

# Travail pratique #3

IFT-2035

November 13, 2017

⏏ Dû le 3 décembre à minuit !!

## 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. 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  $\mu PTS$ <sup>1</sup>. La particularité de  $\mu PTS$  est de mélanger les expressions normales et les types. Une

---

<sup>1</sup> Le nom vient du fait qu'il s'agit d'une sorte de *Pure Type System*, en fait très proche de *System U*.

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**.

La syntaxe du langage ne distingue donc pas le sous-langage des types: les deux sont mis en commun. Plus précisément, le cœur du langage a la syntaxe suivante, valide autant pour les types que pour les expressions “normales”:

$$e ::= c \mid x \mid \lambda x : e_1. e_2 \mid e_1 e_2 \mid \Pi x : e_1. e_2 \mid \text{let } x : e_1 = e_2 \text{ in } e_3$$

où  $c$  représente n'importe quelle constante prédéfinie, cela inclura les nombres entiers, les types prédéfinis (`int`, `list`, **type**, ...) et les opérations prédéfinies (`cons`, `nil`, `+`, `-`, ...);  $x$  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 utiliser  $e_1 \rightarrow e_2$  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 argument, le premier étant le type du deuxième. 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:

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

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
      :
int   : type
float : type
n     : int
+     :  $\Pi x_1 : \text{int}. \Pi x_2 : \text{int}. \text{int}$ 
-     :  $\Pi x_1 : \text{int}. \Pi x_2 : \text{int}. \text{int}$ 

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

```

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

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

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

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 quatre  $\Pi$ , 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 2 différences:

- l'usage de  $\Pi$ , bien sûr.
- 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.

Dans le code fourni, les expressions  $\mu PTS$  sont représentées par les termes Prolog suivants:

$x$	$X$
$\lambda x : e_1.e_2$	<code>fun(X, E1, E2)</code>
$\Pi x : e_1.e_2$	<code>pi(X, E1, E2)</code>
$e_1 \ e_2$	<code>app(E1, E2)</code>
<code>let x : e1 = e2 in e3</code>	<code>let(X, E1, E2, E3)</code>

### 3 Votre travail

Pour ce travail, vous allez directement manipuler la syntaxe abstraite en Prolog, sans avoir une représentation externe lue par un analyseur lexical et syntaxique, parce que la syntaxe de Prolog est suffisamment flexible pour être tolérable.

Le langage  $\mu PTS$  est un peu trop minimaliste et 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, c'est à dire un système

similaire à celui utilisé pour Haskell. Cependant, puisque  $\mu PTS$  est un langage à typage dépendant, la vérification (et l'inférence par la même occasion) de type est sensiblement plus délicate, par exemple, elle peut nécessiter de faire un peu d'évaluation.

L'implantation de  $\mu PTS$  va être faite en deux parties: une partie qui utilise le  $\mu PTS$  présenté ci-dessus (qu'on appellera *langage interne*), et une partie qui utilise une extension de ce langage (qu'on appellera *langage source*), qui amène diverses facilités pour le programmeur.

Le code fourni implémente déjà les éléments utiles de manipulation du langage interne jusqu'ici: l'opération **subst** de substitution d'une variable par un terme, la règle de normalisation, ainsi que la vérification des types selon les règles de la Figure 1.

Votre code sera donc en charge de gérer les ajouts du langage source, telles que les fonctionnalités suivantes:

- On peut omettre une annotation de type dans les définitions locales et écrire seulement **let**( $X, E_2, E_3$ ) lorsque le type de  $E_2$  peut être facilement inféré.
- On peut écrire  $E_1 \rightarrow E_2$  au lieu de **pi**( $X, E_1, E_2$ ) lorsque  $X$  n'est pas utilisé. Note: on utilise ici le sucre syntaxique de Prolog qui interprète automatiquement  $E_1 \rightarrow E_2$  comme si on avait écrit  $\rightarrow(E_1, E_2)$ .
- On peut ajouter des annotations de type explicites, de la forme  $E_1 : E_2$  qui signifie que  $E_1$  doit avoir type  $E_2$ . Là aussi,  $E_1 : E_2$  est en fait lu par Prolog comme si on avait écrit  $:(E_1, E_2)$ .
- On peut omettre l'annotation de type sur l'argument formel des fonctions (et écrire donc **fun**( $X, E_2$ )) si le *contexte* de cette expression nous indique déjà quel devrait être son type. Par exemple on peut utiliser **fun**( $x, x$ ) : ( $\text{int} \rightarrow \text{int}$ ), vu que l'annotation de type nous donne déjà le type de  $x$ .
- Plutôt que d'écrire les appels de fonction **app**( $X, E$ ) on peut écrire  $X(E)$ , et c'est même possible d'utiliser cette notation pour n'importe quel nombre d'arguments dans les appels curriés: e.g., **app**(**app**(**app**( $X, E_1$ ),  $E_2$ ),  $E_3$ ) peut s'écrire  $X(E_1, E_2, E_3)$ . Ainsi au lieu de **app**(**app**( $+$ ,  $E_1$ ),  $E_2$ ), on peut écrire  $+(E_1, E_2)$ , que Prolog nous permet d'écrire  $E_1 + E_2$ .
- On peut écrire une séquence de déclarations locales, **let**(*Decls*,  $E$ ), où *Decls* est une liste de déclarations de la forme  $X = E$ . De plus la partie  $X$  peut avoir la forme d'un appel de fonction avec des annotations de type:

