

# Travail pratique #3

IFT-2035

November 29, 2009

## 1 Survol

Ce TP a pour but de vous familiariser avec le langage Prolog tout en révisant les règles de typage.

Comme pour les TP précédents, les étapes sont les suivantes:

1. Parfaire sa connaissance de Prolog.
2. Lire et comprendre cette donnée.
3. Lire, et comprendre le code fourni.
4. Compléter le code fourni.
5. Écrire un rapport. Il doit décrire votre expérience pendant les 4 points précédents: problèmes rencontrés, surprises, choix que vous avez dû faire, options que vous avez sciemment rejetées, etc... Le rapport ne doit pas excéder 5 pages.

Ce travail est à faire en groupes de 2 étudiants. Le rapport (au format PDF ou Postscript) et le code sont à remettre par remise électronique avant la date indiquée. Aucun retard ne sera accepté. Indiquez clairement votre nom au début de chaque fichier.

Si un étudiant préfère travailler seul, libre à lui, mais l'évaluation de son travail n'en tiendra pas compte. Si un étudiant ne trouve pas de partenaire, il doit me contacter immédiatement. Des groupes de 3 ou plus sont **exclus**.

## 2 Le langage $\mu PTS$

Vous allez écrire un programme Prolog qui va manipuler des expressions d'un arbre de syntaxe abstraite pour un petit  $\lambda$ -calcul dénommé  $\mu PTS$ . La particularité de  $\mu PTS$  est de mélanger les expressions normales et les types. Une manière de représenter ce fait est de dire "type:type": le type de 1 est `int`, mais `int` est aussi une expression normale, de type `type`, qui lui-même est une expression de type `type`.

Le langage a la syntaxe suivante:

$$e ::= c \mid x \mid \lambda x : e_1. e_2 \mid e_1 e_2 \mid \Pi x : e_1. e_2$$

où  $c$  représente n'importe quelle constante prédéfinie, cela inclura les nombres entiers, les types prédéfinis (`int`, `list`, et `type`) et les opérations prédéfinies (`cons`, `nil`, `plus`, et `minus`);  $x$  est une variable;  $\lambda x : e_1. e_2$  est une fonction d'argument formel  $x$ , de corps  $e_2$ , et dont l'argument actuel doit avoir type  $e_1$ ;  $e_1 e_2$  est un appel de fonction; et  $\Pi x : e_1. e_2$  est le type d'une fonction, prenant des arguments de type  $e_1$  et qui renvoie des valeurs de type  $e_2$ ; c'est une généralisation de  $\forall x. e_2$  et de  $e_1 \rightarrow e_2$ : si  $x$  n'apparaît pas dans  $e_2$ , alors on peut considérer  $e_1 \rightarrow e_2$  comme du sucre syntaxique pour écrire  $\Pi x : e_1. e_2$ , et  $\Pi x : \text{type}. e_2$  est équivalent au type polymorphe  $\forall x. e_2$ .

Par exemple, le type de la fonction identité de Haskell  $\text{identity} = \lambda x \rightarrow x$  est généralement noté  $\text{identity} :: \alpha \rightarrow \alpha$ , ce qui signifie vraiment  $\forall \alpha. \alpha \rightarrow \alpha$ . Dans notre langage cela s'écrit à la place  $\Pi \alpha : \text{type}. \Pi x : \alpha. \alpha$ , ce qui signifie que la fonction identité prend en réalité deux arguments, le premier étant le type  $\alpha$ . Donc un appel pourrait ressembler à  $\text{identity int } 5$ , qui correspond à prendre la fonction  $\text{identity}$ , puis à la spécialiser pour le cas des nombres entiers et finalement lui passer le paramètre 5.

Les règles d'évaluation sont les règles habituelles du  $\lambda$ -calcul:

$$\begin{array}{llll} (\lambda x : e_1. e_2) e_3 & \rightsquigarrow & e_2[e_3/x] & \\ \text{plus } n_1 \ n_2 & \rightsquigarrow & n_3 & \text{where } n_3 = n_1 + n_2 \\ \text{minus } n_1 \ n_2 & \rightsquigarrow & n_3 & \text{where } n_3 = n_1 - n_2 \\ \dots & & & \end{array}$$

Ces règles correspondent à un seul pas d'exécution; elles sont donc répétées aussi longtemps de nécessaire, en les appliquant partout où cela est possible. Les constantes prédéfinies ont les types suivants:

```

type   : type
fix    :  $\Pi t : \text{type}. \Pi f : (\Pi x : t. t). t$ 

int    : type
n      : int
plus   :  $\Pi x_1 : \text{int}. \Pi x_2 : \text{int}. \text{int}$ 
minus  :  $\Pi x_1 : \text{int}. \Pi x_2 : \text{int}. \text{int}$ 

list   :  $\Pi t : \text{type}. \Pi n : \text{int}. \text{type}$ 
nil    :  $\Pi t : \text{type}. \text{list } t \ 0$ 
cons   :  $\Pi t : \text{type}. \Pi x : t. \Pi n : \text{int}. \Pi y : \text{list } t \ n. \text{list } t \ (\text{plus } n \ 1)$ 

```

La fonction `fix` est un *opérateur de point fixe* qui offre accès à la récursion:  $\text{fix } f \equiv f (f (f (...)))$ . Bien sûr à l'exécution, l'expansion se fait seulement sur demande:  $\text{fix } f \rightsquigarrow f (\text{fix } f)$ .

