Travail pratique #2

IFT-2035

16 novembre 2021

1 Survol

Ce TP a pour but de vous familiariser avec le langage Prolog, ainsi qu'avec l'inférence de types. Comme pour le TP précédent, 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 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 LaTeX exclusivement (compilable sur `ens.iro`) 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 H2035

Vous allez écrire un programme Prolog qui va manipuler des expressions du langage du futur H2035. C'est un langage fontionnel typé statiquement, avec surcharge et polymorphisme paramétrique similaire à Haskell. Votre travail sera d'implanter la phase d'élaboration qui fait l'inférence de types, l'élimination du sucre syntaxique, et la conversion vers DeBuijn, ainsi que l'évaluateur.

La syntaxe du langage est la suivante :

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{list}(\tau) \mid \tau_1 \to \tau_2 \mid \alpha \mid \mathsf{forall}(\alpha, \tau) \\
e & ::= & c \mid x \mid \mathsf{lambda}(x, e) \mid \mathsf{app}(e_1, e_2) \mid x(e_1, .., e_n) \mid \\
 & & \mathsf{if}(e_1, e_2, e_3) \mid \mathsf{let}(\mathit{rdecls}, e) \mid (e : \tau) \mid ? \\
\mathit{rdecls} & ::= & \mathit{rdecl} \mid \mathit{rdecl}, \mathit{rdecls} \\
\mathit{rdecl} & ::= & \mathit{decl} \mid [\mathit{decls}] \\
\mathit{decls} & ::= & \mathit{decl} \mid \mathit{decl}, \mathit{decls} \\
\mathit{decl} & ::= & x = e \mid x(x_1, .., x_n) = e
\end{array}
$$

où forall est le type donné aux expressions polymorphes. c représente n'importe quelle constante prédéfinie, cela inclut par exemple les nombres entiers et les opérations sur les listes ; lambda et app définissent et appellent les fonctions, et let introduit un ensemble de déclarations locales, où celles qui sont placées entre [...] peuvent être (mutuellement) récursives (d'où le r de *rdecl*).

Comme on le voit, la syntaxe n'inclut presqu'aucune annotation de types, car les types seront complètement inférés, tout comme en Haskell. La seule annotation de type est de la forme $(e : \tau)$, qui permet d'aider l'inférence lorsque nécessaire ou de vérifier que l'inférence trouve le bon type. En plus de cela, la syntaxe inclut aussi le terme $?$ qui fait l'inverse de l'inférence de type, c'est à dire que le code sera inféré sur la base du type attendu.

La syntaxe du langage repose sur la syntaxe des termes de Prolog, donc par exemple, $x = e$ peut aussi s'écrire $=(x, e)$, et $+(e_1, e_2)$ peut s'écrire $e_1 + e_2$, et votre code n'a pas à s'en soucier car ces différences n'affectent que l'analyseur syntaxique, qui est déjà fourni par Prolog lui même.

Le langage inclut parmis ses primitives des opérations sur les listes dont le nom dérive de ceux utilisés en Lisp : *cons*, *car*, *cdr*, *nil*. À l'exécution, ces listes sont représentées par des listes Prolog.

3 Élaboration

Après l'analyse syntaxique (qu'on ne voit pas dans le code vu qu'elle est faite par Prolog), le code passe par une phase d'*élaboration* qui va implémenter certaines fonctionnalités du langages : l'élimination du sucre syntaxique, l'inférence des types (et du code), et la "désambiguation" des identificateurs.

3.1 Sucre syntaxique

H2035 utilise un peu de sucre syntaxique que la phase d'élaboration va devoir éliminer. Plus précisément, les règles d'équivalence suivantes devront être utilisées :

$$
\begin{array}{rcl}
f(e_1, ..., e_n) & \Longrightarrow & \mathsf{app}(...\mathsf{app}(f, e_1)..., e_n) \\
f(x_1, ..., x_n) = e & \Longrightarrow & f = \mathsf{lambda}(x_1, ...\mathsf{lambda}(x_n, e)...) \\
\mathsf{let}(\mathit{decl}, e) & \Longrightarrow & \mathsf{let}([\mathit{decl}], e) \\
\mathsf{let}(d_1, ..., d_n, e) & \Longrightarrow & \mathsf{let}(d_1, ...\mathsf{let}(d_n, e)...)
\end{array}
$$

3.2 Surcharge

H2035, contrairement à Psil, Slip, et Haskell, offre ce qu'on appelle la *surcharge*, en anglais *overloading*. Cette fonctionnalité permet d'avoir plusieurs définitions actives pour un même identificateur. Habituellement, lorsqu'un identificateur est défini plusieurs fois, la définition la plus proche cache les autres (en anglais, on utilise le terme *shadowing*), mais en H2035 ce n'est pas le cas. Ainsi, chaque usage d'un identificateur introduit une ambiguïté s'il y a plusieurs définitions qui y corresponent. L'élaboration va résoudre ces ambiguïtés en utilisant les informations de types. Exemple :

```
let(x = 2,
    x(i) = i + x,
    x(l) = cons(3,l),
    x + x(car(x(nil))))
```

Ces trois définitions de x ont chacune un type différent, et l'élaboration va donc pouvoir utiliser cette information de typage pour savoir que dans l'expression finale, chacun des trois x fait référence à une autre définition.

