Projet – inférence de types et rendu final

à faire pour le samedi 8 mai à 23h59

Pour les binômes concernés, il s'agit d'ajouter l'inférence de types à votre fouine : on commence par vérifier si le programme est typable. Si c'est le cas, on l'exécute. Sinon, on proteste et on s'arrête.

1 Intermédiaires : inférence de types monomorphe

1.1 Fonctions (et entiers et booléens)

Vous avez vu en cours de Théorie de la Programmation comment fonctionne l'inférence de types en Fun. Le but est d'adapter et d'étendre cette approche à fouine, en commençant par la partie "intermédiaires" du rendu 2 : entiers, booléens et fonctions (voir plus bas pour les autres aspects du langage fouine).

L'inférence de types se fait en deux étapes :

1. on engendre des *contraintes de typage* en parcourant le programme que l'on analyse ;
2. ces contraintes constituent un *problème d'unification*, que l'on résoud.

Mise en place. Pour mettre en œuvre l'inférence de types, il vous faudra :

- Définir ce que sont les types, et les contraintes de types. Replongez-vous dans le cours de théorie de la programmation, vous y verrez en particulier que les types contiennent des variables de types (notées $X, Y, Z, \ldots$), qui sont nécessaires pour exprimer les contraintes de types.
- Décider comment gérer l'information de typage, notamment s'agissant de l'affichage qui est demandé (cf. ci-dessous).
- Mettre en place les divers fichiers (sans surprise, un pour l'algorithme d'unification, l'autre pour l'inférence).

  Le code devrait être organisé de telle manière que cela soit transparent vis-à-vis de l'approche utilisée pour implémenter l'algorithme d'unification (naïve ou sophistiquée).

Ce travail de mise en place est à faire à deux, il permettra de vous répartir l'implémentation de l'inférence de types.

1.2 Programmer l'unification

Exploitez le TP numéro 10 du cours de théorie de la programmation, et le cours qui va avec.

Si l'unification échoue, on arrête tout.

Si l'unification réussit, on passe à la suite, à savoir on lance l'interprète.

1.3 Comportement, options

Sans options, `fouine` type le programme, n'affiche rien si le programme est typable, et l'exécute.

Si le programme n'est pas typable, affichez un message d'erreur de votre cru, avec des indications sur l'origine du problème.

Programmez aussi les options suivantes :

`-notypes` : désactive l'inférence de types, et interprète directement les programmes.

`-showtypes` : affiche le type inféré pour toutes les déclarations en surface, l'expression principale étant désignée par `-`. Ainsi, sur le programme

```
let f x = let y = x+1 in y in
let a = 12 in
  f a
```

`fouine -showtypes` répondra

```
f :  int -> int
a :  int
- :  int
```

Les types contenant des variables. Si on lance l'inférence sur `fun x y -> x`, il n'y a pas de problème, c'est typable.

Que faire au moment d'afficher le type ? Vous pouvez par exemple afficher quelque chose qui ressemble à ce qu'afficherait Caml (`'a -> 'b -> 'a`). Ou alors afficher, plus simplement, quelque chose comme `T0 -> T1 -> T0`.

Il est bon de garder à l'esprit que les `'a`, `'b` de Caml correspondent à un système polymorphe, alors que le système que vous implémentez dans cette partie est monomorphe. Vous pouvez vous en convaincre en contemplant le programme

```
let id x = x in fun f -> f (id 3) (id false)
```

et en comprenant pourquoi vous ne savez pas le typer.

1.4 Extensions

Une fois que l'inférence a été testée pour les fonctions, traitez les extensions suivantes, dans cet ordre :

- Références : ajout de `ref` sur les types, et de `unit`.
- Tuples : ajout de `*` sur les types.
- Listes : ajout de `list`
- Exceptions : a priori il n'y a rien à ajouter dans le langage des types, mais il faut pouvoir typer des programmes qui utilisent `raise` et `try.. with`.

  ajout du type `exn`, si vous traitez les exceptions comme des valeurs (cf. la différence entre `let f k = raise (E k)` et `let f k = E k`).

2 Avancés : le polymorphisme

On vous demande ici d'adapter votre code pour traiter le polymorphisme (les types avec des `'a` en Caml).

Important : on veut que vous ayez les deux versions de l'inférence de types : une option `-monotypes` permettra d'exécuter l'inférence de types monomorphes (sans option, c'est la version polymorphe qui tournera).

À vous de réfléchir afin de combiner les deux versions de l'inférence de types sans trop dupliquer le code.

Pour prendre en compte le polymorphisme, les "briques de base conceptuelles" sont les mêmes, mais il faudra adapter un certain nombre de choses.

Schémas de types. Outre les types, on manipule aussi des **schémas de types**.

$$\tau \quad ::= \quad \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \alpha \qquad\qquad \rho \quad ::= \quad \tau \mid \forall \alpha.\rho$$

(les α ci-dessus sont les X de l'inférence de types telle que vue en cours de théorie de la programmation : des variables de types — c'est juste une question de notations).

