Projet I

Alois BRUNEL et Vincent DELAITRE

19 janvier 2008

1 Notre compilateur

1.1 Options de la ligne de commande

Notre compilateur se lance avec la syntaxe suivante :
./dreic -phase fichier_source [arguments] > fichier_cible
Le paramètre -phase doit être l'un des suivants :

Phase	Description
-scanner	Teste la phase du scanner sur le fichier fichier_source
-parser	Teste la phase du parser sur le fichier fichier_source
-printer	Teste la phase du printer sur le fichier fichier_source
-analyser	Teste la phase du analyser sur le fichier fichier_source
-generate	Teste la phase du generator sur le fichier fichier_source

Les arguments possibles sont les suivants :

Argument	Description
-gc	Active notre Garbage Collector

Si on veut activer -gc, il faut quand même concaténer le code de gc.drei au tout début du fichier source ! !

1.2 Conception

Nous avons suivit de près le partitionnement du programme en différents fichiers tel que vous l'aviez proposé : un fichier pour chaque phase du compilateur. Nous avons de même suivit de près le type des fonctions proposées. Passons donc plutôt à ce que nous avons ajouté comme améliorations.

2 Extensions du language

2.1 Commentaire

Nous commencons très fort avec une amélioration digne d'un groupe bleu : les commentaires sur plusieurs lignes. Tout texte compris entre /* et */ compte comme un commentaire. Quitte à faire une amélioration toute pourrie, on ne gère pas les commentaires imbriqués.

2.2 Ajout d'opérateurs binaires

Pour pouvoir coder le garbage collector, nous avons d ajouter quelques opérateurs binaires :

Opérateur	Symbole
et	&
ou	|
ou exclusif	^
non	~
décalage à gauche	<<
décalage à droite (logique)	>>

Remarque : si l'opérande de droite est négatif, << se comporte comme >> et >> comme <<.

2.3 Portées des variables

Nous avons fait l'amélioration de la phase 3 (analyser) : on peut déclarer une variable alors qu'il en existe une ayant le même nom à condition que cette dernière ne soit pas dans le même bloc que la première (n'ait pas la même porté que la première). Techniquement, il suffisait de choisir comme contexte de variable une liste de liste des variables déclarées dans chaque bloc.

3 Améliorations

3.1 Entiers 32 bits

Nous avons remarqué qu'il était impossible de passer en valeur immédiate des entiers de plus de 16 bits (Pour les instructions xxxI). Cela posait quelques problèmes, et nous avons donc utiliser la même technique que celle utilisée dans le code de "init". C'est-à-dire une reconstruction de la valeur 32 bits dans un registre à partir des deux parties fortes et faibles. Nous n'effectuons cette transformation que si la valeur nécessite effectivement plus de 16 bits.

3.2 Expressions

Nous avons initialement implémenté une fonction pour générer des expressions qui marchait bien mais n'était pas efficace, nous avons donc décidé de lui faire une petite soeur. La petite soeur devait supporter deux points : être capable de simplifier des expressions comme $1 + 4 * x * 2 - 4$ en $8 * x - 3$ et éviter de charger plusieurs fois une même variable dans un registre. Pour cela nous avons défini un nouveau type pour les expressions dans expr_tree.ml. C'est une représentation des expressions sous forme d'arbre : les noeuds sont les opérateurs binaires et unaires, ainsi que les appels de champ ou de méthode et les new. Les feuilles sont les variables, les constantes et les fonctions d'entrée readInt et readChar. Cette représentation permet d'accéder facilement aux variables, qui, si elles sont stockées dans un registre, sont directement accessibles aux opérateurs :
LDW R2 R2 0 //la variable à l'adresse R2 est stockée dans R2
MUL R1 R1 R2 //R1¡-R1*R2

est remplacé par :
MUL R1 R1 R8 //on suppose que les variable est déjà en R8
Malheuresement, nous n'avons pas eu le temps d'implémenter la sauvegarde des variables dans les registres. Cependant, la simplification des expressions fonctionne ce qui permet parfois d'utiliser les opérations xxxI, ce que nous ne faisions pas avec l'ancienne version de generate_expression.

3.3 Tableaux

Les tableaux ont été implémentés. Ils peuvent être définis comme suit :

```
var tab : Array Int = [1;3;4];
var tab2 : Array Array Int = [tab;tab];
```

Ils sont représentés dans le tas par un bloc contenant la taille du tableau (en octets) suivie des éléments de ce tableau. On peut accéder à la i-ème case du tableau de la manière suivante :

```
set x = tab2#1#2; // x = 3
```

Attention, les indices commencent à 0. Si par malheur l'utilisateur dépasse le tableau, le programme s'arrête en affichant un 'a' (histoire de comprendre ce qu'il se passe). Nous n'avons malheureusement pas eu le temps de rendre les tableaux mutables, ce qui rend leur utilisation un peu limitée. Néanmoins, c'est assez facile, il aurait suffit de rajouter un opérateur ¡- dans la liste des tokens et dans l'AST. Puis d'effectuer un STW à l'endroit voulu. Le code est laissé en exercice à qui voudra.