L'expression ? généralise cette idée au cas où le nom de la variable n'est pas spécifié et l'élaboration peut utiliser n'importe quelle variable dont le type concorde. Bien sûr, on s'attend à ce que le programmeur n'utilise ce terme que dans le cas où il n'y a qu'une seule variable du bon type.

Le code renvoyé par le prédicat *elaborate* (et attendu par *eval*) n'a plus de telles ambiguïtés, car il utilise ce qu'on appelle les *indices de de Bruijn* plutôt que des variables avec des noms. Plus précisément, il doit utiliser la syntaxe suivante :

$$e ::= n \mid \mathsf{var}(i) \mid \mathsf{lambda}(e) \mid \mathsf{app}(e_1, e_2) \mid \mathsf{if}(e_1, e_2, e_3) \mid \mathsf{let}([es], e)$$
$$es ::= e \mid e, es$$

où $\mathsf{var}(i)$ est une référence à la i-ème variable de l'environnement (en comptant comme 0 la variable la plus récemment ajoutée). Ainsi le code ci-dessus deviendra :

```
let([2],
    let([lambda(app(app(var(Np1), var(0)), var(2)))],
        let([lambda(app(app(var(Nc), 3), var(0)))],
            app(app(var(Np), var(2)), ...))))
```

où `Np1`, `Nc`, et `Np2` sont les entiers correspondants à la position (ou la profondeur, selon comment on le regarde) respective de *cons*, et $+$ dans l'environnement. Notez que `Np1` et `Np2` font référence à $+$ mais ne seront pas égaux ; plus spécifiquement, `Np2` $= 1+$`Np1` vu que l'environnement au deuxième usage du $+$ contient une variable supplémentaire (le 3^e x) par rapport à l'environnement du premier usage du $+$.

3.3 Inférence de types

H2035 utilise un typage statique similaire à celui de Haskell, où les types sont inférés. Dans un premier temps, je vous recommande de faire une inférence simple, sans polymorphisme (ni récursion mutuelle), puis dans un deuxième temps vous ajouterez

$$\frac{}{\Gamma \vdash n \ : \ \mathsf{int}} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x \ : \ \tau} \qquad \frac{\Gamma \vdash e \ : \ \mathsf{bool} \quad \Gamma \vdash e_t \ : \ \tau \quad \Gamma \vdash e_e \ : \ \tau}{\Gamma \vdash \mathsf{if}(e, e_t, e_e) \ : \ \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \mathsf{lambda}(x, e) \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 \ : \ \tau_1}{\Gamma \vdash \mathsf{app}(e_1, e_2) \ : \ \tau_2}$$

$$\frac{\Gamma \vdash e \ : \ \tau}{\Gamma \vdash (e : \tau) \ : \ \tau} \qquad \frac{x{:}\tau \in \Gamma}{\Gamma \vdash ? \ : \ \tau} \qquad \frac{\Gamma, x_1{:}\tau_1 \vdash e_1 \ : \ \tau_1 \quad \Gamma, x_1{:}\tau_1 \vdash e \ : \ \tau}{\Gamma \vdash \mathsf{let}([x_1 = e_1], e) \ : \ \tau}$$

FIGURE 1 – Règles de typage simple

$$\frac{}{\tau \subset \tau} \qquad \frac{\sigma[\tau'/\alpha] \subset \tau}{\mathsf{forall}(\alpha, \sigma) \subset \tau} \qquad \frac{x{:}\sigma \in \Gamma \quad \sigma \subset \tau}{\Gamma \vdash x \ : \ \tau_2}$$

$$\frac{\begin{array}{c}\forall i \quad \Gamma, x_1{:}\tau_1, ..., x_n{:}\tau_n \vdash e_i \ : \ \tau_i \quad \sigma_i = \mathsf{gen}(\Gamma, \tau_i) \\ \Gamma, x_1{:}\sigma_1, ..., x_n{:}\sigma_n \vdash e \ : \ \tau\end{array}}{\Gamma \vdash \mathsf{let}([x_1 = e_1, ..., x_n = e_n], e) \ : \ \tau}$$

$$\frac{\{t_1, ..., t_n\} = \mathsf{fv}(\tau) - \mathsf{fv}(\Gamma)}{\mathsf{gen}(\Gamma, \tau) = \mathsf{forall}(t_1, ...\mathsf{forall}(t_n, \tau)..)}$$

FIGURE 2 – Règles additionnelles de typage polymorphe

ce qu'on appelle le *let-polymorphisme*, en utilisant l'algorithme d'inférence de types de Hindley-Milner.