Les entiers sont sans surprises, surtout si l'on utilise la notation  $\rightarrow$  pour le type de `plus` et `minus`:  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$ .

$$\begin{array}{c}
\frac{\Gamma(x) = e}{\Gamma \vdash x : e} \\
\\
\frac{\Gamma \vdash e_1 : \text{type} \quad \Gamma, x : e_1 \vdash e_2 : e_3}{\Gamma \vdash \lambda x : e_1. e_2 : \Pi x : e_1. e_3} \\
\\
\frac{\Gamma \vdash e_1 : \text{type} \quad \Gamma, x : e_1 \vdash e_2 : \text{type}}{\Gamma \vdash \Pi x : e_1. e_2 : \text{type}} \\
\\
\frac{\Gamma \vdash e_1 : \Pi x : e_4. e_5 \quad \Gamma \vdash e_2 : e_6 \quad e_4 \simeq e_6}{\Gamma \vdash e_1 \ e_2 : e_5[e_2/x]} \\
\\
e_1 \simeq e_2 \quad \Leftrightarrow \quad e_1 \rightsquigarrow^* e_3 \wedge e_2 \rightsquigarrow^* e_3
\end{array}$$

Figure 1: Règles de typage de  $\mu PTS$ .

La partie plus intéressante est celle des listes: les listes prédéfinies ont type  $\text{list } t \ n$  où  $t$  est le type de chaque élément et  $n$  est la taille de la liste. Donc le constructeur de type  $\text{list}$  prend deux arguments (le premier, un type, le second un entier) et le résultat est un type. Aussi  $\text{nil}$  prend un argument  $t$  qui est le type des éléments de la liste et renvoie une liste vide du type approprié. Dans ce cas, le  $\Pi$  est utilisé pour le polymorphisme paramétrique, et le type peut se lire  $\forall t. \text{list } t \ 0$ . Finalement,  $\text{cons}$  prend quatre arguments: le type des éléments de la liste, l'élément à ajouter, la longueur de la queue de la liste, et la queue de la liste, et il renvoie une liste de même type, sauf que la longueur est augmentée de 1. Parmi ces 4  $\Pi$ , le premier correspond à du polymorphisme paramétrique, le deuxième et le dernier correspondent à des fonctions normales (et pourraient s'écrire avec la notation  $\tau_1 \rightarrow \tau_2$ ), et le troisième correspond à ce que l'on appelle le *typage dépendant*, où un paramètre qui est une valeur plutôt qu'un type ( $n$  dans notre cas), apparaît dans les types.

La figure 1 montre les règles de typage. Le jugement principal est de la forme  $\Gamma \vdash e_1 : e_2$  et signifie que  $e_1$  a type  $e_2$  dans le contexte  $\Gamma$  qui est une liste donnant le type de chaque variable connue. Les règles correspondantes doivent s'assurer que le jugement n'est vrai que si l'expression est syntaxiquement valide, qu'elle ne fait référence qu'à des variables existantes, et qu'elle est exempte d'erreurs de typage.

La première règle dit simplement qu'une expression qui n'est qu'une référence à une variable a comme type celui associé à cette variable dans le contexte  $\Gamma$ , ce qui implique bien sûr que la variable doit être présente dans  $\Gamma$ . La règle de typage de  $\lambda$  est identique à la règle classique vue au cours, mis à part l'usage de  $\Pi x : e_1. e_3$  au lieu de  $e_1 \rightarrow e_3$ , et mis à part qu'elle s'assure aussi que  $e_1$  est elle-même une expression de type valide. De même la règle de typage de  $\Pi x : e_1. e_2$  vérifie simplement que  $e_1$  et  $e_2$  sont tous deux des types bien formés. La règle pour l'appel de fonction est un peu plus délicate. La règle habituelle

dit seulement:

$$\frac{\Gamma \vdash e_1 : e_4 \rightarrow e_5 \quad \Gamma \vdash e_2 : e_4}{\Gamma \vdash e_1 e_2 : e_5}$$

Donc il y a 3 différences:

- l'usage de  $\Pi$ , bien sûr.
- l'usage de  $e_4 \simeq e_6$ : cette contrainte vérifie que  $e_4$  et  $e_6$  sont des types équivalents, comme on s'y attend, mais en leur permettant toutefois d'être syntaxiquement différents. Ceci est nécessaire de manière à permettre par exemple de passer un argument de type `list t 7` là où le code attend un objet de type `list t (plus 3 4)`, et vice versa.
- la substitution de  $[e_2/x]$  dans  $e_5$ : c'est là que le  $x$  de  $\Pi x : \dots$  est utilisé. Si  $x$  n'apparaît pas dans  $e_5$  (et donc  $\Pi x : e_4.e_5$  peut s'écrire  $e_4 \rightarrow e_5$ ), alors la substitution ne fait aucune différence, mais si le  $x$  y apparaît, comme dans  $\Pi t : \text{type.list } t \ 0$ , alors le type du résultat doit bien sûr refléter l'argument qui a été passé.

La règle  $e_1 \simeq e_2$  indique que  $e_1$  et  $e_2$  sont équivalentes dans le sens que les deux expressions se réduisent au même résultat. Donc une manière de vérifier cette relation est d'appliquer les règles de réduction sur  $e_1$  et  $e_2$  autant que nécessaire jusqu'à ce que l'on obtienne soit le même résultat, soit qu'il n'y ait plus rien à réduire. Remarquez que la réduction se fait ici dans un contexte non vide, contrairement à l'évaluation classique: il y aura peut-être des variables non instanciées. Par exemple, il faut pouvoir découvrir que  $\Pi x : \text{type.list } x \ (\text{plus } 3 \ 4) \simeq \Pi y : \text{type.list } y \ 7$  malgré que  $x$  et  $y$  ne sont pas connus. Le langage étant pur, l'ordre d'évaluation n'a pas d'importance. De plus, le renommage  $\alpha$  est implicite donc par exemple  $\lambda x : t.x$  et  $\lambda y : t.y$  sont considérées comme deux expressions identiques.

### 3 Votre travail

Le langage  $\mu PTS$  est un peu trop verbeux en pratique, donc on aimerait pouvoir l'utiliser sans avoir à écrire tous ces types. Pour cela, vous allez implanter un système d'inférence de types à la *Hindley-Milner*, c'est à dire un système similaire à celui utilisé pour Haskell.

Puisque  $\mu PTS$  est un langage à typage dépendant, la vérification (et l'inférence par la même occasion) de type peut nécessiter l'évaluation d'expressions, donc vous allez aussi devoir implanter une sorte d'interpréteur de ce langage.

La clé de l'inférence de types à la *Hindley-Milner* réside dans la gestion spéciale de la forme `let x = e1 in e2`: après avoir inféré le type de  $e_1$ , le système va le généraliser, c'est à dire transformer toutes les variables de types restées libres en paramètres supplémentaires, et ensuite remplacer chaque usage de  $x$  dans  $e_2$  par une par un appel à  $x$  avec le nombre correspondant d'arguments

pour instancier ces paramètres supplémentaires. Par exemple, si on commence avec:

```
let id = λx.x in (id id) 5
```

l'inférence va d'abord trouver que

$$\lambda x.x : \Pi x:t.t$$

ensuite, il va collecter toutes les variables libres (dans ce cas, il n'y a que  $t$ ) et les ajouter comme paramètres implicites, autant dans la définition de  $id$  que dans ses usages:

```
let id = λt:type.λx:t.x
in ((id ?) (id ?)) 5
```

Les arguments implicites à passer (notés ? ci-dessus) sont inférés par la suite lors de l'inférence du type de  $e_2$  (i.e. de  $((id ?) (id ?)) 5$ ), pour finir avec:

```
let id = λt:type.λx:t.x
in ((id (Πx:int.int)) (id int)) 5
```

La gestion du `let` est relativement délicate, et est incluse dans le code fourni. Votre travail va se concentrer sur le reste de l'inférence qui consiste principalement en l'application systématique des règles de typage. Comme on le voit, ce genre d'inférence fait plus que d'inférer le type d'une expression, vu qu'elle la modifie en y rendant explicites divers éléments implicites dans le code source. Pour cette raison, on l'appelle parfois *élaboration*.

