Travail pratique #2

IFT-3065/6232

February 27, 2017

¡¡ Dû le 22 mars à 23h59 !!

1 Survol

Ce TP vise à vous familiariser avec l'implantation d'optimisations simples par réécriture. Les étapes de ce travail sont les suivantes:

1. Lire et comprendre cette donnée.
2. Lire, trouver, et comprendre les parties importantes du code fourni. Cela prendra probablement une partie importante du temps total.
3. Écrire du code OCaml et Typer, et faire des mesures de performance.
4. 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 `frontal.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. Si un étudiant ne trouve pas de partenaire, il doit me contacter au plus vite. Des groupes de 3 ou plus sont **exclus**.

2 Typer

Vous allez travailler sur l'implantation du langage expérimental Typer.

Ce langage est de la même famille que OCaml et Haskell. C'est un langage purement fonctionnel, comme Haskell, mais avec un ordre d'évaluation classique (i.e. appel par valeur) comme OCaml. Son système de type est aussi un peu différent:

- Les types peuvent être manipulés comme des valeurs.
- L'inférence de types n'est pas vraiment celle de Hindley-Milner. Pour cette raison, les paramètres de type, qui sont habituellement complètement implicites en Haskell et OCaml sont plus explicites.

Plus spécifiquement, une fonction comme la fonction identité s'écrit en Typer:

$$\begin{array}{l} \mathit{identity} : (a : \mathit{Type}) \equiv> a \to a; \\ \mathit{identity} = \mathsf{lambda}\ a \equiv> \mathsf{lambda}\ x \to x; \end{array}$$

plutôt que (en Haskell):

$$\begin{array}{l} \mathit{identity} :: a \to a \\ \mathit{identity} = \backslash x \to x \end{array}$$

Cependant, la flèche $\equiv>$ indique que c'est un argument implicite (et effaçable, c'est à dire qu'il n'existera pas à l'exécution), donc l'appel à cette fonction n'a pas besoin de fournir l'argument. Donc tout comme en Haskell, l'appel s'écrit:

$$\mathit{identity}\ 3$$

plutôt que

$$\mathit{identity}\ \mathit{Int}\ 3$$

Ceci dit, si cela s'avère nécessaire ou désiré, l'argument implicite peut être fourni de manière explicite lors de l'appel en utilisant la syntaxe suivante:

$$\mathit{identity}\ (a := \mathit{Int})\ 3$$

Une autre différence par rapport à Haskell, est que Typer n'autorise pas les boucles infinies. L'implémentation fournie n'impose à vrai dire pas cette restriction, mais votre optimisateur a la liberté de faire comme si les boucles infinies n'existent pas.

Le fichier `test.typer` contient quelques exemples de code pour vous montrer la syntaxe du langage.

3 Optimisation de Typer

Vous allez devoir implanter une ou plusieurs phases d'optimisation dans l'implantation du langage expérimental Typer.

À l'heure actuelle, l'implémentation passe par les phases suivantes:

1. Analyse lexicale
2. Analyse syntaxique
3. Élaboration: expansion des macros et inférence des types.
4. *Erasure*: élimination des annotations de types et des arguments effaçables, qui sont d'ailleurs généralement aussi des types.
5. Interprétation.

Votre but est de rendre l'exécution du code Typer plus rapide, quitte à ralentir la "compilation" de ce code. Vous ferez cela en ajoutant des phases d'optimisation entre le *erasure* et l'interprétation.

Si vous le voulez, vous pouvez aussi modifier la représentation du code dans l'interpréteur (par exemple, en permettant aux fonctions de prendre plusieurs arguments, de manière à pouvoir implanter le *uncurrying*).

4 Donnée du travail

Le code sur lequel vous allez travailler est disponible par Git. Vous pouvez le télécharger avec la commande suivante:

```
git clone http://www.iro.umontreal.ca/~monnier/3065/typer
```

Vous y trouverez le code de Typer, mais aussi un fichier `test.typer` qui montre comment utiliser le langage. Vous pouvez l'utiliser comme suit:

```
% cd .../typer
% make
% ./_build/typer --batch test.typer
```

Qui devrait simplement afficher un nombre à virgule flottante qui est le nombre de secondes utilisées par le programme de test. Vous pouvez aussi passer l'argument `--debug` pour que Typer vous montre le code renvoyé par *erasure* (et donc passé à l'optimisateur).

Le code actuel de l'optimisateur (qui ne fait rien) est dans `src/optimize.ml`. À vous de le développer.

Autres éléments intéressants pour vous:

- `emacs/typer-mode.el`: permet la coloration et l'indentation automatique du code Typer dans Emacs.
- `btl/builtins.typer` et `btl/pervasive.typer`: Les fichiers qui contiennent les types et fonctions prédéfinis dans Typer.

5 À faire

La travail a plusieurs aspects:

- Écriture d'optimisations, telles qu'élimination de code mort, *inlining*, propagation de constantes, *constant folding*, ... Cela se fait en OCaml.
- Écriture de code Typer pour tester vos optimisations.
- Mesures de performance pour quantifier l'effet de vos optimisations sur vos tests.

Chacun de ces trois aspects est important. Les mesures de performance sont parfois difficiles et frustrantes car elles peuvent varier beaucoup d'une exécution à l'autre, donc il faut les répéter plusieurs fois.

Pour les tests, vous pouvez commencer avec le `test.typer` fourni, bien sûr, mais vous voudrez y faire des changements vu que le code fourni n'offre pas forcément beaucoup d'opportunités pour faire briller vos phases d'optimisation.

Ce qui vous est demandé plus spécifiquement est:

- Implanter au moins 2 types d'optimisations pour ift-3065 et 3 types d'optimisations pour ift-6232.
- Écrire du code Typer qui montre bien l'effet des optimisations.
- Mesurer l'effet de chacune de vos optimisations, ainsi que de leur combinaison.

5.1 Remise

Pour la remise, vous devez remettre 3 fichiers:

- un fichier `rapport.tex` qui contient votre rapport au format LaTeX.
- Un fichier `test.typer` qui contient le code qui vous avez utilisé pour vos mesures.
- Un fichier `optim.patch` qui inclut tous vos changements au code de Typer. Vous pouvez générer un tel patch avec la commande:

  ```
  git format-patch --stdout origin/master >code.patch
  ```

  Avant de lancer cette commande, il faut bien sûr faire un "commit" de vos changements, par exemple avec:

  ```
  git commit -a -m "Prions Emacs que ça marche"
  ```

6 Détails

- La note sera divisée comme suit: 7 points pour les optimisations, 2 points pour `test.typer`, 3 points pour la qualité des mesures, 3 points pour le reste 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.
- La note sera 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. 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) est souvent incompréhensible. L'efficacité de votre code est sans importance, sauf s'il utilise un algorithme vraiment particulièrement ridiculement inefficace.