Compilation - corrigé à moitié du TD8

Compilation de langages fonctionnels

Dans ce TD, on part d'un langage fonctionnel de haut niveau, et on étudie sa compilation en un langage de bas niveau (insultons le C) muni d'une bibliothèque d'aide à l'exécution *(run-time system)*.

Les différentes étapes de cette compilation sont les suivantes :

Inférence de type : on décore chaque nœud de l'AST du programme avec son type. Ce problème a été survolé dans le TD précédent, et nous n'y reviendrons pas.

Désucrage *(desugaring)* : il s'agit d'enlever le sucre syntaxique pour se ramener à un langage fonctionnel minimal (LFM).

Optimisations sur le LFM : on peut essayer d'évaluer à la compilation tout ce qui peut l'être, de remplacer certains appels de fonction par le corps de la fonction, et de partager certaines sous-expressions communes. Ce genre d'optimisation est aussi utilisé pour la compilation de langages séquentiels.

Génération de code : une fois que l'on sait ce que l'on veut générer c'est facile. Donc ce sera dur.

1 Vue d'en haut

Question 1-1 On ne peut répondre à cette question que vers le milieu du TD. Nous sommes arrivés à : un programme fonctionnel c'est

- un certain nombre de définitions de fonctions, toutes globales (pas de fonctions locales), parce que nous voulons définir, dans notre code-objet C, une fonction pour chaque fonction du LFM,
- et une expression à évaluer qui appelle ces fonctions.

Les expressions sont réduite à leur plus simple expression (ou bien c'est le contraire) : un identificateur, un lambda, ou une application.

En résumé un AST minimal est définie par le type caml suivant :

```
program = Defs of (fundef list) * expr ;;
fundef = FunDef of string * (string list) * expr ;;
expr = Id of string | Let of string * expr * expr | Appl of expr * expr;;
```

Là dedans il manque le *if-then-else*. Si on fait de l'évaluation paresseuse, une fonction prédéfinie ternaire à trois arguments fera l'affaire. Sinon, il faut ajouter le *if-then-else* à la grammaire des expressions.

Ce qui n'a pas fini de vous perturber, c'est que la définition des fonctions est décurryfiée, alors que leur application est curryfiée. La raison pour cela sera claire lorsqu'on verra le générateur de code.

Question 1-2 On est vite gêné par des questions de fonctions passées en argument et autres applications partielles.

Question 1-3 On verra au fur et à mesure.

Question 1-4 Tout ce qui est génération de code assembleur, avec les problèmes d'allocation de registres, etc.

2 Désucrage

Question 2-1 On remplace le *pattern-matching* par des *if-then-else* imbriqués, testant les étiquettes des nœuds de l'AST avant de parcourir leurs fils.

Pour savoir quoi tester, il faut avoir déja inféré les types.

Question 2-2 Cela s'appelle du λ-*lifting*. On remplace `svmult` par

```
let multscal_in_svmult scalaire x = scalaire * x;;
let svmult scalaire vecteur =
   in map (multscal_in_svmult scalaire) vecteur
;;
```

Mon exemple n'est pas très convaincant, car il y a une optimisation simple à faire ici (et qu'un vrai compilateur ferait) : *expanser* `multscan` dans `svmult`, c'est-à-dire le remplacer par sa définition. Mais si la fonction `multscan` était récursive, ce ne serait pas possible.

Question 2-3 ..., non.

3 Réduction de graphe

Suite aux étapes précédentes, notre programme est constitué d'un certain nombre de fonctions, et d'une expression à réduire. Les opérateurs arithmétiques seront considérés comme des fonctions à deux arguments curryfiées dans un premier temps.

L'étape de calcul dans un programme fonctionnel est la β-réduction, qui consiste à remplacer $f(E)$ par $B_{\{x:=E\}}$, si la définition de f est de la forme $f(x) = B$. Le nœud de l'AST $f(E)$ est appelé le *redex* pour cette β-réduction.

Le cœur de l'exécution d'un programme fonctionnel est donc un moteur à trois temps :

- sélectionner un redex $f(E)$ dans l'AST,
- construire et instancier $B_{\{x:=E\}}$,
- remplacer la racine du redex par l'expression instanciée.

