2ème année du MPRI
Université Paris VII

Projet
Chantal Keller

Un système de types avec polarité

Assistants de preuve

Professeurs: Christine Paulin
Bruno Barras

3 février 2009

1 Introduction

Ce rapport explique ma démarche concernant le projet d'assistants de preuve, consistant à étudier le typage avec polarités du λ-calcul simplement typé avec constantes, à en montrer la décidabilité, et à prouver la corrections des polarités définies. Il est accompagné d'une archive contenant les fichiers suivants, chacun correspondant à une partie du sujet :

- `definitions.v` ;
- `decidabilite_typage.v` ;
- `correction_polarites.v`.

Un `Makefile` permet de compiler ces trois fichiers, avec la commande `$ make depend` suivie de `$ make`.

J'ai essayé d'utiliser au mieux les sections de Coq afin de regrouper les éléments similaires, et de faire un maximum de lemmes pour que les preuves soient lisibles. La suite du rapport suit le découpage en fichiers et en sections de mon code.

2 Définition du système de types

Dans cette partie, on définit les règles de typage avec polarité. Cela correspond au fichier Coq `definitions.v`.

2.1 Polarité, types, termes environnement

Polarité La polarité est le type inductif `polarity` avec trois constructeurs constants `nonvariant`, `covariant` et `contravariant`. On définit l'opposé d'une polarité `opp_pol` et la multiplication de deux polarités `mul_pol` par cas.

Types Un type `typ` est soit un type de base `int`, soit un type flèche `arrow` avec polarité. Pour ce dernier type, on utilise des notations proches de celles de l'énoncé : `->+` pour $\to^+$, `->-` pour $\to^-$ et `->o` pour $\to^\circ$.

Termes On commence par définir l'ensemble des constantes `cst` comme un type inductif avec les constructeurs constants demandés. La définition des termes et des environnements est donnée dans le sujet.

2.2 Jugement de typage

Typage des constantes Commençons par définir `sigma`, la fonction de typage des constantes, par cas. À part pour `rec`, le typage est assez évident : le successeur et l'addition sont croissants, et la soustraction est croissante par rapport à sa première variable et décroissante par rapport à sa deuxième variable.

La dernière constante est la "récursion sur les entiers de type τ". Sans tenir compte des polarités, j'ai choisi un opérateur `rec` ayant le type `int -> ((int -> int) -> int -> int) -> int`. Cet opérateur permet de définir des fonctions récursives comme suit : par exemple, si

$$g \triangleq \lambda f n.\text{if } n = 0 \text{ then } a \text{ else } f\ (n-1)$$

alors

$$\lambda n.\text{rec } n\ g$$

est la fonction qui double l'entier passé en argument. J'ai fait ce choix car, dans la partie 4, cet opérateur s'interprète aisément dans le modèle du λ-calcul grâce à l'opérateur de point fixe. C'est la fonction qu'il prend en argument qui doit faire le test sur l'entier, et non pas lui-même, ce qui évite d'avoir à coder le test à zéro et à prouver qu'il fait bien ce qu'on pense qu'il fait. Cela m'a donc paru plus simple qu'un récurseur comme celui du système T par exemple, pour lequel je n'ai pas imaginé d'autre interprétation que celle utilisant le test à zéro.

Concernant les polarités, on obtient le type `int ->o ((int ->o int) ->o int ->o int) ->o int` car `rec` n'a pas à être croissant par rapport à l'entier passé en argument, et comme la fonction passée en argument correspond à ce que `rec` va renvoyer, on doit obtenir la même polarité, qui est *a priori* $\circ$.

Jugement de typage On définit ensuite la fonction `inv_pol`, qui calcule $p^{-1}\Delta$. On peut alors définir un inductif `type` qui décrit exactement les jugements de typage.

Pour le cas de la variable, j'ai choisi de mettre deux constructeurs selon que la polarité est $+$ ou $\circ$, plutôt que de mettre un test d'égalité. Cela ne change pas la sémantique du typage, mais permettra de générer directement les deux cas lorsqu'on fera une induction sur une dérivation de typage, plutôt que de devoir utiliser la tactique `destruct`.

