ce qui permet de voir `n` comme un entier et de lui ajouter `1`.

Mais certains programmes peuvent devenir plus compliqué à typer :

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

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

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

Ici, la première branche du `if` retourne une valeur de type `a` et la deuxième une valeur de type `int`. Les deux branches retournent donc des valeurs de types différents pour le monde extérieur qui ne sait pas que $a =$ `int`. Il faut donc rejeter cet exemple car l'introduction de l'égalité de type $a =$ `int` ne doit pas s'échapper de sa portée. Pour passer outre cette restriction, il suffit d'annoter le type de retour de la fonction. Mais on peut avoir envie de ne pas choisir en donnant comme type de retour de cette fonction un type *ambivalent* : ni `int` ni `a`, mais un ensemble (`int` $\approx$ `a`) qui contient ces deux types.

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

3 Langage cible : étendre Système F avec des égalités de types

Quand on fait de l'inférence de type, il faut avoir une façon d'annoter le programme source avec des types, et pouvoir ensuite vérifier que cela correspond bien au typage qu'on attend. Il est donc utile d'implémenter un typeur indépendant pour le langage cible. Pour inférer les types d'un programme ML avec des GADTs, on peut envisager de traduire le programme vers une version de Système F, dans laquelle on aura rajouté la possibilité d'exprimer des égalités entre types.

3.1 Nouvelles règles de typage

On introduit un type $\mathsf{eq}\ \tau_1\ \tau_2$ pour les expressions qui apportent une preuve de l'égalité entre τ_1 et τ_2. On introduit également la possibilité de supposer des égalités de types dans le contexte.

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

Programme OCaml:

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

Notre langage cible (syntaxe imaginaire):

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

Refl_τ est le constructeur de preuves d'égalité :

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

```
Int 42
```

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

La constructions `use .. in` introduit une égalité dans le contexte dans lequel on type une expression. On a également une règle de conversion.

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

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

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

On définit aussi une construction absurd_τ utilisée pour signaler une incohérence dans les égalités de types introduites.

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

Cela est utile par exemple dans le filtrage par motif, pour signaler des branches inaccessibles.

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

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

3.2 Comparaison de type avec union-find

On a besoin de tester l'égalité entre types τ de Système F, dont voici la grammaire :

$$\tau ::= \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha.\tau \mid \mu\alpha.\tau \mid (\tau, \tau)\ \mathsf{eq}$$

Pour qu'un type soit bien formé, il ne faut pas qu'un $\forall$ apparaisse sous un μ.

On a une notion de "structure" s pour les noeuds du langage d'inférence utilisés en interne par le solveur d'Inferno :

$$s ::= (\to) \mid (\times) \mid \mathsf{eq} \mid \dots$$

Inferno testait déjà les égalités entre mu-types, mais on a fait évoluer la façon de faire, avec des changements importants à l'algorithme existant pour traiter aussi les équations de types dans le contexte. Comme l'ancien algorithme, nous utilisons des structures d'union-find que l'on peut représenter sous forme de graphes cycliques.

Dans cet article, nous représentons un type par un graphe, en passant par un système avec des variables (X, Y, ...) comme noeuds du graphe, et des ensembles d'équations récursives E pour décrire le graphe. Dans ce système, on peut supposer des égalités entre noeuds dans le contexte, et on suppose qu'on peut décider efficacement l'égalité entre graphes sous ces hypothèses (grâce à la structure d'union-find).

Les noeuds de nos graphes sont soit des variables X, soit des structures de types s (noeuds "flèche", "étoile", etc), soit des noeuds $\forall$, soit des entiers (niveaux de De Bruijn) $\hat{n}$.

$$\sigma ::= X \mid s\ \bar{X} \mid \forall.\sigma \mid \hat{n}$$

On traduit des types Système F sous des contextes qui peuvent être des correspondances entre variable de type et des noeuds du graphe, ou des égalités entre noeuds du graphe.

$$\Gamma ::= \emptyset \mid \Gamma, \alpha \mapsto X \mid \Gamma, X = Y$$

Au fur et à mesure que l'on parcourt un type, on cherche à construire des ensembles d'équations E, entre des noeuds du graphe et des types, de la forme :

$$E ::= \emptyset \mid E, X = \sigma$$

On peut maintenant définir un jugement qui, étant donné un contexte Γ, transforme un type τ vers un noeud du graphe pointé par une variable fraîche X, et générant un ensemble d'équation E.

