Projet 1
ENS Lyon — L3IF
Génération de code

Sébastien Briais

8 janvier 2008

Dans cette partie, vous allez devoir implémenter le générateur de code pour l'émulateur DLX `risc-emu`.

À rendre pour le 18 janvier 2008 à minuit au plus tard.

1 Consignes générales

Votre compilateur devra pouvoir être invoqué à l'aide de la commande `dreic` dans un terminal. Il prendra en argument un paramètre indiquant la phase du compilateur testée suivi du nom de fichier `drei` à "compiler". Un exemple typique d'utilisation est :

```
./dreic -generator Factorial.drei
```

Les noms des phases sont indiqués ci-après :

Nom de la phase	Nom de l'option
Analyse lexicale	`-scanner`
Analyse syntaxique	`-parser`
Construction de l'arbre	`-printer`
Analyse des types	`-analyser`
Génération de code	`-generator`

En outre, pour le rendu de votre projet,

- Vos sources devront être obligatoirement accompagnées d'un fichier `Makefile` permettant de construire votre projet. *Optionnellement*, vous pouvez aussi joindre un fichier `README` pour y reporter vos remarques éventuelles sur le fonctionnement ou la compilation de votre programme. Avec un peu de chance, il sera lu :p
- Vous devrez rendre vos sources sous forme d'une archive "tar gzippée". Le contenu de votre archive devra se décompresser dans un répertoire portant le nom de votre groupe (i.e. `B1`, `R3`, ...)
- L'archive devra m'être envoyée par mail. Le titre de votre mail sera de la forme "[PROJET] Etape 4"

2 À propos de l'émulateur DLX

Le jeu d'instruction du processeur est décrit dans le document situé à l'URL suivante : `http://tinyurl.com/2rskh7`.

La version caml de l'émulateur RISC est située aux endroits suivants :

- `/home/sbriais/bin/risc-emuf` (version bytecode caml)
- `/home/sbriais/bin/risc-emuf.opt` (version bytecode caml)

La version Java de l'émulateur RISC est située aux endroits suivants :

- `/home/sbriais/bin/risc-emu` (version en ligne de commande)
- `/home/sbriais/bin/risc-gui` (version avec GUI)

Les sources de la version caml sont disponibles à l'URL suivante : `http://tinyurl.com/3azhlx`.

Comme expliqué en cours, la version caml est plus souple au niveau de la gestion des labels mais est moins conviviale que la version graphique en Java. Il sera donc plus facile pour vous de générer du code pour la version caml mais pour déboguer votre générateur de code, vous préférerez sans doute utiliser la version Java. La version caml permet de prémâcher le code DLX fourni en entrée pour donner du code DLX accepté par la version Java (flag `-dump`).

Par exemple, si le fichier `source.asm` contient le code DLX suivant (pas accepté par la version Java de l'émulateur) :

```
classA:
  DATA A_f
  DATA A_g
A_f:
  RET R31
A_g:
  RET R31
```

Alors `risc-emuf.opt -dump source.asm` vous donnera :

```
DW 8
DW 12
RET 31
RET 31
```

qui est accepté par la version Java.

Enfin, comme mentionné en cours, faites attention : la signification des flags n'est pas forcément la même entre la version caml et la version Java.

3 Description de l'étape

3.1 Organisation des registres en pile

Pour une allocation basique des registres, on les organise en pile. Il est pratique d'implémenter les fonctions suivantes :

```
(** alloue un nouveau registre et le place au sommet de la pile *)
val freshReg : unit -> register

(** libere le registre en sommet de pile *)
```

```
val dropReg : unit -> unit

(** renvoie le registre en haut de la pile *)
val topReg : unit -> register

(** renvoie le deuxieme registre a partir du haut de la pile *)
val sndReg : unit -> register
```

où le type register peut simplement êtré défini par :

```
type register = int
```

3.2 Gestion des blocs d'activation

Pour pouvoir accéder aux variables locales et aux paramètres des fonctions, votre générateur de code va tenir à jour en permanence la taille du bloc d'activation courant. Ainsi, vous pourrez vous passer de la manipulation explicite d'un registre `FP` (frame pointer).

Je vous suggère les fonctions suivantes pour vous faciliter la vie :

```
(** alloue n octets dans le bloc courant *)
val incFrameSize : int -> unit

(** libere n octets dans le bloc courant *)
val decFrameSize : int -> unit

(** renvoie la taille du bloc courant *)
val getFrameSize : unit -> int
```

L'idée est qu'à chaque fois que vous générez une instruction qui modifie le pointeur de pile (à savoir le registre `R30`), vous devez appeler la fonction correspondante mettant à jour la taille du bloc d'activation courant.

3.3 Extension des symboles

Vous allez étendre le type des symboles pour contenir des informations nécessaires à la génération de code :

- Un symbole de variable contiendra l'offset par rapport à FP pour accéder à sa valeur.
- Un symbole de champ contiendra l'offset par rapport au début de l'adresse d'un objet pour accéder à sa valeur (n'oubliez pas de compter un champ supplémentaire pour stocker l'adresse de la VMT)
- Un symbole de méthode contiendra l'offset de la méthode par rapport au début de la table de méthode virtuelle.
- Un symbole de classe contiendra un label qui indiquera l'endroit où la table de méthode virtuelle (VMT) se trouve en mémoire.

