TP n°6
Analyse sémantique

Création de la représentation intermédiaire

Florent Bouchez, Christoph Lauter
prenom.nom@ens-lyon.fr

18 avril 2007

Petit récapitulatif : jusqu'à présent, vous avez créé à partir d'un code initial écrit en `pascal-` une représentation abstraite de ce code sous la forme d'un arbre de syntaxe. Cet arbre est encore une représentation de haut niveau puisqu'il contient des structures avancées, par exemple des boucles `while`.

Le but de ce TP est de générer la représentation intermédiaire donnée aux TP précédents. On travaillera à la traduction de l'AST en IR par de l'analyse sémantique ; en effet, l'IR est assez proche du code assembleur et on doit donc épurer notre AST de tout le sucre syntaxique si pratique : les instructions conditionnelles sont découpées en le calcul de la condition, puis les `jump` requis, de même pour les boucles qui font un `jump` au début du code le la boucle, les accès aux tableaux ou aux champs des enregistrements deviennent des séquences de codes qui récupèrent les données aux bons endroits de la mémoire, etc.

1 Nouveaux fichiers de la semaine

À récupérer, comme d'hab, dans

```
/home/fbouchez/compilation/TP6_code_intermediaire
```

Interessons-nous d'abord au fichier principal qui vient de changer, `main.ml`. Comme on est vraiment sympas, on a fait le TP à l'avance et on vous a lancé notre `tp6` sur l'exemple canonique :

```
% ./tp6 ../example.pas > example.out
```

et avec la sortie dotty :

```
% ./tp6 -dot ../example.pas
String Label outofboundsmsg18: '../example.pas: l.12 char 7-15, index out of range\n'
String Label outofboundsmsg30: '../example.pas: l.13 char 9-17, index out of range\n'
Frame of search4:
        dotty in search4.dot
String Label outofboundsmsg36: '../example.pas: l.19 char 9-13, index out of range\n'
Frame of main:
        dotty in main.dot
```

Tadaaa ! Tout ça c'est presque magique, et c'est uniquement grâce à l'ajout de la ligne `Semantic.trans_ast ast ;` À condition évidemment que vous aillez bien terminé le TP précédent...Dans le cas contraire, ne le continuez pas tout de suite, mais complétez ce qu'il vous manque au fur et à mesure que vous progressez dans ce TP.

Au passage, petite note extraite de *Modern compiler implementation in ML* (A. Appel) :

> Life is too short to spend time chasing down irreproducible bugs, and money is too valuable to waste on the purchase of flaky software. When a program has a bug, it should detect that fact as soon as possible and announce that fact (or take corrective action) before the bug causes any harm.

C'est pour cette raison que l'on trouve les *string labels* ci-dessus : c'est facile pour le compilateur de générer des messages personnalisés *a*u cas où de fausses valeurs soient utilisées lors de l'exécution du programme. Vérifiez tout ce que vous pouvez, il sera toujours temps de proposer une option `-unsafe` pour éviter ces vérifications.

Voyons maintenant les autres fichiers :

- `env.ml{,i}` : module trivial où sont définis les différents types d'entrées possible dans les environnements ;
- `translate.ml{,i}` : autre module trivial qui gère les *fragments*. Les fragments seront plus tard transformés en morceaux "atomique" de code assembleur et sont définis dans à la fin de `frame.mli`. Ici, nous n'en avons que de deux sortes : les chaînes de caractères (fragment assembleur `data`) et les fonctions (fragment assembleur `text`) mais cela peut dépendre de votre choix d'architecture ;
- `predefined.mli` : contient les types et constantes prédéfinis du langage d'entrée, par exemple `integer` ou `false` ; en effet, l'utilisateur a la possibilité d'en créer de nouveaux voire même de les redéfinir, ils ne pouvaient donc pas faire partie des mot-clés du langage. La création du fichier `predefined.ml` correspondant est laissée à la discrétion des étudiants en guise d'échauffement ;
- `semantic.mli` : ce petit fichier n'a l'air de rien comme ça mais c'est lui qui fait tout le travail, par le biai de la fonction `trans_ast`, qui s'applique sur un AST, en vérifie les types et les symboles, puis crée les fragments correspondants. La suite du TP consistera à créer le `semantic.ml` correspondant.

