Compilation
Troisième partie, cahier des charges

À rendre le 8 janvier 2002 avant 20h

1 Compilation de votre projet

Votre projet doit se présenter sous forme d'une archive `tar` compressée (option "`z`" de `tar`), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms*. Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `camlette`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre rapport devra être rendu au moment de la soutenance. Outre ce qui est spécifique à cette troisième partie, votre rapport devra contenir, de manière synthétique (une page), un aperçu global des fonctionnalités de votre compilateur et des difficultés rencontrées.

2 Motivation

Le compilateur Camlette va générer du code `C`. Nous avons choisi ce langage pour trois raisons :

- La portabilité : Le code généré par Camlette sera exécutable sur toutes les machines (s'il existe un compilateur `C` pour cette machine).
- La représentation des données : Le langage `C` permet de représenter assez facilement des structures de données complexes. Et son système de type très primitif nous laisse une grande liberté dans l'écriture du code.
- L'utilisation d'un ramasse-miette : Les langages fonctionnels allouent de nombreuses structures de données (notamment pour les fermetures) ; afin d'éviter les fuites mémoire, nous utiliserons une bibliothèque `C` préexistante de gestion de la mémoire.

3 Représentation des données

Le langage Camlette permet au programmeur de ne pas se soucier des problèmes d'allocation et de libération de la mémoire. Cette tâche est laissée au compilateur. Pour cela, nous allons utiliser un ramasse-miette (GC, pour *Garbage Collector*).

On dit qu'un bloc mémoire n'est plus accessible lorsqu'il n'y a plus aucun pointeur vers ce bloc. Un GC est une bibliothèque qui se charge automatiquement de désallouer les zones de mémoire qui ne sont plus accessibles (vous n'aurez pas à utiliser la fonction `free` du langage `C`). Remarque : On dit qu'un bloc est alloué *dans le tas* lorsqu'il est alloué par le GC.

Nous avons choisi le GC de Boehm-Demers-Weiser :

```
http://www.hpl.hp.com/personal/Hans_Boehm/gc/
http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcinterface.html
```

Son utilisation est très simple. Vous n'aurez besoin d'utiliser que deux fonctions de cette bibliothèque (à la place de la fonction `malloc`) :

`void * GC_malloc(size_t nbytes)`
: alloue un bloc de taille `nbytes` ;

`void * GC_malloc_atomic(size_t nbytes)`
: version optimisée si le bloc alloué ne contient pas de pointeur.

Pour pouvoir les utiliser, il vous suffit d'inclure un "`#include <gc.h>`" dans vos programmes et de faire l'édition de liens avec l'option `-lgc`.

Les programmes générés par Camlette devront manipuler des données d'un seul type, `obj_t`, défini comme :

```
typedef long obj_t
```

Ce type permet de représenter les entiers et les pointeurs ; toutes les valeurs dont la représentation demande un espace mémoire supérieur à celui d'un `long` devront donc être allouées dans le tas (se sera donc un pointeur sur un bloc contenant la valeur). Nous donnons l'exemple des flottants.

3.1 Représentation des flottants

Comme nous l'avons fait remarquer ci-dessus, les flottants devront être alloués dans le tas. Un flottant sera donc un pointeur sur un `double`. La fonction suivante permet d'allouer un flottant :

```
obj_t alloc_double(double f) {
  double *res = (double *)GC_malloc_atomic(sizeof(double));
  *res = f;
  return (obj_t)res;
}
```

Vous utiliserez le même principe pour représenter les chaînes de caractères.

3.2 Représentation des fonctions

La particularité des langages fonctionnels est d'accepter les fonctions comme des valeurs de première classe. Ce n'est pas le cas dans le langage `C`. La plus grande difficulté de la génération du code sera la compilation des fonctions. Exemple :

```
let f x =
  let g y = x + y in g
```

Le résultat de l'expression `f 4` est la fonction `g` dans laquelle `x` vaut `4`. Comme le code est immuable, comment faire pour remplacer `x` par `4` dans `g` ? Pour cela, nous avons besoin d'une nouvelle notion : les *fermetures*. Une fermeture est un couple composé d'un pointeur de code (une fonction `C`) et d'un *environnement* ; un environnement est une liste ou un tableau (alloué dans le tas avec `GC_malloc`).

