Travail pratique #2

IFT-3065/6232

March 30, 2017

¡¡ Dû le 26 avril à 23h59 !!

1 Survol

Ce TP vise à vous familiariser avec les phases intermédiaires de compilation d'un compilateur de language de haut niveau. 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 C.
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 Compilation de Typer vers C

Vous allez travailler sur l'implantation du langage expérimental Typer, le même que dans le TP2, en partant du même langage intermédiaire *elexp*, mais cette fois-ci vous allez devoir le compiler vers C.

Pour cela, il vous faudra:

- Implanter la conversion de fermetures, y compris le *hoisting* qui se fait généralement en même temps.
- Décomposer le *Case* en une opération de flot de contrôle suivie d'une série d'extractions des différents champs de l'objet.
- Choisir une représentation des fermetures et des datatypes.
- Générer du code C valide.
- Écrire une petite libraire C de *runtime support*.
- Les élèves de ift-6232 doivent en plus implémenter une phase de *uncurrying*.

Pour simplifier le travail, il ne vous est pas demandé de gérer la récupération des objets, donc votre code peut allouer de la mémoire dynamiquement et n'a pas besoin d'implanter un GC.

3 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 (légèrement modifié par rapport à celui du TP2), ainsi qu'un fichier `src/cexp.ml` qui suggère une nouvelle représentation intermédiaire que vous pouvez utiliser entre *elexp* et le code C. Vous êtes libres de modifier ce fichier ou même de ne pas l'utiliser.

3.1 Remise

Pour la remise, vous devez remettre 3 fichiers:

- un fichier `rapport.tex` qui contient votre rapport au format LaTeX.
- 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"
  ```

4 Détails

- La note sera divisée comme suit: 4 points pour le rapport, 3 points pour le *uncurrying* (pour ift-6232), le reste pour la conversion de fermetures et la génération du code C.
- 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.