Pour ce code, vous aller directement manipuler la syntaxe abstraite qui est représentée en Prolog de la manière suivante:

$n$	un entier Prolog représente un entier $\mu PTS$
$x$	un atome Prolog représente une variable $\mu PTS$
<code>lambda(x, t, e)</code>	$\lambda x:t.e$
<code>app(e<sub>1</sub>, e<sub>2</sub>)</code>	$e_1 e_2$
<code>pi(x, t<sub>1</sub>, t<sub>2</sub>)</code>	$\Pi x:t_1.t_2$

La syntaxe ci-dessus est celle du  $\mu PTS$  verbeux, bien sûr. Pour rendre le code plus agréable à écrire nous avons donc ajouté un peu de sucre syntaxique et d'autres facilités dans le langage source:

<code>let(x, e<sub>1</sub>, e<sub>2</sub>)</code>	la forme <code>let</code> habituelle
<code>let(x, t, e<sub>1</sub>, e<sub>2</sub>)</code>	idem mais avec annotation de type
<code>forall(x, t<sub>1</sub>, t<sub>2</sub>)</code>	comme <code>pi</code> mais pour un paramètre implicite
<code>lambda(x, e)</code>	un $\lambda$ sans annotation de type
<code>t<sub>1</sub> -&gt; t<sub>2</sub></code>	sucre syntaxique pour $\Pi x:t_1.t_2$
<code>app(e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>)</code>	sucre syntaxique pour l'appel $e_1 e_2 e_3$
<code>app(e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>, e<sub>4</sub>)</code>	sucre syntaxique pour l'appel $e_1 e_2 e_3 e_4$
<code>fix(x, e<sub>1</sub>, e<sub>2</sub>)</code>	définition d'un $x$ récursif: <code>let x = fix(λx.e<sub>1</sub>) in e<sub>2</sub></code> .
<code>fix(x, t, e<sub>1</sub>, e<sub>2</sub>)</code>	idem avec annotation de type.

La forme `let` est nécessaire pour indiquer à *Hindley-Milner* quand faire la généralisation. La forme avec annotation de type peut être utile parfois lorsque l'inférence ne fonctionne pas.

La forme `forall` permet de distinguer dans le type d'une expression quels paramètres sont implicites dans le code source (qui utilisent `forall` et lesquels sont explicites (qui utilisent `pi`)).

Le code fourni inclut une règle `wf` qui teste simplement si un terme est syntaxiquement valide. Cette règle n'est pas utile en soi mais vous donne un exemple trivial de code qui utilise cette syntaxe abstraite.

Donc vous allez devoir implanter en Prolog les règles de typage données (y compris la règle de conversion  $e_1 \simeq e_2$ ). Pour cela vous aurez aussi besoin d'implanter la fonction de substitution  $e_1[e_2/x]$ . Une des particularités du typage dépendant et qu'il nécessite l'évaluation de termes dont certaines variables ne sont pas encore connues. Cela pose certaines difficultés du point de vue de la substitution. Par exemple, il faut pouvoir correctement traiter une substitution telle que  $(\lambda y.x+y)[y/x]$ , qui ne doit pas renvoyer  $\lambda y.y+y$  mais  $\lambda y'.y+y'$  vu que les deux  $y$  sont deux variables différentes. Pour générer une nouvelle variable, vous aurez besoin du prédicat `new_atom`, dont l'usage est documenté dans le manuel de GNU Prolog.

Assurez-vous que les relations correspondant à des fonctions, (par exemple `subst`) ne renvoient vraiment qu'un seul et unique résultat.

Concrètement, vous allez donc devoir compléter le code fourni, principalement les relations `subst`, `normalize`, `infer` et `elaborate`. Le point d'entrée principal est `elaborate(SRC, E)`. Vous avez le droit de modifier n'importe quelle partie du code, pour autant que l'usage de `elaborate(SRC, E)` fonctionne. Il n'est cependant pas indispensable de modifier le code fourni.

À remarquer aussi que `subst`, `normalize` et `infer` opèrent normalement sur le code  $\mu PTS$  de bas niveau (i.e. sans sucre syntaxique et sans `let`), sauf que `elaborate` peut les appeler avec des termes qui contiennent des `forall`, auquel cas il faut les traiter comme des `pi`.

## 4 Remise

L'étape 1 consiste à implanter `subst` et `normalize`. Vous devez remettre un fichier:

```
% remise ift2035 tp2 pts-eval.pl
```

Pour l'étape 2 qui inclut les 4 parties, vous devez remettre deux autres fichiers:

```
% remise ift2035 tp2 pts.pl rapport.pdf
```

## 5 Détails

Comme toujours, la note sera basée sur des tests automatiques qui vérifient le bon fonctionnement de votre code, sur le style, où l'élégance et la simplicité sont

beaucoup plus prisées que l'efficacité, et sur la qualité du rapport.

Tout usage de matériel (code ou texte) emprunté à quelqu'un d'autre (trouvé sur le web, ...) doit être dûment mentionné, sans quoi cela sera considéré comme du plagiat.

- Le code ne doit en aucun cas dépasser 80 colonnes.
- Vérifiez la page web du cours, pour d'éventuels errata, et d'autres indications supplémentaires.