Notez qu'on a une grammaire à deux étages : on n'ajoute pas directement la quantification sur les types dans la grammaire des types. Ainsi,

$$\forall \alpha.\alpha \to (\forall \beta.((\beta \to \beta) \to \texttt{int})) \to \alpha$$

n'est pas un schéma de types (car la quantification sur β est sous une flèche).

Généralisation, spécialisation. La quantification universelle permet de généraliser un type en le transformant en un schéma de types. Intuitivement, cela permet de donner le type `'a -> 'a` à la fonction `fun x -> x`.

Il faut bien comprendre que tout se passe dans les `let` : le type de `fun x -> x` *n'est pas* `'a -> 'a`. En effet, le polymorphisme n'est à l'œuvre que dans les déclarations. Par exemple, dans `let f = fun x -> x in e`, le type associé à `f` est `'a -> 'a` pour typer `e`.

Par conséquent, quand on infère le type de `fun x -> e`, on fait comme dans le cas monomorphe. En revanche, pour inférer le type de `let x = e1 in e2`,

- on infère le type de `e1` : ça donne un type ;
- on **généralise** ce type : ça donne un schéma de types, mettons ρ, qui est associé à `x` pour inférer le type de `e2` ;
- l'environnement de typage qui sert pour inférer le type de `e2` est donc une structure associant un *schéma de types* à chaque identifiant de variable ;
- au moment de typer un usage de `x` dans `e2`, on **spécialise** le schéma ρ, qui est instancié avec de nouvelles variables de types. On prend pour ce faire de nouvelles variables de types à chaque fois : c'est ici qu'on met en œuvre le polymorphisme.

Ce mécanisme de généralisation puis spécialisation permet d'associer une instanciation différente à chaque usage de `x` dans `e2` : c'est le polymorphisme, au sens où on peut utiliser `x` selon des types différents, mais tous compatibles au sens où ils proviennent d'un même schéma de types.

Algorithme d'unification : ne pas le changer, mais l'appeler plus souvent. Au vu de ce qui est décrit ci-dessus, on peut observer que :

- l'unification est utilisée, comme précédemment, sur les types, et pas sur les schémas de types ;
- alors que dans le cas monomorphe on procède en deux phases globales (d'abord engendrer le problème d'unification, puis le résoudre), en présence de polymorphisme, on résoud des problèmes d'inférence au fur et à mesure qu'on parcourt le programme : à chaque `let x = e1 in e2`, on infère le type de `e1` (ce qui fait intervenir un appel à l'unification) avant de passer à l'inférence pour `e2`.

Formellement. On donne ci-dessous les règles de typage.

Les Γ sont des ensembles de couples notés $x : \rho$.

On note $\tau_0[\widetilde{\tau}/\widetilde{\alpha}]$ le type obtenu en remplaçant les variables de types $\widetilde{\alpha}$ par les types $\widetilde{\tau}$; le résultat est un type.

$\widetilde{\alpha} \cap \Gamma = \emptyset$ signifie que α n'apparaît pas libre dans Γ.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{e_1 + e_2 \vdash \texttt{int}} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash x : \tau_0[\widetilde{\tau}/\widetilde{\alpha}]}\ \Gamma(x) = \forall \widetilde{\alpha}.\tau_0 \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \forall \widetilde{\alpha}.\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \widetilde{\alpha} \cap \Gamma = \emptyset$$

Rien de surprenant s'agissant des règles pour les fonctions ; notez toutefois que lorsqu'on écrit $\Gamma, x : \tau_1$ dans la règle pour typer une fonction, on ajoute l'hypothèse $x : \tau_1$, où τ_1 est vu comme un schéma de types.

Tout se passe dans les règles pour la déclaration (où l'on fait la généralisation) et pour la variable (où l'on fait la spécialisation).

Attention à la généralisation. Généraliser un type en un schéma de types signifie rajouter des $\forall$ pour quantifier sur les variables de types. On ne généralise cependant pas par rapport à toutes les variables, mais uniquement par rapport à celles qui n'apparaissent pas ailleurs dans l'environnement de typage.

Voici une illustration du risque :

$$\frac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha} \qquad \dfrac{}{x : \alpha, y : \forall \alpha.\alpha \vdash y : \beta}}{x : \alpha \vdash \texttt{let}\ y = x\ \texttt{in}\ y : \beta}}{\emptyset \vdash \texttt{fun}\ x \rightarrow \texttt{let}\ y = x\ \texttt{in}\ y : \alpha \to \beta}$$

Dans la dérivation ci-dessus, au moment d'appliquer la règle pour la déclaration, on généralise le type que l'on associe à y, alors que "α traîne encore". Cela permet de déduire le type $\alpha \to \beta$ pour ce qui est fondamentalement l'identité (pourquoi est-ce une catastrophe ?).

Cela se traduit par la condition $\widetilde{\alpha} \cap \Gamma = \emptyset$ pour contrôler la généralisation dans la règle de typage pour les déclarations.

