Guide de l'utilisateur

Martin Bodin Jérémie Dumas

Table des matières

1 Présentation d'OcamlBike

1.1 Le langage Caml− reconnu

On retrouve les briques de base du langage Caml avec comme demandé :
- Des fonctions
- Des entiers, des booléens, unit
- Des références
- Des listes
- Du polymorphisme

Et voici grâcieusement repompée du deuxième pdf la liste des mots-clefs disponibles :

```
begin end + - * / ~- mod max_int min_int ( ) , ; ;;
true false && & || or <> = < > <= >= == != ! := [ ] ::
let rec and in fun function match with when as -> |
for while to downto do done if then else
```

Au niveau des fonctions prédéfinies, on a le droit aux fonctions suivantes : `print_int`, `print_bool`, `print_newline`, `read_int`. Attention à `print_bool` qui n'est pas une fonction OCaml de base.

1.2 Le fichier C produit

OCamlbike compile du code Caml− vers des insructions pour une machine SECD. Le fichier C produit est alors composé de deux parties :
- Une partie fixe, celle qui permet au code C d'interpréter les instructions de la machine. On pourra retrouver cette partie dans les fichiers C du dossier `src/machine/`.
- Une partie dynamique, contenant les instructions pour la machine. Cette partie change selon le programme Caml− à compiler.

2 Utilisation du compilateur

2.1 Options générales

`-i, -input <file>`

Permet de préciser un fichier d'entrée. Par défaut lit l'entrée standard. Le comportement du parseur est légèrement différent selon que l'on soit sur l'entrée standard ou non (il faut terminer ses phrases par des `;;`).

De plus, si l'option `-m` n'est pas spécifiée et que l'on est sur l'entrée standard, les informations sont traitées au fur et à mesure (alors qu'il faut envoyer un signal EOF avant que tout soit compilé si l'on a précisé `-m`).

Enfin, on peut préciser un fichier d'entrée sans utiliser l'option `-i` ou `--input` si le nom que l'on entre est le dernier argument passé au programme. Exemple :

```
./ocamlbike -m -O all toto.ml
```

`-o, -output <file>`

Permet de préciser un ficher de sortie. Par défaut réglé sur la sortie standard, sauf si l'option `-m` est spécifiée. Dans ce cas, le fichier de sortie est nommé `toto.c` par défaut.

`-h, -help`

Affiche un cours rappel des différentes options disponibles.

`-lg, -logo`

Option inutile d'un point de vue théorique. Affiche le logo ASCII d'OCamlBike sur la sortie en plus des informations habituelles.

2.2 Options pour l'analyse statique

`-l, -lexemes`

Affiche le flot de lexèmes lu sur l'entrée.

`-a, -alpha`

Affiche l'AST après alpha-conversion. Vérifie que le terme est clos, et que les patterns sont bien formés (variables liées plusieurs fois, disjonction de patterns ...).

Par défaut OCamlBike se content d'afficher l'AST tel qu'il est parsé par le programme.

`-t, -typed`

Affiche l'AST après typage (inférence, unification, prise en compte des `ref`, etc.), avec les types associés à chaque nœud.

`-la, -lambda`

Affiche l'AST typé après traduction vers le λ-calcul (élimination des pattern-matching, des for, while, etc.)

`-p, -pseudo`

Affiche le pseudo-code généré à partir du λ-terme. C'est ce pseudo-code qui sera traduit directement en un code C.

`-s, -opt_struct`

Affiche la structure d'optimisation. Cette structure est une abstraction qui représente le pseudo-code donné à la machine SECD. Se référer à la documentation sur les optimisations pour plus de détails.

`-m, -machine`

Génère le fichier C contenant :

- Le template déjà existant pour la machine SECD
- La liste du code encapsulé dans une fonction

On peut ensuite compiler le fichier C produit avec GCC, ou en utilisant le Makefile :

```
./ocamlbike -m -O all -i titi.ml -o titi.c && make titi.bin
```

2.3 Options pour machine SECD

`-r, -rectypes`

Autorise la compilation avec des termes dont le type est récursif. Attention les fonctions d'optimisation peuvent se mettre à boucler dans ce cas. Désactive par défaut l'affichage des types sur les nœuds de l'AST.

`-f, -force`

Force l'affichage des types dans le cas où l'on demande d'afficher l'AST typé ou le λ-terme produit. Cette option n'est pas nécessaire sauf si l'option `-r` a est précisée.

`-ov, -overload`

Autorise la compilation des opérateurs surchargés. Sans cette option le programme refuse de compiler lorsqu'on lui demande de comparer deux objets qui ne sont pas d'un type de base connu.

`-O, -optimise <option>`

Permet d'appliquer au pseudo-code généré une ou plusieurs optimisations. Le champ `<option>` peut prendre une des valeurs suivantes :

`none` N'ajoute aucune optimisation (pas très utile hein ?).

`inline` Le programme effectuera des β-réduction là où cela lui semble opportun.

`reduce` Le programme va effectuer des précalculs arithmétiques, ainsi que des simplifications de blocs `if` triviaux.

