Compilation de machine abstraite vers UXN

Timothée MENEUX
Stagiaire
Ecole Normale Supérieure
Rennes, France
timothee.meneux@ens-rennes.fr

Alan SCHMITT
Encadrant
Epicure, IRISA-INRIA
Rennes, France
alan.schmitt@inria.fr

Martin ANDRIEUX
Co-encadrant
Epicure, IRISA-INRIA
Rennes, France
martin.andrieux@inria.fr

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.**

***Index Terms*—Machines Abstraites, Compilation, UXN**

I. Introduction

Contexte. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At. [1], [2]

Objet de recherche. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere.

Contributions. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos.

Travaux des pairs. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At.

II. Première machine abstraite

A. Syntaxe

On introduit notre machine abstraite cible, un évaluateur d'expressions arithmétiques simples.

$$\begin{aligned} expr ::= \ & Int(i) && i \in \mathbb{Z} \\ \mid \ & Add(expr, expr) \\ \mid \ & Sub(expr, expr) \end{aligned}$$

Figure 1: Syntaxe des expressions

$$\begin{aligned} cont ::= \ & Id \\ \mid \ & Ar_{Add}(expr, cont) \\ \mid \ & Fn_{Add}(expr, cont) \\ \mid \ & Ar_{Sub}(expr, cont) \\ \mid \ & Fn_{Sub}(expr, cont) \end{aligned}$$

Figure 2: Syntaxe des continuations

On notera $\mathcal{L}(expr)$ et $\mathcal{L}(cont)$ comme étant les langages reconnus par les grammaires $expr$ et $cont$.

B. Sémantique concrète

Nos expressions admettent une sémantique concrète, celle de l'arithmétique. On évalue ces termes avec la fonction $[\![\cdot]\!]_{sc} : \mathcal{L}(expr) \longrightarrow \mathbb{Z}$ définie comme suit :

$$\begin{aligned} [\![Int(i)]\!]_{sc} &= i \\ [\![Add(e_1, e_2)]\!]_{sc} &= [\![e_1]\!]_{sc} + [\![e_2]\!]_{sc} \\ [\![Sub(e_1, e_2)]\!]_{sc} &= [\![e_1]\!]_{sc} - [\![e_2]\!]_{sc} \end{aligned}$$

Figure 3: Sémantique concrète

C. Sémantique de la machine abstraite

Pour faire marcher notre machine abstraite, on la munie de trois états $eval$, $apply$ et res ainsi que d'un

système de transition $\longrightarrow^\sharp$ étant une sémantique à petits pas.

$$\begin{aligned} eval &::= \langle expr \mid cont \rangle_e \\ apply &::= \langle cont \mid expr \rangle_a \\ res &::= \langle i \rangle_r \qquad\qquad i \in \mathbb{Z} \end{aligned}$$

Figure 4: États de la machine abstraite

$$\begin{aligned} \langle Int(i) \mid \kappa \rangle_e &\longrightarrow^\sharp \langle \kappa \mid i \rangle_a \\ \langle Add(e_1, e_2) \mid \kappa \rangle_e &\longrightarrow^\sharp \langle e_1 \mid Ar_{Add}(e_2, \kappa) \rangle_e \\ \langle Sub(e_1, e_2) \mid \kappa \rangle_e &\longrightarrow^\sharp \langle e_1 \mid Ar_{Sub}(e_2, \kappa) \rangle_e \end{aligned}$$

Figure 5: Transitions pour les états *eval*

$$\begin{aligned} \langle Id \mid Int(i) \rangle_a &\longrightarrow^\sharp \langle i \rangle_r \\ \langle Ar_{Add}(e, \kappa) \mid Int(i) \rangle_a &\longrightarrow^\sharp \langle e \mid Fn_{Add}(Int(i), \kappa) \rangle_e \\ \langle Ar_{Sub}(e, \kappa) \mid Int(i) \rangle_a &\longrightarrow^\sharp \langle e \mid Fn_{Add}(Int(i), \kappa) \rangle_e \\ \langle Fn_{Add}(Int(i_1), \kappa) \mid Int(i_2) \rangle_a &\longrightarrow^\sharp \langle Int(i_1 + i_2) \mid \kappa \rangle_e \\ \langle Fn_{Sub}(Int(i_1), \kappa) \mid Int(i_2) \rangle_a &\longrightarrow^\sharp \langle Int(i_1 - i_2) \mid \kappa \rangle_e \end{aligned}$$

Figure 6: Transitions pour les états *apply*

En Fig. 6, les opérations "+" et "−" représentent le vrai calcul opéré par la machine. De plus, l'état res n'a pas de successeur puisqu'il symbolise l'état final de la machine.

On définit alors la clôture réflexive transitive de la relation de transition $\longrightarrow^\sharp$ par $\longrightarrow^{\sharp *}$. Pour deux états σ et σ', on a $\sigma \longrightarrow^{\sharp *} \sigma'$ si et seulement si il existe une famille $(\sigma_i)_{0 \leq i \leq n}$ telle que $\sigma_0 = \sigma$, $\sigma_n = \sigma'$ et pour tout $0 \leq i < n$, $\sigma_i \longrightarrow^\sharp \sigma_{i+1}$.