Justification Dans le cas de la variable, si $p = +$, on obtient bien que la variable n croît si n croît. Si $p = \circ$, on a bien que la variable n'est pas modifiée, donc croît, si on ne la modifie pas. Enfin, on doit bien écarter le cas $p = -$, car dans ce cas la variable décroît si on la fait décroître.

Le cas des constantes et celui de la λ-abstraction sont évidents, et la fonction `sigma` a été détaillée plus haut.

Le cas de l'application est le plus intéressant. Regardons par cas sur p :

- si $p = +$, u est croissante par rapport à chacun de ses arguments. Donc la polarité des variables de v est la même que celle des variables de $u\ v$;
- si p$= -$, u décroît lorsque ses arguments croissent. Donc, si une variable x a une polarité $+$ dans v, elle doit décroître pour que $u\ v$ croisse ; et si elle a une polarité $-$ dans v, elle doit croître pour que $u\ v$ croisse. Il faut donc bien typer v dans un contexte dans lequel toutes les polarités ont été inversées ;
- si $p = \circ$, on ne sait pas ce que fait u lorsque ses arguments avec la polarité $+$ croissent et ses arguments avec la polarité $-$ décroissent. Il ne faut donc pas que ces arguments interviennent dans le typage de v.

3 Décidabilité du typage

Dans cette partie, on montre la décidabilité du typage. Cela correspond au fichier `decidabilite_typage.v`.

Inférence de types La fonction `infer` infère le type d'un terme dans un environnement donné. Comme demandé, si le terme n'est pas typable, la fonction renvoie `None`. Cette fonction est facile à écrire, car les règles de typage sont dirigées par la syntaxe.

La seule subtilité est que l'on doit pouvoir décider l'égalité du type de départ de u et du type de v dans le cas de l'application. C'est pour cela qu'on prouve d'abord la décidabilité

de l'égalité entre types, qui dépend de celle de l'égalité entre polarités, qui elle-même découle directement de l'injectivité des constructeurs des inductifs.

Unicité du typage On montre que si l'inductif `type` permet de typer un même terme dans un même environnement avec deux types, alors ces derniers sont égaux. Cette preuve se fait par induction sur une des deux dérivations de typage, sans difficulté.

Utilitaires En Coq, lorsqu'on veut faire une analyse par cas sur une variable qui apparaît dans le contexte et dans le but, on n'obtient pas forcément ce à quoi on s'attendait (parfois, les substitutions ne sont pas faites dans le contexte). C'est pourquoi j'ai défini quelques lemmes qui me servent pas la suite à pouvoir faire des études de cas facilement, et avec les cas qui m'intéressent.

Décidabilité du typage Pour prouver la décidabilité d'une propriété, connaissant un algorithme qui la calcule, le plus simple est de prouver que cet algorithme est correct et complet.

Correction La correction consiste à prouver que, lorsque l'inférence de types renvoie un type, ce type est bien correct. La preuve se fait par induction sur le type, et ne présente pas de difficulté.

Complétude La complétude consiste à prouver que, lorsqu'un terme est typable, l'inférence de types trouve bien ce type. La preuve se fait par induction sur le jugement de typage et, comme l'inférence de types se fait exactement de la même façon, elle est très simple.

4 Correction des polarités

Cette partie cherche à prouver la correction des polarités. Cela correspond au fichier `correction_polarites.v`

Exemples Voici comment définir les λ-termes demandés :

```
Definition id := mk_fun (fun x => x).
Definition delta := mk_fun (fun x => app x x).
Definition omega := app delta delta.
Definition ze := mk_fun (fun z => mk_fun (fun f => z)).
Definition su := mk_fun (fun n => mk_fun (fun z => mk_fun (fun f =>
     app f (app (app n z) f)))).
Definition fix_fun := mk_fun (fun f => app (mk_fun (fun x =>
     app f (app x x))) (mk_fun (fun x => app f (app x x)))).
```

On montre que `fix_fun` est un opérateur de point fixe en utilisant la β-équivalence.