Concrètement, cela revient à modifier les types des symboles comme ceci :

```
type var_symbol =
    { var_type : dtype;
      mutable var_offset : int option;
    }
```

```
and field_symbol =
    { field_type : dtype;
      mutable field_offset : int option;
    }
and method_symbol =
    { params_type : dtype list;
      return_type : dtype;
      (** offset dans la VMT *)
      mutable method_offset : int option;
      (** debut du code de la methode *)
      mutable method_label : label option;
    }
and class_symbol =
    { parents : string list;
      fields : field_scope;
      methods: method_scope;
      mutable class_label : label option;
    }
```

où le type label peut simplement êtré défini par :

```
type label = string
```

Vous aurez probablement besoin d'une fonction pour générer des nouveaux labels.

```
val new_label : unit -> label
```

3.4 Génération du code

Une bonne manière d'aborder cette étape est d'ajouter progressivement chaque aspect du langage `drei`. Par ordre de difficulté croissante, vous avez :

- Les entrées/sorties.
- Les expressions arithmétiques.
- Les classes (sans les méthodes) et les objets.
- Les blocs et les variables locales.
- Le `if` et le `while`.
- Les conditions.
- Les méthodes.

Concrètement, vous aurez trois fonctions mutuellement récursives pour générer le code des expressions, des conditions et des énoncés.

```
(** genere le code evaluant l'expression et
    place le resultat dans RSP *)
val generate_expression : expression -> unit

(** genere le code d'une condition. Saute au label si la condition
vaut le booleen passe en argument *)
val generate_condition : label -> bool -> expression -> unit

(** genere le code correspondant a l'enonce donne *)
val generate_statement : statement -> unit
```

Pour générer le code d'un membre d'une classe (méthode ou champ), vous aurez la fonction suivante :

```
val generate_member : member -> unit
```

Le code pour générer une définition de classe sera alors simplement une suite d'appels à la fonction generate_member.

```
val generate_class : class -> unit
```

Le code pour générer la VMT qui correspond à une classe sera implémenté par :

```
val generate_vmt : class -> unit
```

C'est cette fonction qui se chargera d'attribuer un label pour le symbole de classe, un label pour chacun des symboles de méthodes et qui calculera les offsets de chacun des symboles de champs.

Enfin, le code d'un programme sera généré par :

```
val generate_program : program -> unit
```

3.5 Initialisation de l'émulateur

Pour rappel, le code que vous allez générer aura la structure suivante :

```
start:
  BEQ R0 init // saute au code d'initialisation
vmts:
  ... // les VMTs des classes
methods:
  ... // le code des methodes
main:
  ... // le code de l'expression principale
  RET R0 // quitte l'emulateur
init:
  SYSCALL R30 0 13                  // initialise le pointeur de pile
  ORIU R1 R0 ((init >> 16) & 0xffff)
  LSHI R1 R1 16
  ORIU R1 R1 (init & 0xffff)        // le tas commence en init
  SUB R2 R30 R1                     // taille memoire sans le code
  DIVIU R2 R2 (3*4)                 // coupe en trois morceaux
  LSHI R2 R2 1                      // deux tiers pour le tas
  ORIU R3 R0 30                     // registre de pile = R30
  LSHI R3 R3 27
  OR R2 R2 R3
  SYSCALL R1 R2 11                  // initialise le GC
  BEQ R0 main                       // saute a l'expression principale
```

Attention, le code ci-dessus est accepté uniquement par la version caml de l'émulateur DLX.

4 Extensions

Je propose ci-dessous quelques extensions. J'apprécierais que les groupes de type B essaient d'en réaliser au moins une. Clairement, toutes ne font pas le même poids en terme de travail à fournir. Si vous avez d'autres extensions à proposer, n'hésitez pas à m'en parler.

Encore une fois, je conseille la lecture de l'excellent livre de Andrew Appel *Modern Compiler Implementation in ...* pour vous aider à réaliser ces extensions.

Les extensions qui me semblent intéressantes sont les suivantes :

- Écrire un vrai allocateur de registres.
- Écrire un garbage collector pour ne plus devoir se reposer sur celui fourni par l'émulateur DLX.
- Optimiser les appels de méthodes quand on peut connaitre statiquement l'instance de la méthode qui doit être appelée.
- Ajout de la coercition et du typage dynamique.
  Certains langages objet tels que Java disposent d'un mécanisme permettant :
  - de tester dynamiquement de quelle classe un objet est une instance (isInstanceOf),
  - de coercer le type statique d'une classe vers quelque chose de plus précis, c'est-à-dire un sous type du type statique, et de générer une erreur dynamique si cette coercition s'avère finalement impossible (asInstanceOf).

  Ce mécanisme, très utilisé dans des langages avec des systèmes de types «pauvres» (comme Java 1.4), peut également être ajouté à `drei`.
  Il faudra ici étendre la syntaxe de `drei` pour autoriser ces constructions, étendre le système de type et étendre le générateur de code.
- Ajout des tableaux.
- Autoriser des champs mutables dans les objets.