```
let([add(x : int, y : int) = x + y,
    div(x : float, y : float) = x / y],
    ...)
```

qui correspond à

```
let(add, fun(x, int, fun(y, int, x + y)),
    let(div, fun(x, float, fun(y, float, x / y)),
        ...))
```

### 3.1 Arguments implicites

En plus des ajouts syntaxiques ci-dessus, le plus gros changement dans le langage source est que les arguments peuvent être *implicites*. Pour cela, on ajoute le type `forall(X, E1, E2)` (que l'on peut abrégé `forall(X, E2)` si  $E_1$  peut être inféré) qui est identique à `pi(X, E1, E2)` sauf qu'il indique que l'argument est implicite. Par exemple, on peut écrire:

```
let([identity : forall(t, (t -> t))
    = fun(x, x)],
    identity(3))
```

que l'on peut aussi écrire:

```
let([identity(x) : forall(t, (t -> t))
    = x],
    identity(3))
```

<sup>2</sup> Dans le langage interne, cela va correspondre à:

```
let(identity, pi(t, type, pi(x, t, t)),
    fun(t, type, fun (x, t, x)),
    app(app(identity, int), 3))
```

Contrairement à Haskell qui découvre ce polymorphisme automatiquement (ce qu'on appelle le *let-polymorphisme*, qui est l'élément clé de l'inférence de type à la *Hindley-Milner*), en  $\mu PTS$  c'est au programmeur d'écrire les `forall` là où ils s'appliquent.

### 3.2 Élaboration

Le lien entre le langage source et le langage interne se fait par la conversion du premier dans le second, que l'on appelle l'*élaboration*, une phase qui combine l'élimination du sucre syntaxique avec l'inférence des types.

Cette conversion se base principalement sur les règles de typage du langage source. Celles-ci sont clairement basées sur celles du langage interne, sauf qu'elles sont *bidirectionnelles* pour aider à la propagation de l'information. Plus spécifiquement il y a deux règles mutuellement récursives, chacune ne couvrant que certains cas:  $\Gamma \vdash e_1 \Rightarrow e_2$  qui, à partir de  $e_1$  et de l'environnement  $\Gamma$ , vérifie que  $e_1$  est correctement typé et calcule son type ( $e_2$ ), et  $\Gamma \vdash e_1 \Leftarrow e_2$

$$\begin{array}{c}
\frac{\Gamma(x) = e}{\Gamma \vdash x \Rightarrow e} \\
\\
\frac{\Gamma \vdash e_1 \Leftarrow \text{type} \quad \Gamma, x:e_1 \vdash e_2 \Rightarrow e_3}{\Gamma \vdash \text{fun}(x, e_1, e_2) \Rightarrow \text{pi}(x, e_1, e_3)} \\
\\
\frac{\Gamma, x:e_1 \vdash e_2 \Leftarrow e_3}{\Gamma \vdash \text{fun}(x, e_2) \Leftarrow \text{pi}(x, e_1, e_3)} \\
\\
\frac{\Gamma \vdash e_1 \Leftarrow \text{type} \quad \Gamma, x:e_1 \vdash e_2 \Leftarrow \text{type}}{\Gamma \vdash \text{pi}(x, e_1, e_2) \Rightarrow \text{type}} \\
\\
\frac{\Gamma \vdash e_1 \Leftarrow \text{type} \quad \Gamma, x:e_1 \vdash e_2 \Leftarrow \text{type}}{\Gamma \vdash \text{forall}(x, e_1, e_2) \Rightarrow \text{type}} \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow \text{pi}(x, e_4, e_5) \quad \Gamma \vdash e_2 \Leftarrow e_4}{\Gamma \vdash \text{app}(e_1, e_2) \Rightarrow e_5[e_2/x]} \\
\\
\frac{\Gamma \vdash e_1 \Leftarrow \text{type} \quad \Gamma, x:e_1 \vdash e_2 \Leftarrow e_1 \quad \Gamma, x:e_1 \vdash e_3 \Rightarrow e_4}{\Gamma \vdash \text{let}(x, e_1, e_2, e_3) \Rightarrow e_4} \\
\\
\frac{\Gamma, x:e_1 \vdash e_2 \Rightarrow e_1 \quad \Gamma, x:e_1 \vdash e_3 \Rightarrow e_4}{\Gamma \vdash \text{let}(x, e_2, e_3) \Rightarrow e_4} \\
\\
\frac{\Gamma \vdash e_2 \Leftarrow \text{type} \quad \Gamma \vdash e_1 \Leftarrow e_2}{\Gamma \vdash e_1 : e_2 \Rightarrow e_2} \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow e_2 \quad e_2 \simeq e_3}{\Gamma \vdash e_1 \Leftarrow e_3} \\
