COA

Pablo De Oliveira

Table des matières

Retards Attention : un train d'étudiants en retard à 5 minutes d'intervalle fait perdre 20 minutes au cours ! Une petite tolérance est certes normale, mais cela ne sera en aucun cas systématique (revenez à la pause).

1 Introduction

Pourquoi faire de la compilation dans un master de HPC ? Depuis quelques années, les architectures deviennent de plus en plus complexes (pipeline, machines vectorielle, multi-noeud, interfaces réseaux haute performance, etc).

Dans les machines actuelles, chaque instruction est codée comme un nombre binaire, qui a un sens pour la machine. Pour des raisons évidentes de compréhension, on préférera utiliser des *mnémoniques* du type `mov $r0, $r1` au lieu de 10001001 11 000 011. L'*assembleur* est le programme convertissant les mnémoniques en binaire. Le soucis est que chaque processeur a ses propriétés propre : il possède un coprocesseur à unité flottantes, ou un accélérateur vectoriel. On cherche alors à rendre le programme *portable* : ne pas le réécrire pour chaque machine. De plus, le langage assembleur est très *bas niveau* : on voudrait pourvoir exprimer des boucles, des conditions, etc, de telle manière à le rendre compréhensible par des humains.

Des *interpréteurs* sont des programmes dont le rôle est de décoder mot-à-mot le langage et le transcrire à la volée en instructions machines. Le soucis est que ces interpréteurs ne sont pas très optimisés : on n'obtient pas le programme le plus rapide pour une machine donnée.

Il existe également des *machine virtuelles*, qui interprètent un assembleur relativement simple sur une machine hôte, comme *Java* et *.net*. Cela permet des optimisations à la compilation et à la volée lors de l'exécution. Il reste cependant toujours une couche intermédiaire.

Enfin, les compilateurs prennent un langage et le traduisent vers un code assembleur directement compréhensible par la machine. La *sémantique* doit être cependant conservée : sur toute entrée du programme, les sorties doivent être identiques. Le compilateur est donc un outils du chercheur HPC permettant de l'aider à déployer son code.

Par contre, un compilateur est moins flexible et dynamique ; certaines optimisations ne pouvant pas être effectuées à la compilation (celle dépendant des valeurs à l'exécution).

1.1 Anatomie d'un compilateur

Le but de ce cours et de développer un compilateur du langage *Tiger* en se basant sur LLVM.

Il existe une multitude de langage sources et d'architecture cibles ; c'est pourquoi on a introduit l'idée de *Représentation intermédiaire* ; qui se divisent en *front-end* (langage vers IR) et *back-end* (IR vers cible). On écrit alors des optimisations qui fonctionneront directement sur le code IR. De plus, le compilateur est plus modulaire, ce qui est un grand avantage. Certains compilateurs utilisent même plusieurs représentation intermédiaires afin d'améliorer la simplicité des optimisations et la maintenabilité du code (encore plus de modularité !).

Le *front-end* se décompose en une analyse *syntaxique*, elle même décomposée en un *lexer* et un *parser*, et une analyse *sémantique*. Le lexer va casser le programmes en tokens (lexèmes) lors que l'analyse syntaxique, qui seront par la suite arrangées dans un arbre de syntaxe abstrait (AST).

L'analyse sémantique se décompose en une phase de *binding* (lier les symboles à leur déclaration), une phase *EscapeChecker* (trouver les symboles d'une fonction définis dans une fonction parent par exemple dans le cas de fonctions imbriqués) puis un *TypeChecker* (vérification des types).

Le *middle-end* va traduire le langage vers le langage intermédiaire, qui permettra ensuite d'appliquer toutes les optimisations LLVM via `opt`.

1.2 Le langage Tiger

Il a été intégré dans le livre *Modern Compiler Implementation* par A. Appel. Il est impératif (ouf), typé selon deux types primitifs : *entier/integers* et *chaines de caractères/strings* (ainsi que *void*).

`let .. in ... end` sépare les déclarations des calculs. Les déclarations se font sous la forme suivante : explicite : `var a : int := 0`, ou implicite : `var b := 1` ou `var c := "hello"`. Des fonctions peuvent également être déclarées : `function get_temperature () : int = thermostat`. Les commentaires se font via les une syntaxe similaire au C : `/* commentaire */`.

Tiger gère également les récursions mutuelles (ce qui est embêtant pour les analyses du binder...). Attention, les fonctions mutuellement récursives ne peuvent être appelés que dans des blocs contigus de déclaration de fonctions (on ne peut pas déclarer de variables entre deux fonction s'appelant l'une l'autre).