$$\Gamma \vdash \tau \Rightarrow_X E$$

Ainsi, la règle pour les flèches peut se définir :

$$\frac{\mathsf{fresh}\ X_1, X_2 \quad \Gamma \vdash \tau_1 \Rightarrow_{X_1} E_1 \quad \Gamma \vdash \tau_2 \Rightarrow_{X_2} E_2}{\Gamma \vdash (\tau_1 \to \tau_2) \Rightarrow_X E_1 \cup E_2 \cup (X = X_1 \to X_2)}$$

On introduit une règle pour les structures de l'utilisateur, dans laquelle on s'appelle récursivement sur les arguments du type :

$$\frac{\mathsf{fresh}\ (X_i)_i \quad (\Gamma \vdash \tau_i \Rightarrow_{X_i} E_i)_i}{\Gamma \vdash s\ (\tau_i)_i \Rightarrow_X \bigcup_i E_i \cup (X = s\ (X_i)_i)}$$

Pour les variables et $\mu\alpha.\tau$:

$$\frac{\alpha \mapsto Y \in \Gamma}{\Gamma \vdash \alpha \Rightarrow_X (X = Y)} \qquad \frac{\Gamma, \alpha \mapsto X \vdash \tau \Rightarrow_X E}{\Gamma \vdash \mu\alpha.\tau \Rightarrow_X E}$$

Pour définir $\forall\alpha.\tau$, on crée une variable $\hat{n}$ pour un n correpondant au nombre de lieurs qu'on a traversé jusque là, à la façon des niveaux de De Bruijn, et on associe un type Z à α, rajoutant l'association $\alpha \mapsto Z$ dans le contexte. Le nom de la variable α n'est donc pas présent dans le graphe, dans lequel il y aura un noeud $\forall$ qui a comme fils le reste du type.

$$\frac{n := \mathsf{count}\{\beta \mapsto \hat{m} \in \Gamma\} \quad \mathsf{fresh}\ Y, Z \quad \Gamma, \alpha \mapsto Z \vdash \tau \Rightarrow_Y E}{\Gamma \vdash \forall\alpha.\tau \Rightarrow_X (E, X = \forall.Y, Z = \hat{n})}$$

Pour décider l'égalité entre deux types sous un contexte d'égalité, on les traduit vers des graphes et on teste l'égalité entre les graphes.

État actuel On a complètement implementé (et testé) cette partie-là.

4 Contrainte d'hypothèse d'égalité

Il nous faut maintenant aborder le fonctionnement de l'inférence de types avec une approche de résolution de contraintes. On doit définir une contrainte qui représente l'ajout d'équations de types dans le contexte de typage et établir sa sémantique.

4.1 Rappel: inférence par contraintes

Rappel Définissons d'abord des types utilisateurs t (une structure s peut représenter le type flèche, produit, etc) qui peuvent contenir des variables de type (rigides) a :

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

Durant l'inférence de type, on raisonne avec des types qui, eux, ont des variables d'inférence X aux feuilles, et des contraintes C que l'on va résoudre. Une solution à une contrainte fournit un type t pour chacune de ces variables d'inférence.

$$T ::= X \mid a \mid s\ \bar{X} \qquad C ::= \mathsf{true} \mid \mathsf{false} \mid C \wedge C \mid \exists X.C \mid T = T$$

Dans une implémentation complète, il faudrait un générateur de contraintes, validé avec la sémantique, mais c'est hors de la portée de cet article. Nous nous concentrons dans la suite

sur l'introduction de contraintes pour les GADTs, leurs sémantiques et le fonctionnement du solveur. On travaille avec des règles de réécriture (discutées dans la section suivante) qui transforment la contrainte jusqu'à une forme résolue (si elle est résoluble). Mais il nous faut également définir une sémantique des contraintes, qui correspond à une interprétation déclarative des contraintes.

Donnons quelques règles de sémantique du solveur de contraintes.

$$\frac{\exists t, \quad \gamma[X \mapsto t] \models C}{\gamma \models \exists X.C} \qquad \frac{\forall t, \quad \gamma[a \mapsto t] \models C}{\gamma \models \forall a.C} \qquad \frac{\gamma(X) = \gamma(t)}{\gamma \models X \text{ is } t}$$

L'idée est d'avoir des contraintes faciles à exprimer. Il faut ensuite être capable de trouver une correspondance entre la sémantique les décrivant, et le solveur (*correction* : si le solveur trouve une solution pour la contrainte C, alors $\emptyset \models C$). Celui-ci devrait être en mesure de donner une solution à chaque fois que le typage est possible (*completude* : si $\emptyset \models C$ alors le solveur trouve une solution).