\\
\frac{\Gamma, x:e_2 \vdash e_1 \Leftarrow e_3}{\Gamma \vdash e_1 \Leftarrow \text{forall}(x, e_2, e_3)} \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow \text{forall}(x, e_2, e_3)}{\Gamma \vdash e_1 \Rightarrow e_3[e_4/x]} (\text{coercion du } \forall) \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow \text{int}}{\Gamma \vdash e_1 \Rightarrow \text{float}} (\text{coercion int\_to\_float}) \\
\\
\frac{\Gamma \vdash e_1 \Rightarrow \text{int}}{\Gamma \vdash e_1 \Rightarrow \text{bool}} (\text{coercion int\_to\_bool})
\end{array}$$

Figure 2: Règles de typage bidirectionnelles du langage source

qui reçoit non seulement  $e_1$  et l'environnement  $\Gamma$  mais aussi son type  $e_2$  et doit alors vérifier que  $e_1$  est correctement typé et qu'il a effectivement le type  $e_2$ .

La Figure 2 montre les règles en question. Votre travail sera d'implanter l'élaboration en suivant ces règles. La différence principale est que l'élaboration ne vérifie pas seulement le type, mais renvoie aussi la forme élaborée du code source, i.e. une version équivalente mais dans le langage interne, avec toutes les annotations de types et sans sucre syntaxique. Par exemple, si la vérification du code source a dû utiliser la règle “coercion `int_to_bool`”, le code renvoyé devrait appeler explicitement la fonction `int_to_bool` pur faire cette conversion. De même si la règle “coercion du  $\forall$ ” vous a permis d'inférer que `nil` a non seulement type `forall(t, type, list(t, 0))` mais aussi type `list(int, 0)`, alors il faut renvoyer le code `app(nil, int)` pour explicitement instancier l'argument implicite `t`.

Pour faciliter le travail, les opérations que je vous fournis qui opèrent sur langage interne (e.g. `subst`, `normalize`) acceptent en fait les termes de la forme `forall(X, E1, E2)` et les considèrent comme équivalents à `pi(X, E1, E2)`. De cette manière on peut utiliser les mêmes types pendant l'élaboration (où la distinction entre `forall` et `pi` est importante) et après (où la distinction n'existe plus).

### 3.3 Vérification

L'élaboration est une phase qui peut être complexe et il est tentant d'y ajouter toujours plus de bébelles, donc pour se protéger des erreurs dans cette phase, le résultat de l'élaboration est *vérifié*, ainsi la sécurité du système de type ne dépend pas de l'élaboration. Plus précisément, le code du langage interne renvoyé par l'élaboration est passé à `verify` qui implémente les règles de type du langage interne pour vérifier que le code élaboré est correctement typé.

Cette partie du code vous est aussi fournie et vous permettra donc de vous assurer que votre code fonctionne correctement.

## 4 Remise

Vous devez remettre deux fichiers: `upts.prolog` et `rapport.tex`.

## 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.

---

<sup>2</sup>Les parenthèses autour de `(t -> t)` sont nécessaires parce sinon Prolog va penser qu'on a écrit `forall((t, t) -> t)`.



- Vous pouvez bien sûr définir autant de nouvelles relations que vous désirez, mais vous ne devriez pas modifier de relations autres que celles mentionnées. Si toutefois il vous paraît nécessaire de les modifier, décrivez et justifiez la modification dans le rapport.
- Le code ne doit en aucun cas dépasser 80 colonnes.
- Vérifiez la page web du cours et le groupe de discussions, pour d'éventuels errata, et d'autres indications supplémentaires.