Compilation

AST et RISC : la grande traversée

Jules Chouquet

SOM2IF15 – 2024

Où on en est :

D'où l'on part : Un arbre de syntaxe abstrait correct, cohérent, dont on s'est assuré des bonnes propriétés sémantiques (le typage, la cohérence des déclarations, en particulier).
Où l'on va : Le langage adapté aux machines d'assemblage[1]. Mais on va se donner une structure intermédiaire.

Objet de ce cours

Définir le langage intermédiaire, et voir comment y plonger l'AST.

1. Par exemple RISC, mais pas seulement

L'intérêt d'une représentation intermédiaire

En gros

Extrait du manuel d'A.Appel [2] :

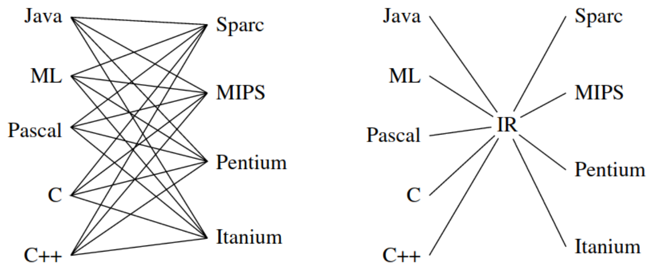

FIGURE 7.1. Compilers for five languages and four target machines: (a) without an IR, (b) with an IR.

2. *Modern compiler implementation in Java* (deuxième édition, 2002, Cambridge University Press)

En assembleur

Les expressions ne sont pas composées, ce sont :

- des registres
- des adresses
- des constantes
- des étiquettes d'instruction

Les instructions ne disposent pas de structure de contrôle :

- copie d'un registre vers la mémoire
- copie de la mémoire vers un registre
- enregistrement dans un registre de op e e' où e et e' sont des expressions
- saut (conditionnel ou non)

Quel langage ?

Un compromis

Il faut que le langage soit facilement traduisible en code assembleur (mais sans connaître à l'avance l'architecture précise du code cible). Plusieurs questions se posent pour la conception de cette représentation intermédiaire :

- Les procédures sont elles explicites ? (structure pour les cadres d'activation ?)
- Les expressions peuvent-elles être composées ?
- Comment représenter les registres et la mémoire de façon abstraite (mais pas trop non plus) ?
- Le typage intervient-il dans cette représentation ?

Plusieurs granularités possibles

Un exemple : le code trois adresses

On autorise uniquement des instructions de la forme :

```
t := t1 op t2
```

- Les instructions ressemblent à l'assembleur : facile à traduire vers le code cible.
- On ne fait pas d'hypothèse sur l'architecture, le nombre et la taille des temporaires : on reste modulaire.
- La traduction de l'AST vers du code trois adresses peut être complexe : comment traduire les définitions et appels de fonctions ?

Notre langage intermédiaire : Un peu plus de structure

Mais pas beaucoup

On autorise les expressions composées.[3] (Notez l'absence de variables.)

```
E::= Int|Byte|Temp|Read(Temp)|Unary(op,E)|Binary(op,E,E)
```

3. Leur décomposition est donc reléguée à la génération finale de code. Cela permet certaines optimisations.

Notre langage intermédiaire : Un peu plus de structure

Mais pas beaucoup

On autorise les expressions composées.[3] (Notez l'absence de variables.)

```
E::= Int|Byte|Temp|Read(Temp)|Unary(op,E)|Binary(op,E,E)
```

Et les instructions ?

```
I::= CJump(E,Label,Label)|Jump(Label)
    |ProcCall(Frame,List<E>)|WriteTemp(Temp,E)
```

Pour les fonctions : On se donne une structure pour les appels de procédure correspondant aux cadres d'activation de la pile d'appels. Ce sera la classe `Frame`.

3. Leur décomposition est donc reléguée à la génération finale de code. Cela permet certaines optimisations.

Une difficulté

Les expressions du langage source ne sont pas forcément des expressions de la représentation intérmédiaire. (une expressions comme `f(42)` n'a pas d'équivalent dans `E`).

Mais alors (me direz-vous) que doit renvoyer le traducteur quand il visite une expression ?

4. Merci de me l'avoir posée

Une difficulté

Les expressions du langage source ne sont pas forcément des expressions de la représentation intérmédiaire. (une expressions comme `f(42)` n'a pas d'équivalent dans `E`).

Mais alors (me direz-vous) que doit renvoyer le traducteur quand il visite une expression ?

Excellente question [4] : une classe `Result` dont le constructeur sera surchargé de façon à ce qu'elle puisse être instanciée avec :

- Une expression (`E`)
- Une instruction (`I`)
- Les deux

4. Merci de me l'avoir posée

Une difficulté

Les expressions du langage source ne sont pas forcément des expressions de la représentation intérmédiaire. (une expressions comme `f(42)` n'a pas d'équivalent dans `E`).