`tailrec` Le programme va détecter les appels terminaux et les remplacer par un instruction approprié. Comme cela ne fait pas grandir la pile, cela permet de faire boucler indéfiniment des programmes si l'on utilise en plus l'option `-rc`.

`all` Le programme effectuera les trois optimisations décrites ci-dessus (option conseillée).

Si l'on précise plusieurs options de cette manière, les effets sont cumulatifs.

Par exemple, les deux commandes suivantes sont équivalentes :

```
./ocamlbike -m -O all toto.ml
./ocamlbike -m -O inline -O reduce -O tailrec toto.ml
```

`-rc, -refcount`

Active le ramasse-miette à base de compteur de références au niveau de la machine SECD. Ralenti un peu le programme mais fait gagner beaucoup en espace. Problème : les structures cycliques ne seront toujours pas libérées (fonctions récursives, objets récursifs, etc.).

`-mc, -mcount`

Compte le nombre de malloc-free effectués au cours de l'exécution. Affiche le total obtenu quand le programme quitte.

`-g, -debug`

Mode de débugage pour la machine SECD. Effectue des tests de vérifications avant chaque opération. Ralenti le programme final.

`-v, -verbose`

Affiche l'état de la machine à chaque pas de l'exécution. On peut visualiser jusqu'à 10 éléments au sommet de la pile et de l'environnement.

3 Les fichiers d'exemples

3.1 Exemples de base

1plus1.ml Le programme le plus basique qui soit : il calcule $1 + 1$ et l'affiche (sans retour à la ligne d'aucune sorte).

compile.ml Pour illustrer le fait que l'on peut déclarer des fonctions à la volée.

for.ml Pour illustrer que la boucle For marche. Ce fichier affiche les 10 premiers entiers.

fun.ml Juste pour illustrer le fait que l'on peut faire du λ-calcul sans problème, et donc passer des fonctions en argument.

if.ml Juste pour montrer que le programme accepte les blocs If.

input.ml Fichier utilisé initialement pour tester le parseur. Il est issu du `list.ml` des sources d'OCaml. Mais comme il compile actuellement, on l'a gardé. Il contient beaucoup de petits exemples simples.

letrecand.ml Sert à illustrer les fonctions récursives croisées. Le fichier définit des fonctions qui disent si un entier est pair ou impair par récurrence.

match.ml Pour illustrer le fait qu'OCamlBike gère les `match`, même complexes (avec des `|` à l'intérieur des clauses, etc.).

palindromes.ml Un fichier totalement inutile mais tellement poétique. Mais il est bien compilé par OCaml comme OCamlBike.

sum.ml Il permet de montrer qu'OCamlBike gère bien les fonctions récursives et les listes.

3.2 Exemples de boucles

delta.ml La plus jolie de toutes les boucles. Ce programme ne compilera qu'avec l'option `-r`. C'est le $\delta\delta$ du λ-calcul.

lists.ml C'est un programme qui va (entres autres) tenter de calculer la longueur d'une liste infinie (créée récursivement).

loop.ml Une boucle sympathique elle aussi, qui consiste à définir une fonction par sa valeur. Évidement, ça boucle. Mais si on active `-O tailrec -rc`, le programme ne fait pas de stack overflow et boucle indéfiniment.

loop_ref.ml Ou comment simuler un `let \ldots rec` avec des références.

loop_while.ml La boucle classique de l'impératif.

3.3 Exemples intéractifs

fact.ml L'incontournable factorielle ne saurait être oubliée par OCamlBike !
Ce fichier est l'implémentation récursive simple de cette fonction.
Il commence par calculer et afficher 1! et 10!.
Puis, tant que l'utilisateur entre un nombre i différent de 0, il affiche i!.

ackermann.ml Encore un classique.
Le programme prend en entrée un nombre. Si ce nombre est non nul, il demande alors deux nombres n et m et affiche $A(n, m)$. Puis il recommence. Si ce nombre est nul, il s'arête.

3.4 Exemples avancés

assoc.ml Définit une fonction `assoc`, qui associe à un objet sa valeur associée dans une liste de couples.

church.ml Pour montrer que l'on peut faire du λ-calcul pur, voici une simulation des entiers de CHURCH en OCaml !

continuations.ml Quelques exemples de fonctions programmées par continuations. Attention aux `print_bool`

fact_ref.ml Pour montrer qu'OCamlBike gère l'impératif, voici une implémentation de la factorielle dans un code qui se veut proche de ce qu'on ferait en C.

parties.ml Il calcul quel est l'ensemble des parties de l'ensemble des parties de ... de l'ensemble vide. En sortie, il se contente d'afficher la taille de la liste optenue.

Ce code est un bon exemple pour tester les limites d'OCamlBike. Avec les options `-O all -rc`, il met vraiment *très* longtemps avant d'afficher "Memory full".

reclists.ml Pour montrer que les listes récursives infinies sont gérées par OCamlBike.

rectypes.ml Il construit des fonctions dont le type est récursif.

tri_fusion.ml Un autre bon test d'OCamlBike (par exemple pour comparer un programme généré optimisé ou non optimisé). Il va trier et afficher une liste de taille 10, initialement mélangée.