2 Analyse sémantique

Il est conseillé de diviser les tâches que doit accomplir `trans_ast`. Convainquez vous qu'il serait stupide de décider d'effectuer d'abord la vérification des types, puis celle des symboles, puis de tout traduire vers la représentation intermédiaire.

Puisque vous êtes convaicus, vous allez tout faire en une seule passe et vérifier que les types sont cohérents et que les symboles utilisés existent bien pendant la phase générale de traduction de l'AST en IR. Nous vous proposons d'utiliser les fonctions suivantes, qui sont mutuellement récursives et qui vous permettront de paralléliser le TP avec votre binôme. Le type `expty` est défini dans `translate.mli` et est un enregistrement contenant la représentation intermédiaire d'une expression, et le type de la valeur que calcule cette expression.

- `val trans_var : frame -> venv -> tenv -> var -> expty`
  Cette fonction renvoie la représentation intermédiaire de l'accès à une variable `Ast.var`, utilisée dans la fonction dont le cadre dans la pile est `frame`, dans les environnements de variables et types `venv` et `tenv` ;
- `val trans_exp : frame -> venv -> tenv -> exp -> expty`
  Idem, mais traduit une expression de l'arbre de syntaxe abstraite : `Ast.exp` ;
- `val trans_ty : frame -> venv -> tenv -> ty_ty -> ty`
  Celle-ci traduit un type de l'ast (`Ast.ty_ty` en un type normal (`Types.ty` ;
- `val trans_dec : frame -> venv -> tenv -> dec -> unit`
  Fonction principale qui effectue la traduction d'une déclaration (`Ast.dec`). Elle est le point d'entrée, évidemment appelée par `trans_ast` sur la déclaration du programme principal.

2.1 `trans_var` et le type `PARREC`

Vous avez peut-être remarqué dans le TP précédent un type étrange nommé `PARREC` dont l'origine éthymologique est *Pointer on ARray or RECord*.

Explication du problème : Considérez un tableau de tableaux. Deux possibilités s'offrent à vous :

- stocker dans le tableau des pointeurs vers les adresses des autres tableaux ;
- linéariser le tableau et mettre en mémoire tous les sous-tableaux les uns à côté des autres.

La deuxième solution utilise moins de place en mémoire, mais elle a un inconvénient : alors qu'avec la première méthode, `x[1]` crée un nœud de l'IR de type `MEM`(calcul de l'adresse de la case 1), dans le deuxième cas, il faut d'abord savoir le type de `x[1]` :

- type de base (entier, booléen) : idem
- tableau (ou enregistrement) : l'adresse de la case 1 *est* l'adresse du tableau `x[1]`, donc pas de nœud `MEM`.

Pour savoir si il faut ou pas ajouter un noeud `MEM`, nous vous proposons d'utiliser le constructeur `Types.PARREC` qu'on ajoute devant toute construction de type `ARRAY` ou `RECORD` qui n'est pas englobée par un autre constructeur `ARRAY` ou `RECORD`. Le constructeur `PARREC` peut-être vu comme le type d'un *Pointer on ARray or RECord*. Ce constructeur suit ensuite l'analyse `trans_var` en se propageant vers une feuille du type. Enfin, si le type de la sous-expression retournée par `trans_var` est `PARREC (Q,t)` où `t` est un type de base (`STRING, INT, BOOL, POINTER _`) et `Q` est soit un constructeur de records ou d'arrays, il faut alors ajouter un noeud `MEM`.

Prenons un exemple, avec la variable suivante :
`var x : array[1..n] of array[1..n] of integers ;`
Voici des exemples de types lors d'accès à `x` :

accès	type	IR
`x`	`PARREC of ARRAY of ARRAY of INT`	e
`x[1]`	`PARREC of ARRAY of INT`	`BINOP (+,`e`,offset)`
`x[1][2]`	`INT`	`MEM(BINOP(+,BINOP (+,`e`,offset),offset'))`

2.2 Librairie d'appels

Il existe un certain nombre de fonctions de base en `pascal-`. Si l'on regarde `example.pas`, on y trouve `read` et `write` qui poseront des problèmes à notre analyseur sémantique :
`Fatal_error("../example.pas: l.19 char 4-14, read is undefined"`

Créez un module `Lib_mon_compilo_qui_dechire` dans lequel seront définies ces fonctions, notamment le nombre et le type des arguments pour la vérification des types.