3.4 Injection de code assembleur

Pour pouvoir utiliser des routines bas niveau (manipulation de la mémoire via des pointeurs), nous avons inséré une nouvelle directive d'injection de code assembleur dans le code drei. Si nous écrivons dans un bloc @ INS ... @, cela insère en dur, et sur place, le code assembleur écrit entre @ et @ (ce code ne peut être écrit que sur une ligne). Nous avons ainsi pu coder les fonctions de lecture et d'écriture en mémoire, et de ce fait coder le Garbage Collector directement en Drei.

3.5 Directive de détournement de typage

L'utilisation de la directive précédente pose problème, lorsque l'on veut par exemple faire qu'une fonction dont le code est entièrement injecté en assembleur retourne un entier. Comme nous ne pouvons utiliser l'instruction return, il nous est impossible de donner le type Int au retour de la fonction. Pour gérer au mieux ce problème, la directive "asm" remplace "def" pour signaler à l'analyseur que cette fonction peut retourner tous les types (l'utilisation de TBad le permet assez facilement). Voici un exemple de fonctions déclarées ainsi.

```
//Lecture d'un mot \'a l'adresse x
asm rdm_w (x : Int) {
  @LDW R29 R30 4@
  @LDW R1 R29 0@
```

```
}

// Shift droit : x << y
asm shiftr (x:Int, y:Int) {
  @LDW R29 R30 8@
  @LDW R28 R30 4@
  @ASH R1 R29 R28@
}
```

3.6 Le Garbage Collector

Nous avons tenté d'implémenter un garbage collector. Celui-ci fonctionne en Mark&Sweep, ce système étant assez facile d'utilisation, et la "pause GC" n'est pas trop longue. Nous avons d'abord codé en Drei grce aux améliorations précédentes un malloc, un free ainsi qu'un mergefreeblocks (qui fusionne les blocs libres adjacents). Les blocs sont simplement représentés par un espace de n+4 octets (n est la taille du bloc côté utilisateur), le premier octet servant à stocker la taille du bloc qui est un multiple de 4, et ses deux bits de poids faibles servant au coloriage du garbage collector et à renseigner sur le caractère utilisé ou non du bloc (cela est possible car la taille est multiple de 4 donc n'utilise pas ces 2 bits). Finalement, une fonction de coloriage a été utilisée. On effectue cela de manière brutale : pour toutes les adresses trouvées dans la pile qui dénotent un début de bloc alloué par le malloc, on colorie tous les fils, c'est-à-dire qu'on parcourt ce bloc à la recherche d'adresses de début de bloc (pointant dans le tas donc) et on les colorie récursivement.
Après avoir réfléchi, nous avons pensé à utiliser un arbre binaire de recherche pour stocker les adresses de bloc, ce qui aurait augmenté la vitesse du garbage collector. Néanmoins, nous n'avons pas eu le temps de tester cela, et la structure semblait assez lourde. Nous n'avons pas non plus beaucoup eu le temps de tester notre Garbage collector (le malloc marche très bien normalement, mais la recherche de blocs vivants et la libération de mémoire n'a pas été testée énormément), il ne faut donc pas hésiter à utilise le GC de l'émulateur en cas de problème.
Voici à titre d'exemple le code de la fonction Alloc s'occupant de l'allocation mémoire.

```
def Alloc (n : Int) : Int = {
  var t : Int = (n-(n%4))+4;
  var p : Int = this.search_freeblk(t);
  var prochain : Int = -1;
  var t1: Int = this.Size(p);
  var res : Int = p;

  /* Si le bloc est libre et de taille suffisante, on l'alloue */
  if ((p != -1) && (t1 > t))
  {
    /* On d\'ecoupe */
    do this.setSize(p, t);
    do this.setUsed(p);

    set prochain = this.blk_next(p);
    do this.setSize(prochain, t1 - t);
    do this.setFree(prochain);
```

```
      set res = res + 4;
    }
    else if ((p != -1))
    {
      do this.setUsed(p);
      do this.setSize(p,t);
      set res = res + 4;
    }

    return res
  };
```

Nous avons eu un problème de dernière minute et le GC ne marche plus dans la dernière version du compilateur, c'est pourquoi on a mis aussi une ancienne version où le GC fonctionne.

4 Conclusion

Nous avons tenté de coder un compilateur effectivement capable de compiler du drei, et nous espérons avoir au moins réussi un peu. Mais nous avons un peu délaissé le côté efficacité en termes de nombre d'opérations, par exemple dans le Garbage Collector, qui est du coup moins rapide que celui de l'émulateur. C'est ce que nous avons tenté de réduire avec l'amorce d'un allocateur de registre. C'est certainement le côté qui reste le plus à développer dans notre travail.