Les règles de typage sans polymorphisme sont données en Fig. 1. Dans ces règles, le jugement $\Gamma \vdash e \ : \ \tau$ signifie que l'expression e a type τ dans un contexte Γ qui donne le type de chaque variable dont la portée couvre e.

Comme vous pouvez le voir, la règle du let n'accepte pas que le let déclare simultanément plus qu'un identificateur, et n'autorise donc pas la récursion mutuelle. Vous pouvez aussi remarquer que ces règles présument que le sucre syntaxique a déjà été éliminé, c'est pourquoi il n'y a pas de règle par exemple pour la forme $x(e_1, .., e_n)$.

3.3.1 Récursion mutuelle et inférence du polymorphisme

La deuxième étape introduit le let-polymorphisme de Hindley-Milner qui va donc automatiquement inférer quand une définition est polymorphe, ainsi que spécialiser automatiquement une définition polymorphe chaque fois qu'elle est utilisée. Cette étape ajoute aussi les définitions récursives.

Les nouvelles règles de typage de la deuxième étape sont données en Fig. 2. Ces règles remplacent la règle de typage des variables, et la règle de typage du let. Dans ces nouvelles règles, σ est utilisé pour indiquer un type qui peut être *polymorphe* (i.e.

avec un forall), alors que τ représente un type *monomorphe* (i.e. sans forall).

Dans la règle de typage des variable, le $\sigma \subset \tau$ indique que τ est une instance de σ, par exemple, σ pourrait être forall$(\alpha, \alpha \to \alpha))$, et τ pourrait être int $\to$ int qui est l'instance correspondant au choix de int pour α. C'est donc dans cette règle que les forall sont *éliminés*.

La règle de typage du let utilise une fonction auxiliaire gen qui va *généraliser* une définition, en découvrant son polymorphisme. Cela fonctionne simplement en regardant dans le type τ de la définition quelles sont les variables non-instanciées (ou libres : fv veut dire "free variables") : celles-ci peuvent être généralisées (sauf si elles apparaissent aussi dans le contexte Γ), donnant alors un type polymorphe. C'est donc cette règle qui *introduit* le forall.

Cette nouvelle règle du let autorise non seulement l'introduction de plusieurs définitions simultanées ($x_1 ... x_n$), mais autorise aussi la récursion mutuelle. Cela se voit dans le fait que e_i est typé dans un environnement qui n'est pas limité à Γ mais inclus aussi les identificateurs $x_1 ... x_n$, ce qui autorise donc e_i à faire des références récursives non seulement à x_i mais aussi à n'importe lequel des autres identificateurs définis par ce let.

Détail important pour la règle du let : si on a par exemple les déclarations $y = x, x = 1$ et on infère le type de y avant d'inférer le type de x, le type de y sera incomplet tant que le type de x ne sera pas inféré. Pour cette raison, il est important d'inférer le type de tous les e_i *avant* de les passer à gen, faute de quoi, gen risque de penser que y est polymorphe.

4 Votre travail

Votre travail consiste à compléter le code des prédicats *elaborate* et *eval*. Le code qui vous est fourni contient déjà un squelette de ces prédicats ainsi que quelques autres prédicats qui vous seront utiles principalement pour la deuxième phase :

— *instantiate* : implémente la règle $\sigma \subset tau$.
— *freeIvars* : implémente la fonction fv.
— *generalize* : fait le travail de gen.

5 Remise

Vous devez remettre deux fichiers, `h2035.pl` et `rapport.tex`. Ces fichiers seront remis sous forme électronique, par l'intermédiare de Moodle/StudiUM.

6 Détails

La note est basée d'une part sur des tests automatiques, d'autre part sur la lecture du code, ainsi que sur le rapport. Le critère le plus important, et que votre code doit se comporter de manière correcte. En règle générale, une solution simple est plus souvent correcte qu'une solution complexe. Ensuite, vient la qualité du code : plus c'est simple, mieux c'est. S'il y a beaucoup de commentaires, c'est généralement un symptôme

que le code n'est pas clair ; mais bien sûr, sans commentaires le code (même simple) et souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.

Les points seront répartis comme suit : 25% sur le rapport, 25% sur les tests automatiques, 25% sur le code de l'élaboration, et 25% sur le code de l'évaluateur.

— 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.
— 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.