$\Diamond$

4.2 Syntaxe

$$C ::= \cdots \mid a = t \Rightarrow C$$

On ajoute la contrainte $(a = t \Rightarrow C)$ qui introduit l'égalité $a = t$ dans le contexte dans lequel sera résolu la contrainte C.

4.3 Sémantique naturelle pour la nouvelle contrainte

Une façon naturelle de définir la sémantique de la contrainte précédente est :

$$\frac{\gamma(a) = \gamma(t) \quad \Longrightarrow \quad \gamma \models C}{\gamma \models a = t \Rightarrow C}$$

Si on peut prouver que l'égalité $\gamma(a) = \gamma(t)$ implique que γ est une solution de C, alors on est capable de résoudre la contrainte $a = t \Rightarrow C$

On peut, par exemple, résoudre la contrainte $\forall a \; \exists X. \; a = \texttt{int} \Rightarrow X \text{ is } \texttt{int}$. On s'aperçoit qu'on doit trouver un type t' qui satisfait $a = \texttt{int} \Rightarrow X \text{ is } \texttt{int}$ (on considère pour le moment que le mot-clef is représente l'égalité entre types)

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

On a alors deux choix possibles pour t' : soit `int` soit t.

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

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

Comme prévu, on rejette la contrainte suivante :

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

Comme aucune égalité entre a et `int` n'a été introduite, on ne peut pas en déduire que X peut valoir à la fois l'un et l'autre.