On pose donc la fonction d'évaluation associée à la machine abstraite $[\![\cdot]\!]^\sharp : \mathcal{L}(expr) \longrightarrow \mathbb{Z}$ comme étant le résultat de cette dernière sur l'entrée. Explicitement, on a $[\![e]\!]^\sharp = i$ si et seulement si $\langle e \mid Id \rangle_e \longrightarrow^{\sharp *} \langle i \rangle_r$.

III. Au niveau machine

A. Présentation rapide de UXN

En *UXN*, on manipule une machine à pile simple ayant un nombre restreint d'*opcodes* (32 et 3 modes possibles) qui permettent de manipuler des octets (8 bits) ou des *shorts* (2 octets, 16 bits). On a alors un espace mémoire adressable de 2^{16} octets (64ko) et deux piles de 256 octets, une de travail et une de retour.

Ainsi, en *UXN* on écrit un code assembleur en notation polonaise inversée.

```
1  |0100        //
2      #01      // 01
3      #02      // 01 02
4      ADD      // 03
5      #04      // 03 04
6      MUL      // 12
```

Exemple 1: calcul de $1 + 2$ en *UXN* (en commentaire : l'état de la pile de travail).

Remarquons que sur l'exemple précédent, on peut déplacer des lignes sans que cela n'affecte le calcul. Par exemple, on pourrait remonter la ligne 5 en ligne 2 et tout ce passerait pareil. On aurait :

$$\text{// 04 01 02} \xrightarrow[\text{ADD}]{} \text{// 04 03} \xrightarrow[\text{MUL}]{} \text{// 12}$$

B. Représentation machine des objets

Pour la représentation des objets, j'ai décidé de m'inspirer de la représentation à la *OCaml*. Anisi, un objet est représenté en mémoire comme un triplet $\langle tag \mid size \mid data \rangle$ où :

- tag : indique le numéro de constructeur utilisé pour créer l'objet, codé sur un octet,
- $size$: indique le nombre de *shorts* réservés après, codé sur un octet,
- $data$: est l'espace utilisable en mémoire, un tableau de $size$ shorts.

On remarque alors qu'un objet occupe $2 \times (size + 1)$ octets, soit $size + 1$ *shorts*. Ainsi, le champs *data* peut contenuir au plus 255 shorts (car $size \leq 255 = 2^8 - 1$).

Il est intentionnel que les champs tag et $size$ soient codés sur un octet chacun, ils sont contigus en mémoire et l'on ne travaillera qu'avec des *shorts*, ce qui est le cas de la concaténation de ces deux champs.

On appelle alors une case mémoire l'espace mémoire nécessaire pour stocker un *short* alligné.

C. Gestion mémoire

Afin de gérer la mémoire (de 64ko), on va équiper notre machine d'un équivalent de *malloc* et de *free*.

Le principe est simple. L'espace addressable est divisé en trois, une section *text*, une section *code* et une section *raw*. Les sections *text* et *code* contiennent le code assembleur chargé et les différentes variables d'environnement (qui ne peuvent pas avoir de valeur par défaut dans la section *text*).

Il nous reste alors toute la section *raw* qui peut être utilisée pour stocker dynamiquement des objets.

Lors d'un *malloc*, on va chercher la première case ayant un *tag* nul (*ie* le *tag* vaut 00) puis vérifier que les *size* cases suivantes ont aussi un *tag* nul. Si c'est le

cas on écrit, sinon on reprend la recherche à partir de là. S'il n'y a plus de place, on renverra un pointeur nul (*ie* le pointeur 0000).

Pour effectuer un *free*, on a juste à écrire 00 sur le premier octet de chaque case de notre objet ($size + 1$ cases), ainsi chaque case sera alors considérée comme une case libre (*tag* 00).

IV. Ce qu'il r_este à faire

A. Terminer malloc-free

Je dois rajouter le *free*, ça ne devrait pas être long car c'est une simple boucle.

Je dois modifier le *malloc* pour qu'il recherche vraiment de l'espace libre car pour le moment c'est une simple pile avec un *push*. Ça ne devrait pas être beaucoup plus long.

B. Gérer les input

Pour le moment, le calcul à effectuer est écrit dans le *main* du code. Ce serait bien de rendre le programme un minimum interactif, qu'il puisse prendre en *input* via le terminal une expression et l'évaluer.

C. AST et génération de code

Créer la syntaxe générale d'une machine abstraite et donc de l'AST qui lui correspond afin de pouvoir générer du code automatiquement, pour toute machine abstraite prise en entrée.

References

[1] M. Ager, "A Functional Correspondence Between Evaluators and Abstract Machines," 2003, doi: 10.1145/888251.888254.

[2] S. Diehl, P. Hartel, and P. Sestoft, "Abstract machines for programming language implementation," *Future Generation Computer Systems*, vol. 16, no. 7, pp. 739–751, 2000, doi: https://doi.org/10.1016/S0167-739X(99)00088-6.