Par la suite, on aura également besoin de λ-termes calculant l'addition, la soustraction et le récurseur (tel que défini dans la partie 2.2). J'ai choisi de les définir comme suit :

```
Definition plus := mk_fun (fun n => mk_fun (fun m =>
   app (app m n) su)).

Definition pred := mk_fun (fun n => mk_fun (fun x =>
    mk_fun (fun f => app (app (app n (mk_fun (fun _ => x)))
   (mk_fun (fun g => mk_fun (fun h => app h (app g f)))))
   (mk_fun (fun u => u))))).
Definition minus := mk_fun (fun n => mk_fun (fun m =>
   app (app m n) pred)).

Definition fix_point := mk_fun (fun n => mk_fun (fun g =>
   app (app fix_fun g) n)).
```

Pour l'addition et la soustraction, j'ai préféré un codage direct plutôt que l'utilisation du récurseur. En effet, cela permet de mieux voir les propriétés dont on a besoin sur ces opérateurs pour prouver la correction des polarités ; en revanche, on ne pourra pas utiliser les propriétés du récurseur, il faudra tout remontrer.

Pour le prédécesseur, j'ai préféré un codage direct, contrairement au codage qui utilise des couples. Il est moins intuitif, mais comme pour le récurseur, j'ai pensé que les preuves se feraient plus facilement que s'il fallait déjà encoder les couples, puis prouver un certain nombre de propriétés dessus.

Relation d'ordre Définissons la "relation d'ordre" comme demandé dans l'énoncé (on verra par la suite que cette relation n'est en fait pas réflexive (elle n'est pas antisymétrique non plus, mais nous n'utilisons pas cette propriété)).

Sur les entiers, la relation d'ordre `leq` est définie inductivement de manière usuelle. La relation typée est définie par induction sur le type, comme demandé dans l'énoncé. Enfin, on a besoin d'un typage avec polarités et d'une relation d'ordre entre contextes de typage, comme dans l'énoncé.

Interprétation L'interprétation des constantes se fait grâce aux termes du modèle définis dans la partie **Exemples**. Pour ajouter une variable à une valuation, comme on utilise les indices de De Bruijn, il suffit de décaler les valeurs de la fonction, et de mettre le terme correspondant à la valuation de la nouvelle variable à l'indice 0. L'interprétation des termes se fait alors par induction sur la structure des termes, de manière usuelle.

Utilitaires On a besoin de lemmes qui vont nous être très utiles par la suite : des propriétés de la multiplication des polarités et de l'opposée d'une polarité, ainsi que sur la polarité $\circ$.

Réflexivité `leq` est réflexive par définition, mais `ty_leq` ne l'est pas : en effet, dans le cas où f a un type flèche, le fait qu'il soit inférieur à lui-même signifierait qu'il est obligatoirement croissant par rapport à chacune de ses variables !

En revanche, on peut prouver une propriété plus faible et qui nous servira par la suite, à savoir que si on a deux termes comparables x et y, alors x est inférieur à lui-même et y également. Cette propriété s'explique par le fait qu'à partir du moment où l'on sait que x est comparable avec un autre terme, si x a le type flèche, alors x est croissant par rapport à chacun de ses arguments, qui est la propriété qui nous manquait précédemment.

On dispose ensuite de diverses propriétés liées à la réflexivité pour comparer des contextes.

Transitivité `leq`, `ty_leq` et `ty_leq_p` sont transitives, ce qui se démontre aisément.

Propriétés de l'addition Pour pouvoir montrer que l'addition a bien la polarité énoncée dans la partie 2.1, on va avoir besoin de propriétés sur l'addition dans notre modèle. En particulier, on souhaite montrer que la relation d'ordre est compatible avec l'addition :

$$\begin{cases} b \leqslant c & \Rightarrow & a+b \leqslant a+c \\ a \leqslant b & \Rightarrow & a+c \leqslant b+c \end{cases}$$

Si la première propriété découle directement de la définition que j'ai choisie pour coder l'addition, la deuxième est beaucoup moins évidente. Je n'ai pas réussi à la prouver, mais je pense qu'elle n'est effectivement pas prouvable : en effet, cette propriété n'est *a priori* vraie que pour les entiers de Church, et on cherche à la prouver pour un λ-terme quelconque...!