Un cas plus intéressant concerne 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) =
  match x with Refl -> if y > 0 then y else 0
```

Quand on applique les règles de sémantique, on obtient :

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

Cette contrainte est donc satisfiable avec notre sémantique naturelle (elle a deux solutions : $X = a$ ou $X = \texttt{int}$). On peut en effet annoter le programme de deux façons différentes :

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

ou

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

Mais aucun des deux typages n'est principal. On est trop permissif, X ne devrait pas pouvoir être de deux types à la fois. Notre solveur va détecter cette ambiguité et réécrire la contrainte vers false. Le solveur n'est plus complet par rapport à cette sémantique. On introduit une autre sémantique, qui repose sur une notion de types ambivalents et rejette aussi cette contrainte.

4.4 Sémantique ambivalente

Pour cela, on raisonne avec des types ambivalents Ψ, qui sont des ensembles de variables ou de structures. Dans les règles de sémantique, les solutions γ proposent des types ambivalents pour les variables de type X. La sémantique de X is t devient donc un test d'appartenance à un ensemble:

$$\frac{\gamma(t) \in \gamma(X)}{E; \gamma \models^a X \text{ is } t}$$

E est un contexte d'égalités. Un type ambivalent Ψ est un ensemble $\{\rho_1, \rho_2, \rho_3 \dots\}$ de types simples ρ, que nous notons comme une multi-équation $\rho_1 \approx \rho_2 \approx \rho_3 \approx \dots$.

$E ::= \quad \emptyset \mid E, a = t$
$\Psi ::= \quad \emptyset \mid \rho \approx \Psi$
$\rho ::= \quad a \mid s\ (\Psi_i)_i$

On introduit un jugement $E \vdash \Psi$ qui assure que les types dans Ψ peuvent être prouvés égaux (on note $E \vdash \rho = \rho'$ quand les égalités dans E impliquent l'égalité $\rho = \rho'$):

$$\frac{\forall \rho \approx \rho' \in \Psi,\ E \vdash \rho = \rho'}{E \vdash \Psi}$$

$$\frac{E \vdash \Psi \qquad E; \gamma[X \mapsto \Psi] \models^a C}{E; \gamma \models^a \exists X.C} \qquad\qquad \frac{E;\ \gamma, a \models^a C}{E;\ \gamma \models^a \forall a.C}$$

La règle pour la contrainte d'introduction d'égalité donne :

$$\frac{(E, a = t); \gamma \models^a C}{E; \gamma \models^a a = t \Rightarrow C}$$

Cette sémantique nous permet de rejeter les programmes ambigüs qui avaient des solutions dans la sémantique naturelle. Pour s'en convaincre, on peut regarder quelle forme prendrait une dérivation de la sémantique ambivalente sur l'exemple donné plus haut. On s'aperçoit qu'on est bloqué :

$$\dfrac{\exists \Psi, \quad \emptyset \vdash \Psi \qquad \dfrac{\dfrac{\dfrac{\exists \Psi, \quad (a = \texttt{int}); [a; X \mapsto \Psi] \models^a X \text{ is } a \quad (a = \texttt{int}); [a; X \mapsto \Psi] \models^a X \text{ is } \texttt{int}}{\exists \Psi, \quad (a = \texttt{int}); [a; X \mapsto \Psi] \models^a X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\emptyset; [a; X \mapsto \Psi] \models^a a = \texttt{int} \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\emptyset; [a] \models^a \exists X. a = \texttt{int} \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}}{\emptyset; \emptyset \models^a \forall a\ \exists X.\ a = \texttt{int} \Rightarrow X \text{ is } a \wedge X \text{ is } \texttt{int}}$$

Pour trouver un Ψ approprié, il faudrait que $\Psi \supseteq \{a, \texttt{int}\}$ mais alors Ψ ne serait pas cohérent dans le contexte d'égalité vide (i.e. on ne pourrait pas prouver $\emptyset \vdash \Psi$). 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.

On voudrait prouver une équivalence entre les deux sémantiques quand il n'y a pas de contraintes d'implication : si une contrainte C ne contient pas de sous-contrainte d'implication, alors

$$\{\gamma \mid \gamma \models C\} = \{\gamma \mid \gamma \models^a C\}$$

On voudrait également s'assurer que s'il existe une solution à une contrainte en sémantique ambivalente, il en existe une en sémantique naturelle :

$$Si \quad \exists \gamma, \gamma \models^a C \quad alors \quad \exists \gamma', \gamma' \models C$$

5 Résolution de la contrainte dans le solveur

Une fois définies la syntaxe et la sémantique de cette nouvelle contrainte, nous expliquons comment s'effectue sa résolution dans le solveur d'Inferno. Nous donnons des règles de réécriture qui étendent le système de réécriture de Pottier and Rémy (2005), puis nous donnons des éléments sur comment l'implémenter efficacement dans Inferno.

5.1 Règles de réécriture

Rappel Dans Pottier and Rémy (2005), les auteurs définissent un système de réécriture de la forme

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

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 un ensemble de multi-équations représentant une partie de la contrainte déjà résolue.

$$S ::= [] \mid S[\exists X.[]] \mid S[[] \wedge C] \mid ... \qquad U ::= \mathsf{true} \mid \mathsf{false} \mid (E \vdash \epsilon) \mid U \wedge U \mid \exists X.U$$

La configuration $S \ ; \ U \ ; \ C$ correspond à la contrainte $S[U \wedge C]$.

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

$\Diamond$

