SE202 : la représentation intermédiaire (IR)

Samuel Tardieu
Année scolaire 2016/2017

La représentation intermédiaire

Modèle

L'arbre syntaxique abstrait (AST)

- est dépendant du langage source ;
- permet de faire des analyses et des manipulations ;
- reste éloigné de l'architecture de la machine cible ;
- ne prend pas en compte l'ABI (*application binary interface*).

La représentation intermédiaire (IR)

- est indépendante du langage source ;
- utilise des instructions simples ;
- reste relativement générique ;
- prend en compte le modèle mémoire et l'ABI de la cible.

Exemple de traduction : les fonctions

Une déclaration de fonction

- mettra en place l'environnement (*frame*) de la fonction et sauvegardera certains registres (*callee-save*) ;
- récupérera les arguments dans les registres idoines ou la pile, selon l'ABI ;
- placera le résultat dans le registre correspondant à l'ABI.

Un appel de fonction

- placera les arguments aux endroits idoines ;
- sauvegardera les registres qui doivent être préservés (*caller-save*).

Les frames

À l'entrée dans une fonction, une nouvelle *frame* est créee pour pouvoir y placer tous les éléments qui doivent être stockés sur la pile (et pointée par le registre *frame pointer*) :

- les variables locales et valeurs intermédiaires qui ne peuvent être gardées dans des registres ;
- les variables sur lesquelles on prend un pointeur, qui doivent nécessairement être en mémoire, ou celles accessibles depuis des fonctions imbriquées.

Pour pouvoir accéder à une variable d'un niveau supérieur (champ *depth* différent de celui de la fonction courante), il faut utiliser une indirection : le *static link*.

Le static link

Le *static link* est une valeur qui représente la *frame* de niveau **statique** directement supérieur à celui de la fonction courante :

- quand on référence une variable de niveau supérieur, on remonte la chaîne de liens statiques pour trouver la *frame* contenant la variable ;
- quand on appelle une fonction imbriquée, on passe la *frame* courante comme nouveau *static link* ;
- quand on appelle une fonction de même niveau (appel récursif par exemple), on passe la valeur courante du *static link* comme nouveau *static link* ;
- quand on appelle une fonction plus externe, on remonte les liens statiques pour trouver le *static link* du bon niveau.

Représentation du static link

Pour un accès facile, le *static link* :

- est passé systématiquement comme premièr argument d'une fonction ;
- est stocké dans la *frame* courante en tout premier (à l'adresse du *frame pointer*).

Exemple 1

```
let
  function f() =
    let var a := 2
    in
      a := a + 1;
      a
    end
in
  f()
end
```

Ici, la variable `a` est une variable locale à la fonction. Elle peut être stockée dans un registre, le *static link* n'est pas utilisé.

Exemple 2

```
let var a := 2
    function f() = (a := a + 1; a)
in f() end
```

Ici, la variable `a` est une variable accédée depuis `f` mais déclarée en dehors. Depuis `f`, elle est accédée à travers le *static link*, qui permet d'accéder aux variables du niveau le plus haut.

Les nouveaux objets

En plus des nœuds assez proches de ceux de l'AST, l'IR utilise deux types d'entités qui ne sont pas des nœuds :

- Un `Label` désigne un nom, généré automatiquement, permettant d'identifier un endroit dans le code généré pour pouvoir, par exemple, s'y rendre à travers un saut. Un `Label` est utilisé comme argument d'un nœud `LABEL` (déclaration) ou `NAME` (référence).
- Un `Temp` désigne un registre virtuel, avec un nom arbitraire, qu'on fera ensuite correspondre lors de la génération de code avec un ou plusieurs registres ou emplacements physiques sur la pile. Un `Temp` est généralement utilisé dans un nœud `TEMP`.

Les types de nœuds de l'IR

Les nœuds de l'IR, dont les noms sont en majuscule (comme `CONST`) pour les distinguer facilement des nœuds de l'AST, sont regroupés en deux sous-hiérarchies :

- les `Sxp` (*Simple eXPression*) représentant des nœuds qui sont associés à un résultat ;
- les `Stm` (*StaTeMent*) représentants des nœuds associés à un effet sans résultat (une affectation par exemple).