Pour résoudre ce problème, j'ai eu l'idée d'imposer que `leq` ne compare que des entiers de Church, en modifiant sa définition ainsi :

```
Inductive leq (n: lambda) : lambda -> Prop :=
  | leq_n : is_church n -> leq n n
  | leq_s : forall m: lambda, leq n m -> leq n (app su m).
```

Pour cela, j'ai essayé deux codages différents de `is_church` :

```
Inductive is_church : lambda -> Prop :=
  | ch_ze : is_church ze
  | ch_su : forall (n: lambda), is_church n -> is_church (app su n).
```

et :

```
Fixpoint app_n (n: nat) (f z: lambda) {struct n} : lambda :=
  match n with
    | O => z
    | S p => app f (app_n p f z)
  end.

Definition is_church (x: lambda) : Prop :=
   exists (n: nat), x = mk_fun (fun z => mk_fun (fun f => app_n n f z)).
```

Aucun des deux essais n'a abouti, c'est pourquoi je ne les ai pas laissés dans la version définitive du fichier Coq, et que la deuxième propriété est admise.

Propriétés de la soustraction Là encore, pour pouvoir montrer que la soustraction a bien la polarité énoncée dans la partie 2.1, on a besoin de propriétés de compatibilité avec la relation `leq`. Je n'ai pas réussi non plus à prouver ces propriétés, qui sont en outre plus difficiles que pour l'addition à cause du cas particulier de `pred O = O`, et de la définition plus complexe pour le prédécesseur que pour le successeur.

Propriétés du récurseur De même, on a besoin de propriétés sur le récurseur, que je n'ai pas réussi à définir correctement. J'ai donc admis que la polarité de `rec` était correcte.

Correction des polarités La correction des polarités se fait par induction sur la dérivation de typage de $\Delta \vdash m : \tau$.

- Dans le cas des variables, il suffit d'appliquer la définition (j'ai choisi d'utiliser la fonction `nth` sur les listes aussi bien pour le jugement de typage que pour définir la comparaison des contextes, ce qui rend la preuve facile).
- Dans le cas des constantes, il faut montrer que les interprétation dans notre modèle ont bien les polarités choisies. Cela se fait très facilement pour zéro (ce n'est pas une fonction) et pour le successeur. Pour l'addition et la soustraction, on a besoin de propriétés qui semblent correctes, mais que je n'ai pas toujours réussi à démontrer (voir ci-dessus). Pour le récurseur, je n'ai pas réussi à extraire des propriétés qui peuvent sembler vraies, et j'ai donc directement supposé qu'il avait la bonne polarité (voir ci-dessus).
- Dans le cas de l'application, il suffit de prouver le lemme suggéré dans l'énoncé. Ce lemme se prouve en démontrant que `ty_leq_p` est compatible avec l'application (lemme `ty_leq_p_app`).
- Dans le cas de l'abstraction, on utilise le fait que :

  $$\text{si } x \leqslant^{qp} y \in \tau \text{ et } \rho_1 \leqslant^{q} \rho_2 \in \Delta, \text{ alors } \rho_1[0 := x] \leqslant^{q} \rho_2[0 := y] \in (\Delta; (p, \tau))$$

  Ce lemme se démontre simplement. On en déduit alors, en étant obligé de considérer beaucoup de cas, mais sans difficulté, la correction des polarités de l'abstraction.

5 Conclusion

J'ai prouvé la décidabilité du typage ainsi que la correction des polarités sauf dans le cas des constantes. On dispose donc d'un algorithme prouvé correct et complet pour décider les types avec polarités dans ce cas très simple de λ-calcul simplement typé avec constantes, et d'un début de preuve pour la correction des polarités. Il faudrait affiner mes définitions d'opérateurs sur les entiers de Church dans le modèle, pour pouvoir obtenir toutes les propriétés souhaitées.