Tiger comprend des primitives standards :

— print(s : string)

— print_int(i : int)

— ord(s : string) : int (valeur ASCII)

— chr(i : int) : string (inverse du précédent)

— size(s : string) : int

— et d'autres (concat, substring, not, exit, print_err) (liste explicite).

2 Arbre de syntaxe abstraite

2.1 Définition

Le rôle de l'AST est de donner une représentation que le programme peut comprendre. Les symboles et les mots peuvent avoir des sens différents en fonction de la phase de compilation. L'arbre ne doit pas stocker des informations superflues telles les espaces, les retours à la ligne et les commentaires.

2.2 Parcourir l'arbre

On va utiliser un visiteur. Un arbre est composé de noeuds de différents type : constant ou opération binaire. Un visiteur est un moyen d'évaluer un arbre en utilisant à chaque fois la fonction adaptée au type de l'opérande suivante.

On va utiliser deux méthodes : une `accept` côté arbre et une `visit` côté visiteur. Cela permet de pouvoir faire deux *double dispatch* : appeler une fonction différente en fonction du couple (type de noeud, type de visiteur).

2.3 Le lexer

Le but ici est de différencier les tokens. On va utiliser des expressions régulières. On utilisera *flex* (et par la suite *bison*) pour générer un automate, et même des sous-automates (ou états) de règles (pour gérer les commentaires ou les chaînes de caractère par exemple).

2.4 Le parser

Il sert à construire un arbre permettant un parcours simple du programme, car il est organisé de manière hiérarchique.

On va définir deux types de règles : les *terminaux* et les *non-terminaux*. Les terminaux sont les token du lexer, les non-terminaux sont produits par les règles de grammaire de la forme $\alpha \rightarrow \beta_1 \beta frm - e ... \beta_k$.

Pour déclarer des règles dans Bison, on utilise la syntaxe suivante :
`varDecl := VAR ID typeannotation ASSIGN expr {$$ = new VarDecl(@1, $2, $5, $3);}` `@` est la position dans le code du mot-clef parsé (cela permet de décorer l'arbre pour donner des informations de débugage), et `$i` est le i-ème argument de la règle parsée (β_i).

Il faut ensuite déclarer le type retourné par `$$` via : `% Type < VarDecl *> varDecl`.

On peut écrire même des règles récursives !

```
    %type <std::vector<Expr *» expr nonemptyexpr;
...
seqExp := LPAREN exprs RPAREN;
exprs := /*empty*/ | nonemptyexpr
```
avec
```
nonemptyexpr := expr { $$ = std::vector<Expr *>($1);}
| nonemptyexprs SEMICOLON expr /* récursif !*/ { $$ = std::move($1); $$push_back($3);};
```

Il peut y avoir des ambiguïtés sur les règles : $4+2\times 3$ doit-il être compris $(4+2)\times 3$ ou $4+(2\times 3)$? On doit en informer Bison : il faut définir la *priorité* des opérateur et leur associativité $(4+2+3)$.

Pire, il existe des grammaires intrinsèquement ambiguë, ou plusieurs arbres peuvent être valides. Il faut donc les éviter au maximum !

Pour appliquer les règles de priorité dans le bon ordre selon les priorité, Bison utilise un automate à pile.

2.5 Théorie : les parseurs SLR

Simple Left-to-right Rightmost derivation

On définit une grammaire G par une règle de départ S. Par exemple : les terminaux sont $a, b, \#$ ou $\#$ est la fin du fichier, avec les règles suivantes :

— $S \to T\#$

— $T \to aTbT$

— $T \to U$

— $U \to a$

On réalise pour tout mot une *dérivation*. On suppose qu'un curseur est présent sur les règles pouvant être appliqué. Cela consiste en réaliser la clôture transitive et empiler un à un les non-terminaux pouvant être appliqués. On va ensuite lire un caractère, puis avancer le curseur et supprimer les règles. On peux ensuite réitérer le processus tant qu'il reste des caractères et des lettres. On effectue ainsi des *shift* et des *reductions*.

On définit le *Follow Set* comme l'ensemble des symboles qui peuvent suivre un non-terminal. On effectue les réduction en fonction des follow sets, qui ne sont pas ambiguë pour un SLR.