La représentation d'une valeur fonctionnelle de Camlette (cf : section 5) sera une fermeture. La fonction `C` de la fermeture prendra deux arguments (l'environnement de la fermeture et l'argument de la fonction Camlette). L'environnement contiendra les valeurs des variables libres de la fonction Camlette.

Pour représenter les fermetures, vous pourrez utiliser un pointeur vers une structure à deux champs (un pointeur de code, un environnement). La compilation d'une définition de fonction se fait donc en deux étapes :

1. La génération d'une fonction C (qui prend un argument supplémentaire, l'environnement).
2. L'allocation et l'initialisation d'une fermeture (ce qui implique également l'allocation et l'initialisation d'un environnement).

Remarque historique. Première apparition de la notion de fermeture dans la littérature : P.J. Landin, "The mechanical evaluation of expressions", The Computer Journal, 1964.

4 Compilation

Il y a plusieurs manières de compiler le langage Camlette. Nous vous proposons ici un schéma de compilation simple. Rien ne vous empêche d'en suivre un autre, du moment qu'il est correct.

Globalement, le schéma de compilation d'une expression est très simple :

1. Pour chaque sous-expression du programme source on crée une variable locale, qui reçoit le résultat de la sous-expression.
2. On fait le calcul correspondant à l'expression.
3. On rend le résultat.

Dans la suite, on note $[\![e]\!]$ la compilation de l'expression e.

4.1 Compilation des fermetures

Nous commençons par définir le type des fermetures, en pseudo C.

```
typedef ???? env_t;

typedef struct {
  obj_t (*code)();
  env_t env;
} closure;

typedef closure* closure_t;
```

Une fermeture est donc un pointeur sur une structure qui contient deux champs. Le champ `code` est un pointeur sur une fonction C (un pointeur de code). Le champ `env` contient l'environnement de la fermeture (nous vous laissons le choix du type `env_t` des environnements). On a alors

$[\![\texttt{fun } x \to e]\!] =$

```
    obj_t f(env_t env, obj_t x){
      return (* le resultat de l'évaluation de e *);
    }

    obj_t fclos;
    env_t fenv;
    {allocation et initialisation de fenv};
    {allocation de fclos};
    ((closure_t)fclos)->env = fenv;
    ((closure_t)fclos)->code = f;
```

Remarque : La compilation des fonctions récursives n'est pas plus compliquée. Il suffit que l'environnement de la fermeture contienne la fermeture elle-même.

4.2 Compilation des applications

$[\![e_1\ e_2]\!] =$

```
{ obj_t v1, v2;
  /* v1 recoit le résultat de l’évaluation de e1 */;
  /* v2 recoit le résultat de l’évaluation de e2 */;
  return (*(((closure_t)v1)->code))( ((closure_t)v1)->env, v2); }
```

Remarque : Le typage de l'expression $e_1\ e_2$, en Camlette, nous assure que la compilation en `C` de e_1 s'évaluera vers une fermeture. On peut donc faire la conversion `obj_t` vers `closure_t` sans crainte.

4.3 Les fonctions externes

Le langage Camlette nous permet de définir des fonctions externes (écrites en `C`) et de définir des types abstraits (externes). Nous imposons des restrictions particulières :

1. Si `t` est un type abstrait, on a `sizeof(t)` = `sizeof(obj_t)`.
2. Les fonctions externes ne doivent prendre en arguments que des variables de type `obj_t`.
3. Le type du résultat d'une fonction externe doit être `obj_t`.

Attention, on ne peut pas faire d'application partielle de fonction en `C`. Votre compilateur devra donc construire les fermetures correspondant aux fonctions externes.

Un ensemble de fonctions externes, qui seront notamment utilisées pour tester votre compilateur, sont fournies dans `~filliatr/lib` sous la forme des trois fichiers suivants :

prims.mlt : Les déclarations Camlette (à copier par exemple au début des vos fichiers de tests).

prims.h : Les déclarations `C` des fonctions correspondantes. Le code `C` produit par votre compilateur devra inclure ce fichier avec `#include <prims.h>`.

prims.c : Le code des fonctions `C`. Vous devrez le lier avec le code produit par votre compilateur.