Ceci n'est en fait pas du calcul mais uniquement de la réécriture. Le calcul proprement dit est fait lorsqu'on β-réduit un opérateur arithmétique (ou autre fonction de bibliothèque), ce qui n'est possible que si ses arguments ont été réduits dans le type qu'il attend.

C'est ici qu'on commence à avoir l'intuition de ce que sera notre code objet :

- les expressions seront compilées en du code qui construit un arbre en C avec des pointeurs etc,
- une fonction à trois arguments sera compilée en une fonctions C à trois arguments (des arbres), et qui construira l'arbre de l'expression de la fonction, en utilisant pour ce faire les arbres passés en arguments à la place de ses paramètres formels,
- le `Let` sera construit en utilisant un environnement local en C,
- l'application de fonction sera laissée à la charge du programme qui fera les réductions.

Et une réduction ce sera : trouver le redex à réduire, attraper ses arguments et les passer à la fonction, qui renvoie un arbre tout comme il faut, qu'on met à la place du redex.

Question 3-1 Je me demande ce que cette question foutait là.

On considère à présent le code suivant :

```
let twice f x = f (f x) ;;
let square n = n * n ;;
twice square 3 ;;
```

Question 3-2 Je ne pousserai pas la piraterie jusqu'à recopier tous les dessins du Grune et al. Les remarques à faire sont :

- en général on a le choix pour le prochain redex à réduire,
- la réduction des redex des opérateurs est contrainte par les types de leurs arguments, contrairement à un redex de fonction utilisateur,
- on sait qu'on peut réduire un redex ayant une fonction f d'arité n si on a un peigne gauche d'`Appl` avec f tout en bas,
- en compilant les fonctions, on construira des DAG et pas des arbres, pour ne pas ensuite dupliquer du calcul,
- après une β-réduction, l'ancien redex est bon pour le ramasse-miette.

5 Génération de code

On va produire du C qui correspond exactement à l'AST précédent.
Voici les déclarations :

```
typedef enum {FUNC, NUM, NIL, CONS, APPL} node_type;

typedef struct node *Pnode;

typedef Pnode (*unary)(Pnode *arg);

struct function_descriptor {
  int arity;
  const char *name;
  unary code;
};

struct node {
  node_type tag;
  union {
    struct function_descriptor func;
    int num;
    struct {
      Pnode hd;
      Pnode tl;
    } cons;
    struct {
      Pnode fun;
      Pnode arg;
    } appl;
  } nd;
};

/* Constructor functions */

extern Pnode Func(int arity, const char *name, unary code);

extern Pnode Num(int num);

extern Pnode Nil(void);

extern Pnode Cons(Pnode hd, Pnode tl);

extern Pnode Appl(Pnode fun, Pnode arg);
```

Voici un constructeur (celui des nœuds `Appl`)

```
Pnode Appl(const Pnode fun, Pnode arg)
{
  Pnode node = (Pnode) heap_alloc(sizeof(*node));
  node->tag = APPL;
  node->nd.appl.fun = fun;
  node->nd.appl.arg = arg;
  return node;
}
```

Voici le code de `twice` (enfin, une partie) :

```
// La fonction facile à faire
Pnode twice (Pnode f, Pnode x) {
  return(Appl(f, Appl(f,x)));
}

// La fonction avec le bon type
Pnode _twice(Pnode *arg) {
  return twice(arg[0], arg[1]);
}
// on peut très bien inliner twice dans _twice


// La feuille correspondant à twice
struct node __twice = {FUNC, {2, "twice", _twice}};
```

Et voici un moteur de réduction simplifié :

```
#include "node.h"
#include "eval.h"
#define STACK_DEPTH 10000

static Pnode arg[STACK_DEPTH];

static int top = STACK_DEPTH; /* grows down */

Pnode eval(Pnode root)
{
  Pnode node = root;
  int frame, arity; frame = top;
  for (;;)
    {
      switch (node->tag)
{
case APPL: /* unwind */
  arg[--top] = node->nd.appl.arg; /* stack argument */
  node = node->nd.appl.fun; /* application node */
  break;
case FUNC:
  arity = node->nd.func.arity;
  if (frame-top < arity) { /* curried function */
    top = frame;
    return root;
  }
  node = node->nd.func.code(&arg[top]); /* reduce */
  top += arity; /* unstack arguments */
  *root = *node; /* update root pointer */
  break;
default:
  return node;
}
    }
}
```

6 Paresse optimisée

Tout est tiré du Grune et al.