Mais une traduction directe en profondeur de l'AST en IR pose des problèmes d'efficacité du code généré, il va falloir introduire d'autres mécanismes.

Le problème de la traduction directe

Un nœud de l'AST représentant l'expression `a < 3` peut venir de plusieurs contextes :

- une expression, renvoyant 0 ou 1, dont le résultat peut être utilisé dans un calcul, comme `4 + (a < 3)` ;
- une expression conditionnelle, comme dans `if a < 3 then f() else g()`.

Utiliser systématiquement la première traduction lors du parcours de l'arbre est inefficace. Il ne permettrait pas d'utiliser les sauts conditionnels présents dans les processeurs modernes, et obligerait à des normalisations (0 ou 1) inutiles.

Une solution

Pour pouvoir traduire l'arbre avec un parcours classique, on ne retourne pas directement un nœud lorsqu'on traite les nœuds enfants, mais une coquille (*Shell*), qui encapsule une représentation un peu plus abstraite du nœud. Ensuite, on sortira de la coquille la représentation qui nous intéresse selon le contexte.

Le *Shell* peut être de quatre types :

- `Cx` : représentation d'un opérateur de comparaison (opération, argument de gauche, argument de droite);
- `Ex` : représentation d'une expression classique ;
- `Ix` : représentation d'un opérateur ternaire (test, *Shell* si vrai, *Shell* si faux) ;
- `Nx` : représentation d'un statement.

Traduire un *Shell* en nœud IR

Les trois opérateurs suivants, qui s'appliquent sur les différents *Shell*, permettent d'obtenir l'arbre IR correspondant à l'opération utilisée :

- `unCx(ifTrue, ifFalse)` : si la valeur du *Shell* est un vrai logique, sauter à `ifTrue`, sinon sauter à `ifFalse` ;
- `unEx()` : renvoie la valeur correspondant au *Shell* ;
- `unNx()` : renvoie un statement correspondant à l'exécution de ce qui se trouve dans le *Shell* et ignore le résultat éventuel.

Par exemple, `unEx()` sur un nœud `Cx` (opérateur de comparaison) renverra un arbre calculant 0 ou 1. `unCx()` appelé sur le même nœud utilisera probablement un opérateur de saut conditionnel suivi d'un saut inconditionnel.

Conséquences d'utilisation du *Shell*

Sans les `Shell`, l'expression `if a < b then 4 else 5` serait traduite comme :

- mettre 1 dans une registre `r1` si a < b, 0 sinon ;
- si `r1` est différent de 0, mettre 4 dans le résultat, 5 sinon.

Avec le *Shell*, elle est traduite comme :

- mettre 4 dans le résultat si a < b, 5 sinon.

Autre exemple

Avec le *Shell*, on peut facilement traduire

```
result := if a < b & c < d then 4 else 5
```

en (pseudo-nœuds) :

```
CJUMP(">=", TEMP(a), TEMP(b), NAME(falseLabel))
CJUMP(">=", TEMP(c), TEMP(d), NAME(falseLabel))
MOVE(TEMP(result), CONST(4))
JUMP(NAME(endLabel))
LABEL(falseLabel)
MOVE(TEMP(result), CONST(5))
LABEL(endLabel)
```

Optimisations

On peut profiter de la traduction pour faire certaines optimisations, par exemple :

- Si la valeur d'un `Shell` est connue, `unCx()` peut directement générer un saut inconditionnel correspondant à la branche vrai ou la branche faux.
- Si la valeur de la condition est connue, `unEx()` sur un `Ix` (opérateur ternaire) peut remplacer par celle des deux expressions qui est appropriée.
- On peut lors d'une traduction construire un nœud plus facile à traduire. Par exemple `unEx()` sur un `Cx` (opérateur conditionnel) doit normaliser le résultat à 0 ou 1, et est écrit comme l'évaluation d'une condition ternaire :

```
def unEx(self):
    return Ix(self.frame, self,
              Ex(CONST(1)), Ex(CONST(0))).unEx()
```