Par défaut, Bison utilise un LALR, qui est une extension un peu plus puissante que SLR. Le fichier en sortie donne l'automate dans `bison-report.txt`, qui donne les conflits shift/reduce et reduce/reduce.

3 Analyse Sémantique

3.1 Variables

Une variable est une entité dans laquelle on stocke une valeur. On peut la stocker en RAM ou dans un registre, ou encore intermittente entre RAM et registres.

En Tiger, les variables sont déclarées avec le mot-clef `var` dans la première partie d'une expression `let`, avec un type implicite ou explicite.

On définit la *durée de vie d'une variable* et sa *portée* (*scope*). Cela contient toutes les variables vivantes à ce moment là du programme. Dans Tiger, les scopes sont crées par les `for`, `let` et déclaration de fonctions. On recherche les variables du scope le plus proche vers le plus lointain pour éviter les phénomènes de *masquage*, lorsque l'on utilise des variables de même nom. On ne peut donc pas utiliser un gros tableau $var \to values$.

Il existe également des scopes sur les fonctions. Sauf que les fonctions peuvent être mutuellement récursives : on ne peut donc pas analyser ligne par ligne en ne se basant que sur les fonctions déjà rencontrées. Au sein d'un bloc contiguë de fonctions, *toutes les fonctions sont visibles* !

3.2 Le binder

Le binder vérifie que tous les objets utilisés ont été déclarées et associent leur valeur à leur déclaration. Dans l'AST, `VarDecl` représente une déclaration, `Identifier` une utilisation de la variable. Le binder est un

visiteur qui traverse l'arbre pour construire les scopes. Il va chercher les variables utiliser dans `Identifier` en cherchant dans le scope le plus proche. Il associe ensuite à chaque `Identifier` une référence vers le bon `Vardecl`, ou renvoie une erreur lorsque cela n'est pas possible.

On va représenter les scopes par un `chain map` : une pile (liste chainée) de map $var \to value$.

On va réaliser la même chose pour les fonctions.

En sortie, on obtient un AST décoré par les déclarations. On a donc un lien vers la déclaration, qui est l'identifiant unique de l'emplacement mémoire utilisé.

Soucis Il peut y avoir des fonctions récursives !

3.3 Les frames

Il faut donc conserver en mémoire les environnement des fonctions et allouer des espaces mémoires différents pour chaque variable locale d'une fonction. Il faut donc une structure pour chaque appel de fonction, et donc une structure de taille arbitrairement grande : on choisit une pile, où *frame*. Historiquement, la pile descend alors que le tas monte. Il ne faut *surtout pas* confondre *frame* et *scope*. Un scope est lexical (blocks du programme), alors que les scopes sont présent à l'exécution.

Une frame associe donc à chaque `varDecl` sa `Value`. Chaque frame va contenir un pointeur `up` contenant l'adresse de la frame parente ainsi que les valeurs des variables concernées. Le binder va associer chaque variable à sa `varDecl`, mais la fonction va empiler ses variables au runtime. On ne va regarder que les variables locales déclarée dans la précédente frame. Sauf que ce n'est pas le cas : il faut pouvoir remonter ! D'où le pointer `up`, qui indique dans quel scope regarder pour accéder à une variable non déclarée dans la frame courante. Le pointeur `up` pointe donc vers la fonction *contenante* et *attention ce n'est PAS la fonction appelante* (cas des fonctions récursives justement !). Cela s'appelle un *static link*.

Remarque Il existe une second manière de faire, plus courante dans les langages fonctionnels. On rajoute des arguments fantômes aux fonctions, qui représentent l'intégralité des variables utilisées (on convertie les variables extérieures en variable d'arguments).

On rajoute dans l'AST une *profondeur* qui grandit avec les déclaration imbriquées de fonction. La différence de profondeur entre l'utilisation d'une variable/fonction et sa déclaration donne alors le nombre de frame à remonter via les pointeurs `up` pour obtenir la frame contenant la déclaration de la variable. On parle alors de *variable échappée*.

Problème On ne connait pas à la compilation l'adresse de la frame appelante ! On va donc le rajouter comme argument fantôme à chaque fonction, représentant l'adresse du cadre contenant. Dans le cadre d'un fonction récursive, le même pointeur est donc passé à chaque appel ! Le cadre à passer est choisi statiquement : si la fonction est déclarée à la même profondeur, on passe un pointeur vers notre propre frame, sinon lorsque la fonction est définie k niveau au-dessus, on passe une pointeur vers la frame un niveau au-dessus du notre.