Attention aux références (en présence de polymorphisme). Si on combine le polymorphisme et les références sans faire attention, on risque de typer le programme suivant :

```
let f = ref (fun z -> z) in
    begin
      f := fun x -> x+1;
      !f false
    end
```

La solution est de restreindre la généralisation dans le typage d'une déclaration :

- On ne généralise pas le type inféré pour un `ref`. Autrement dit, lors de l'inférence pour `let x = ref e in e'`, on infère un type τ pour `ref e`, et on passe à l'inférence pour `e'` en associant τ (vu comme schéma de types) à `x`.
- On ne généralise pas le type inféré pour une application : même chose que ci-dessus pour `let x = e1 e2 in e'`. La raison est qu'on a peur que l'évaluation de `e1 e2` ne fabrique des références, donc soit "risquée".

  Vous pouvez vérifier ce qui se passe en typant `(fun x -> x) (fun z -> z)` (l'identité *appliquée* à elle-même) dans Caml : on récupère un type monomorphe (signalé par un _).

Il existe des manières plus fines de restreindre la généralisation en présence de références, vous pouvez chercher cela sur internet.

3 Le rendu 4 : options, etc.

Options. Mêmes options que pour les rendus précédents, avec en plus les options `-notypes` et `-showtypes` pour les intermédiaires, et aussi `-monotypes` pour les avancés.

Des tests. Une petite archive avec quelques fichiers de tests est fournie sur la page du portail des études (rubrique "quatrième rendu").

Comme toujours, prévoyez des fichiers de test à vous qui illustrent les mécanismes à l'œuvre dans l'inférence de types. À la fois des programmes qui typent et des programmes qui ne typent pas, avec des commentaires sur les tests que vous fournissez.

Indiquez s'il y a des tests qui échouent, en raison de bugs pas encore résolus.

Le rendu. Vous pouvez vous reporter aux rendus 1 à 3 : l'idée est la même en ce qui concerne la propreté du rendu, les explications, etc. N'oubliez pas d'expliquer dans le README comment vous avez procédé pour vous répartir le travail pour implémenter l'inférence de types.

4 Tout le monde : le README qui termine le projet.

Le fichier README sera à préparer dans le même esprit que ce qui a été fait pour les rendus précédents. En plus du retour sur ce rendu 4, ajoutez quelques mots sur l'ensemble du projet pour la partie fouine :

- ce qui a marché, ce qui n'a pas marché, au long des rendus 2, 3 et 4.
- Les bugs importants que vous avez rencontrés : ceux que vous avez pu corriger, ceux que vous n'avez pas été en mesure de corriger.
- Toute autre remarque sur votre fouine.

5 ...et ça n'est pas tout à fait terminé : tout le monde, rapport+présentation

En plus du rendu, vous devrez faire un bref compte-rendu sur le projet. On vous demande d'envoyer avec le dernier rendu un court rapport (entre 2 et 5 pages), rédigé en LaTeX, qui sera accompagné d'une présentation de 4 minutes, *avec des transparents,* lors de la séance du 11 mai (mais tout est à rendre pour le 8 mai).

Vous trouverez sur la page www du cours des exemples simples de fichiers desquels vous pourrez partir. *Lisez ces fichiers,* pour savoir comment s'en servir, et comment engendrer un fichier au format pdf.

Mettez rapport et présentation dans un sous-répertoire dédié, et ne les nommez pas `rapport.pdf` et `presentation.pdf`.

Passage obligé. Il vous faudra nécessairement mentionner le nom de Peter Landin dans votre rapport, en faisant référence à son article sur les 700 langages, par le biais d'une citation comme celle-ci [1].

Pour cela, il vous faudra éditer le fichier `ex-biblio.bib`, en y insérant les données au bon format (le format BibTex) pour l'article de 1966. Comment trouver ces données ? Par exemple sur le site DBLP. Une fois trouvées les informations, copiez-collez-les directement dans le fichier `.bib` (des sites comme DBLP permettent d'exporter les données au format BibTex).

Rien de bien difficile dans tout cela, mais il faut l'avoir fait au moins une fois avant de commencer son stage de L3.

Pourquoi ce compte-rendu ? Soyons lucides, l'objectif essentiel de ce compte-rendu est de s'assurer que vous avez une certaine familiarité avec les outils mis en jeu. Du point de vue du contenu, ne perdez pas de temps à raconter ce que tout le monde sait (ce qu'est une fouine, qu'est-ce qu'un environnement, quelles options propose votre programme). Concentrez-vous sur ce qui est propre à votre travail sur le programme `fouine` : comment il est fabriqué, ce qu'il sait bien faire, ce qu'il devrait savoir faire, ce qui a été facile/difficile à réaliser au cours du semestre, etc. (bref, rendez-vous compte vous-même de ce qu'il est pertinent d'exposer).

Cet aspect des choses ne comptera pas pour beaucoup dans l'évaluation du projet, mais jouez quand même le jeu : encore une fois, cela vous sera utile, entre autres pour l'évaluation du stage de L3.

References

[1] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.