5 Sémantique de Camlette

Les environnements d'évaluation et les valeurs sont donnés par la grammaire suivante :

$$\begin{array}{rcl} v & ::= & c \mid [\lambda x.e;\Gamma] \\ \Gamma & ::= & \epsilon \mid \Gamma,(x,v) \end{array}$$

où v est une valeur, c une constante de base et x une variable. La notation $[\lambda x.e;\Gamma]$ désigne la fermeture dont le code est $\lambda x.e$ et l'environnement Γ. Le jugement $\Gamma \vdash e \longmapsto v$ signifie que dans l'environnement Γ l'expression e s'évalue en v. Les règles sont les suivantes :

$$\frac{}{\Gamma \vdash c \longmapsto c} \qquad \frac{}{\Gamma \vdash x \longmapsto \Gamma(x)} \qquad \frac{}{\Gamma \vdash \texttt{fun}\ (x:\tau) \to e \longmapsto [\lambda x.e;\Gamma]}$$

$$\frac{\Gamma \vdash e_1 \longmapsto v_1 \quad \Gamma,(x,v_1) \vdash e_2 \longmapsto v}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longmapsto v} \qquad \frac{\text{soit } v_f = [\lambda x.e;\Gamma,(f,v_f)] \quad \Gamma,(f,v_f) \vdash e_2 \longmapsto v}{\Gamma \vdash \texttt{let rec}\ f = \texttt{fun}\ (x:\tau) \to e\ \texttt{in}\ e_2 \longmapsto v}$$

$$\frac{\Gamma \vdash e_1 \longmapsto v_1 \quad \Gamma \vdash e_2 \longmapsto v_2 \quad v = v_1\ op\ v_2}{\Gamma \vdash e_1\ op\ e_2 \longmapsto v}$$

$$\frac{\Gamma \vdash e_1 \longmapsto [\lambda x.e; \Gamma'] \quad \Gamma \vdash e_2 \longmapsto v_2 \quad \Gamma', (x, v_2) \vdash e \longmapsto v}{\Gamma \vdash (e_1\ e_2) \longmapsto v}$$

$$\frac{\Gamma \vdash e_1 \longmapsto \texttt{true} \quad \Gamma \vdash e_2 \longmapsto v}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longmapsto v} \qquad \frac{\Gamma \vdash e_1 \longmapsto \texttt{false} \quad \Gamma \vdash e_3 \longmapsto v}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longmapsto v}$$

Vous noterez la restriction faite par rapport aux parties 1 et 2 du projet : une fonction récursive est nécessairement de la forme `fun` $(x : \tau) \rightarrow e$. Cela vaut également pour les déclarations globales de fonctions récursives.

6 Invocation du compilateur

La commande `camlette` [*options*] *fichier* compile le fichier *fichier*. La valeur de retour de la commande est

- 0 si tout s'est bien passé ;
- 1 si le fichier donné en entrée est incorrect ;
- 2 si vous détectez une erreur dans votre compilateur (par exemple, vous arrivez sur un cas qui n'aurait pas dû être possible).

Les options possibles doivent comprendre au moins :

-help : affiche un message décrivant la façon d'invoquer le compilateur.

-about : affiche les noms des auteurs, et éventuellement le numéro de version du compilateur.

-no-parse : s'arrête avant l'analyse syntaxique ; en d'autres termes, effectue uniquement l'analyse lexicale.

-no-sem : s'arrête avant l'analyse sémantique ; autrement dit, effectue uniquement l'analyse lexicale et l'analyse syntaxique.

-no-code : s'arrête avant la production de code ; autrement dit, effectue uniquement les analyses lexicale, syntaxique et sémantique.

-out : pour nommer le fichier de sortie (par défaut il devra s'appeler $a.c$).

Vous pouvez de façon optionnelle ajouter une option **-debug** qui fera afficher diverses informations de déverminage à votre compilateur.