3.4 Le typage

Tiger est un langage *statiquement typé*, c'est-à-dire que le type de chaque objet est bien connu à la compilation. Le type peut être définit de manière explicite ou implicite, mais il reste néanmoins présent.

Le type checker a donc deux rôles : faire de l'inférence de type et de la vérification de type. Il existe trois types en Tiger : entier, string et void (ne retourne rien). A chaque entité est associé un type, et la cohérence des types doit être effectuée. Le type checker passe après les binder (il faut d'abord connaître les symboles !), et s'implémente par - ô surprise - un visiteur.

Par exemple

— '+' n'opère que sur les entiers

— '>' doit prendre soit deux entiers soit deux strings

En Tiger, toutes les expressions (litterals) ont un type, par exemple `if/then/else`. La règles est donc `if` *entier* `then` *type1* `else` *type1*. On remplace alors `if/then` par un `if/then/else` avec le else vide (et donc then prend une expression de type void.

Le typage d'une fonction est toujours explicite : soit on met rien (type void), soit on précise son type. De même, le type de chaque argument doit être explicite.

Avec ces informations, ont peut ne typer qu'en une seule passe (type cascade).

Dans certains langages, on ne connait pas à l'avance les types (mais on les types statiquement quand même). On va rassembler les objet de même type dans une *clique*, et on assigne un type à la clique dès que l'on peut. Si un conflit de type arrive, une erreur se produit.

4 La représentation intermédiaire

4.0.1 Introduction

Une représentation intermédiaire doit être simple et bas-niveau : on retire tous le sucre syntaxique. Il doit également être facilement optimisable, c'est à dire qu'il doit subsister suffisamment d'information pour réaliser des "raccourcis" dans le code. Il y a donc un compromis à faire. Par exemple, doit-il y avoir un type tableau ? Si oui, on peut facilement savoir si deux accès sont sur le même tableau, pratique pour optimiser. Si non, on ne peut pas faire aisément des optimisations comme la propagation de constantes ou la réduction de force.

Comment représenter les variables ? On peut utiliser ou les registres virtuels ou via une pile. LLVM utilise des registres, permettant une analyse de dépendance simple et des optimisations plus faciles. Pour une pile, la génération de code et l'interprétation est plus simple (pas de notions de nommage), ce qui est le cas des machines virtuelles Java.

Une autre question est si la représentation du langage doit être plat ou hiérarchique (doit-on garder une structure d'arbre ?).

On va devoir faire du *lowering* ou *concrétisation*, qui est le contraire de l'abstraction : transformer un AST en code IR. Par exemple, on va traduire des expressions en code 3-adresses. L'objectif final étant de se rapprocher au maximum de l'assembleur.

Sur GCC, il y a plusieurs niveau d'IR : Generic → Gimple (registre + hybride hiérarchique/plat) → Gimple SSA → RTL (Register Transfer Langage, hybride Pile/Registre + hiérarchique, très proche du langage machine).

4.1 Le projet LLVM

A la base, LLVM est un projet académique qui avait pour but d'exécuter dans une machine virtuelle du code IR, puis on pouvait l'optimiser et la transcrire en assembleur. Le but était d'être modulaire au maximum. LLVM a été créé au début des années 2000, il a donc bénéficié de nombreuses expériences au niveau des langages (LLVM est en C++ vs GCC en C) et des théorie sur la compilation.

Apple a mis beaucoup de ressources pour étendre le projet qui est devenu académique et industriel (problématiques de licence et de maintenabilité), et plus recemment Google.

4.2 LLVM IR

Il est prévu pour être compréhensible par l'humain. Il comprend le type void et i1/i8/i32 les types entier sur 1/8/32 bits. Il existe des types dérivé : i32* pour les pointeurs, void (i8) une fonction prenant en argument un i8 et retournant void ; ainsi que i8,i16 une structure comprenant un i8 et un i16. @ dénomine les symboles

globaux (visibles par l'extérieur). Il *faut* une fonction `main`. Pour allouer sur la pile, on utilise `alloca`. Pour lire et écrire depuis un pointeur, on utilise `store` et `load`. On ne *peut pas* directement utiliser le pointeur !

Pour les variables locales, on va directement les allouer sur la pile avec `alloca`. cela n'est peut-être pas optimal, mais il existe une optimisation *mem2reg* qui permet de supprimer de code inutile.

On décompose le langage en *Basic Block*, unité du code sans branchement, que l'on peut relier par le *graphe de flot de contrôle*.