Mais alors (me direz-vous) que doit renvoyer le traducteur quand il visite une expression ?

Excellente question [4] : une classe `Result` dont le constructeur sera surchargé de façon à ce qu'elle puisse être instanciée avec :

- Une expression (`E`)
- Une instruction (`I`)
- Les deux

Le traducteur sera donc une implémentation de `ast.Visitor<Result>`

4. Merci de me l'avoir posée

Temporaires et mémoire

Dans la représentation intermédiaire, le nombre de temporaires disponibles n'est pas borné. Dans la vraie vie, c'est faux (voir TP5 et registres MIPS32).

A-t-on besoin de mentionner la **mémoire** au niveau de la représentation intermédiaire ?

5. tant qu'on n'a pas introduit les tableaux, chaînes de caractères,. . .

Temporaires et mémoire

Dans la représentation intermédiaire, le nombre de temporaires disponibles n'est pas borné. Dans la vraie vie, c'est faux (voir TP5 et registres MIPS32).

A-t-on besoin de mentionner la **mémoire** au niveau de la représentation intermédiaire ?

Non [5]. Pour le reste, ce sera lors de la génération finale de code que les temporaires pourront être affectés à des registres ou à des emplacements mémoire (utilisation de la pile, etc...).

5. tant qu'on n'a pas introduit les tableaux, chaînes de caractères,...

Temporaires et mémoire

Dans la représentation intermédiaire, le nombre de temporaires disponibles n'est pas borné. Dans la vraie vie, c'est faux (voir TP5 et registres MIPS32).

A-t-on besoin de mentionner la **mémoire** au niveau de la représentation intermédiaire ?

Non [5]. Pour le reste, ce sera lors de la génération finale de code que les temporaires pourront être affectés à des registres ou à des emplacements mémoire (utilisation de la pile, etc...).

A-t-on besoin de spécifier un **type** pour les valeurs contenues dans les temporaires ?

---

5. tant qu'on n'a pas introduit les tableaux, chaînes de caractères,...

Temporaires et mémoire

Dans la représentation intermédiaire, le nombre de temporaires disponibles n'est pas borné. Dans la vraie vie, c'est faux (voir TP5 et registres MIPS32).

A-t-on besoin de mentionner la **mémoire** au niveau de la représentation intermédiaire ?

Non [5]. Pour le reste, ce sera lors de la génération finale de code que les temporaires pourront être affectés à des registres ou à des emplacements mémoire (utilisation de la pile, etc...).

A-t-on besoin de spécifier un **type** pour les valeurs contenues dans les temporaires ?

Oui. Pour la taille qu'elles occupent en mémoire (`int`, `byte`, `address`).

5. tant qu'on n'a pas introduit les tableaux, chaînes de caractères,...

Déclaration

Comme au TP5 : une `Map<String,Temp>` reçoit une nouvelle entrée à chaque déclaration.[a]

- Création d'un nouveau temporaire
- Enregistrement de l'association dans la `Map`
- Ajout du temporaire au *cadre d'activation* courant
- C'est tout (la traduction retourne un code *vide*)

---

a. Mais chaque variable est associée à un bloc, comme pour la vérification de types. Donc en fait c'est une `Map<Pair<String,Block>,Temp>`

Affectation

- Récupérer la traduction de l'expression.
- Récupérer le temporaire associé au nom de la variable[a].
- Ajouter au code l'instruction `WriteTemp` adaptée et le retourner.

a. Nous avons déjà vérifié lors des phases précédentes que la variable *doit* avoir été déclarée.

Traduction des expressions
Hors appels de fonction

Entiers : immédiate. **Variables** : recherche dans la `Map`.

Expressions binaires : facile, en visitant les deux sous expressions, on obtient `resultLeft` et `resultRight`. On crée une nouvelle expression `exp=Binary(resultLeft.getExp(),op,resultRight.getExp())`.

mais n'oublions pas que les calculs des expressions gauches et droites peuvent nécessiter des *instructions*. Donc, on récupère aussi ce code : `resultLeft.getCode()` [6], `resultRight.getCode()` que l'on concatène dans une nouvelle liste code.

Pour finir, on retourne quelque chose comme `new Result(exp,code)`.

Expressions unaires : ...

6. code est une liste d'instructions de la représentation intermédiaire.

Traduction des instructions

Linéarisation

Listes d'instructions : on se contente de concaténer le code produit par chaque traduction.

conditionnelle : `if(e) b1 else b2;` On visite les trois composantes, pour récupérer `result_e`, `result_b1`, `result_b2`.

