Moteur d'optimisation de programmes fonctionnels

Contact enseignant: Pierre-Evariste Dagand
Etudiantes: Laure Runser et Maryline Zhang (M1 LP)

Introduction générale

La programmation fonctionnelle offre la possibilité d'écrire des programmes de haut-niveau. Les programmeurs fonctionnels peuvent ainsi communiquer plus naturellement leurs intentions à la fois au compilateur mais également aux mainteneurs futurs, dans un style efficace et épuré proche des mathématiques. En conséquence, un compilateur de langage fonctionnel doit être en mesure de transformer agressivement les programmes afin de permettre leur exécution efficace.

Dans ce projet long, nous proposons d'étudier et d'implémenter un moteur d'optimisation pour un fragment du langage Core de GHC. Il s'agit d'un langage pure (sans effet de bord), offrant des types algébriques, du typage explicite de second ordre et une sémantique paresseuse.

L'optimiseur fut le sujet de la thése de Andre Santos ainsi que 3 publications de conférence:

- Let-floating: Moving Bindings to Give Faster Programs; Peyton Jones, Partain et Santos
- A Transformation-Based Optimiser for Haskell, Peyton Jones et Santos
- Secrets of the Glasgow Haskell Compiler inliner; Peyton Jones et Marlow

L'objectif est de réduire le nombre de déclarations (`let`) ainsi que le nombre de `match` tout en évitant une explosion combinatoire de la taille du code généré.

L'optimiseur fonctionne de source à source, en préservant le typage : il consomme un programme Core bien typé et produit un programme Core bien typé.

Objectifs

L'objectif de ce projet long est d'apprendre les différentes techniques (et astuces) d'implémentation des langages fonctionnels. Il s'agira en particulier de modéliser une syntaxe avec lieurs (`let`, `lambda`), d'implémenter un algorithme de typage et d'effectuer des transformations symboliques (optimisations) sur les termes de ce langage. Le travail sera effectué en OCaml.

On implémentera l'inlining, et peut-être le let-floating si on a le temps.

L'inlining consiste à remplacer un appel de fonction par son code.
Par exemple:

```
let f x = x + 4 in f 5
```

est remplacé par

```
5 + 4
```

Le let-floating déplace certaines définitions afin de rendre l'évaluation plus efficace. Par exemple:

```
let f x = let w = 4 in x + w
in f 5
```

est remplacé par

```
let w = 4 in
let f x = x + w in
f 5
```

Si le temps le permet, on abordera des optimisations avancées, en lien avec le flot de contrôle des programmes fonctionnels et basé sur la notion de "join point".

Ce projet a aussi un but pédagogique, pour expliquer comment on implémente la théorie de ces optimisations. On produira donc aussi une série de posts de blog qui expliquent le travail réalisé. On écrira pour une audience déjà familiarisée avec la compilation et la théorie des transformations, et qui souhaite en comprendre l'implémentation.

Testabilité

On portera une attention particulière aux tests pour vérifier le bon fonctionnement du projet. On écrira des tests unitaires pour confirmer le comportement du type-checker et des optimisations.

Calendrier

Le calendrier prévisionnel est divisé en 3 phases:
- modélisation du langage source en OCaml (2 mois)
- implémentation d'un vérificateur de type (1 mois)
- implémentation de transformations de programmes (inlining, let-floating) (2 mois)

On réserve 2 périodes de 15 jours pour la rédaction des rapports, ce qui donne en tout 6 mois de travail.

Références

- Compilation by transformation for non-strict functional languages; Santos
- Let-floating: Moving Bindings to Give Faster Programs; Peyton Jones, Partain et Santos
- A Transformation-Based Optimiser for Haskell, Peyton Jones et Santos
- Secrets of the Glasgow Haskell Compiler inliner; Peyton Jones et Marlow
- Compiling without continuations; Maurer, Downen, Ariola et Peyton Jones
- Compiling with continuations, or without? whatever; Cong, Osvald, Essertel et Rompf