Nos multi-équations $E \vdash \epsilon$ est la donnée d'égalités ϵ entre des variables d'inférence et au plus une variable rigide ou une structure (comme dans les travaux précédents), et en plus d'un ensemble E d'égalités de types qui garde trace des hypothèses d'égalités dont nous avons eu besoin pour construire la multi-équation.

$$E ::= \emptyset \mid E, \phi : a = t \qquad \epsilon ::= X_1 = \cdots = X_n (= a \mid s \ \bar{Y})$$

Pour unifier deux multi-équations $(E_1 \vdash \epsilon_1)$ et $(E_2 \vdash \epsilon_2)$ ensemble, il faut regarder l'eventuelle variable rigide ou structure dans chacune d'elle, voir si elle sont unifiables, et rajouter les équations correspondante si c'est le cas. On obtient ainsi une opération de la forme :

$$S \vdash (E_1 \vdash \epsilon_1) + (E_2 \vdash \epsilon_2) \to (E_1, E_2, E \vdash \epsilon_3)$$

Nous avons quelques idées pour définir cette opération dans un style en réécriture par petits pas, qui s'intègrerait bien avec ce qui existait avant dans Pottier and Rémy (2005), mais nous ne l'avons pas encore fait.

Comme on rajoute une nouvelle contrainte, on rajoute aussi un contexte

$$S ::= ... \mid S[\phi : a = t \Rightarrow []]$$

On a annoté l'égalité $a = t$ avec un nom ϕ pour pouvoir y faire référence dans les conditions de bord des règles de réécriture.

On aura besoin de se référer à l'ensemble des égalités dans le contexte :

$$\mathsf{Eqs}([]) = \emptyset \qquad \mathsf{Eqs}(S[\exists X.[]]) = \mathsf{Eqs}(S) \qquad ... \qquad \mathsf{Eqs}(S[\phi : a = t \Rightarrow []]) = \mathsf{Eqs}(S); a = t$$

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

$$\frac{\mathsf{fresh} \ \phi}{S \ ; \ U \ ; \ a = t \Rightarrow C \to S[\phi : a = t \Rightarrow []] \ ; \ U \ ; \ C} \tag{1}$$

Pour les autres règles de réécriture, on a besoin de définir

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

$$(E \vdash \epsilon)\#X = X \notin \epsilon \qquad (E \vdash \epsilon)\#\phi \; = \; \phi \notin E \qquad (\exists Y.U)\#X = U\#X \; siX \neq Y$$

$$(U_1 \wedge U_2)\#X = (U_1\#X) \wedge (U_2\#X)$$

Les règles suivantes s'appliquent quand on a réussi à réécrire la contrainte à résoudre en true et qu'on a une hytpothèse d'égalité sur la pile.

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). On doit faire attention à ne pas extruder des variables qui dépendent l'égalité ϕ dans l'unification.

$$\frac{U_{|\bar{Y}}\#\phi}{S[\phi : a = t \Rightarrow \exists \bar{Y}\bar{Z}.[]] \; ; \; U \; ; \; \mathsf{true} \rightarrow S[\exists \bar{Y}.\phi : a = t \Rightarrow \exists \bar{Z}.[]] \; ; \; U \; ; \; \mathsf{true}} \tag{2}$$

Quand on a appliqué la règle (2) autant que nécessaire, il peut rester des variables $\bar{X}$ qui dépendent de l'égalité ϕ. On a U de la forme $U_1 \wedge U_2$. U_1 est la partie qui dépend de variables extérieures mais pas de ϕ (car $U_1\#\phi$), et elle ne dépend plus des variables locales $\bar{X}$ (car $U_1\#\bar{X}$) qui n'ont pas pu être extrudée. U_1 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.

$$\frac{U_1\#\phi, \bar{X} \quad \big(\mathsf{Eqs}(S), a = t \Rightarrow \exists \bar{X}.U_2\big) \equiv \mathsf{true}}{S[\phi : a = t \Rightarrow \exists \bar{X}.[]] \; ; \; U_1 \wedge U_2 \; ; \; \mathsf{true} \rightarrow S \; ; \; U_1 \; ; \; \mathsf{true}} \tag{3}$$

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). Si un bout de U dépend à la fois de variables extérieures et de phi on a une erreur (on est bloqué).

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).

$$Si \quad S \; ; \; U \; ; \; C \rightarrow S' \; ; \; U' \; ; \; C' \quad alors \quad \{\gamma \mid \gamma \models^a \; S[U \wedge C]\} = \{\gamma \mid \gamma \models^a \; S'[U' \wedge C']\}$$

On peut également conjecturer que la configuration $S \; ; \; U \; ; \; C$ possède une chaîne de réécriture vers $[] \; ; \; U' \; ; \; \mathsf{true}$ si et seulement si $\emptyset; \emptyset \models^a S[U \wedge C]$

5.2 Mécanisme de portées

Les règles ci-dessus donnent 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 – c'est le rôle des conditions $U\#\phi$ dans nos règles ci-dessus. 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 ϕ utilisées pour justifier sa cohérence.

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.

Quand on unifie des variables d'inférence, on fait l'union de leurs multi-équations, et la cohérence du résultat peut dépendre d'égalités $\phi : a = t$ qui n'étaient pas nécessaires pour les multi-équations avant unification. Cela peut donner une portée maximum plus grande que les portées maximum précédentes. 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; cela correspond exactement à une violation de la condition de non-échappement des égalités dans la règle (2) ci-dessus.

État actuel Nous comptons ajouter la notion de portée au solveur Inferno, mais ce n'est pas encore fait. (En plus des hypothèses d'égalité, elle est utile pour les variables rigides ou les types abstraits locaux.)

6 Travaux liés

TODO [J'ai regardé les travaux cités dans les références notamment Simonet and Pottier (2007) mais je n'ai pas fini de bien comprendre comment bien lier ces travaux à ce que je fais]

References

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

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

Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.