On crée trois nouveaux `Label` : `L1`, `L2`, `L3`.

Le code intermédiaire créé ressemble à la séquence suivante :

- `CJump(result_e.getExp(),L1,L2)`
- `L1: result_b1.getCode(); goto L3`
- `L2: result_b2.getCode(); goto L3`
- `L3:`

boucle while : ...

Traduction des fonctions

Une première approche

Traduire le code du bloc de la fonction, et utiliser des temporaires pour stocker les paramètres :
On crée deux nouveaux labels `L` et `L'`. `t f(a1,a2,...){ inst }` est traduit en un `Result result_f` contenant le code de la fonction, de la forme suivante :

`L : [code]`[7]`; goto L';`.

`f(x1,x2,...)` est ensuite traduit en :

- `r1,r2,...` sont les temporaires associés à `a1,...,a2` dans `result_f`.
- `WriteTemp(x1,r1)`
- `WriteTemp(x2,r2)`
- `goto L`
- `L': ...`

7. le code contient une instruction comme `WriteTemp(e,tempReturn_f)`

Un problème avec l'approche précédente

8. En tout cas pas tout le temps

Un problème avec l'approche précédente

Ça marche pas [8]

8. En tout cas pas tout le temps

Les cadres d'activation

On se donne une structure `Frame` permettant d'enregigstrer toutes les informations nécessaires au calcul associé à un appel de fonction.

Ses champs :

- `parametres` (une liste de temporaires)
- `variables locales` (idem)
- `output` (un temporaire)
- `entrée` (un Label)
- `sortie` (idem)
- `taille` (un entier : correspondra au nombre d'octets occupés par la donnée)

Les cadres d'activation

Création et gestion

Dans le traducteur en code intermédiaire, une `Map<String,Frame>` est créée, et remplie par un visiteur `FrameBuilder`[9] : pour chaque définition de fonction, les champs évoqués plus haut sont affectés et une entrée est ajoutée à la `Map`.
Dans ce `FrameBuilder`, on peut aussi ajouter des fonctions prédéfinies (pour `print` par exemple).

9. Qui hérite du visiteur générique et ne réécrit que la visite des définitons de fonctions.

Traduction des fonctions

Définitions

- On récupère le `Frame` associé au nom de la fonction lors de l'étape précédente.
- On calcule le `Result` associé au corps de la fonction (donné par la visite du bloc).
- On ajoute la paire `Pair<frame,result>` à une structure globale. (liste contenant une telle paire pour chaque définition de fonction du programme).

Traduction des fonctions

Appels

- On récupère les arguments sous forme de `Pair<List<E>,List<I> >` : `args,code` (on a une expression par paramètre, et une liste d'instructions dont on ne connaît en revanche pas la taille).
- On récupère la signature de la fonction (on a accès à la table des symboles construite à la phage précédente)[10]
- On récupère le `Frame` associé à la fonction (*via* le `FrameBuilder` discuté plus haut).
- On retourne `funCall(typeRetour,frame,args,code)`

10. En fait on n'a besoin que du type de retour de la fonction.

- On récupère les arguments sous forme de `Pair<List<E>,List<I> >` : `args,code` (on a une expression par paramètre, et une liste d'instructions dont on ne connaît en revanche pas la taille).
- On récupère la signature de la fonction (on a accès à la table des symboles construite à la phage précédente) [10]
- On récupère le `Frame` associé à la fonction (*via* le `FrameBuilder` discuté plus haut).
- On retourne `funCall(typeRetour,frame,args,code)`

Mais qu'est-ce que `funCall` ?

10. En fait on n'a besoin que du type de retour de la fonction.

Que retourne l'appel de fonction ?

- Crée un nouveau temporaire `temp` (adapté au type de retour de la fonction)
- Ajoute ce temporaire au `frame` courant[11].
- Crée une instruction
- Retourne un `Result` contenant :
  - L'expression `ReadTemp(Temp)`
  - Le code correspondant aux arguments (instructions), récupéré plus tôt
  - L'instruction `Call(Temp,frame,args)`.

11. Une variable globale `currentFrame` est accessible tout au long de la traduction

Traduction des instructions

`return e`

- Récupère `res`, le `Result` récupéré en visitant `e`.
- Récupère le temporaire `returnTemp` du cadre `currentFrame`
- Retourne les instructions suivantes :
  - `res.getCode()` (liste des instructions pour l'évaluation de `e`.)
  - `WriteTemp(returnTemp,res.getExp())`
  - `Jump(currentFrame.sortie)` (le `Label` correspondant à la sortie du cadre d'activation).

Bientôt sur vos écrans

Suite du cours : traduire le code intermédiaire en assembleur MIPS32.