

Rapport sur la première partie du projet de compilation

Lucas PESENTI

J’ai choisi d’implémenter le compilateur Mini Ada en OCaml, en utilisant les outils `ocamllex` et `menhir`. Le projet est pour le moment constitué de trois blocs : l’analyse lexicale — `lexer.mll` —, l’analyse syntaxique — `parser.mly` — et le typage — `typer.ml`. Le fichier `ast.ml` définit les différents types d’arbres de syntaxe abstraite manipulés par les autres composants, `exceptions.ml` déclare les exceptions et `main.ml` coordonne le tout.

1 Analyse lexicale

Lors de la phase d’analyse lexicale, le fichier source est découpé en lexèmes, et les commentaires sont supprimés. Ceux-ci sont gérés dans une règle particulière `comment` qui commence avec les caractères `--` et se termine à la fin de la ligne (ou du fichier).

Tous les identificateurs sont transformés en minuscules. On a veillé à garder exactement la même liste de mots-clés que dans l’énoncé ; les cas de `character_val` et `Ada.Text_IO` sont traités à part. Il a été choisi arbitrairement, comme ce n’était pas précisé dans l’énoncé, de laisser une flexibilité au niveau de la casse et les caractères blancs pour le premier, mais pas pour le second.

Les littéraux caractères sont filtrés par `['\x00' - '\x7f']`. Enfin, on veille à n’accepter que les constantes numériques entre 0 et 2^{31} . On suppose pour cela — quitte à limiter temporairement la portabilité — que le compilateur est exécuté sur une architecture où un `int` permet effectivement de stocker cet intervalle de valeur¹. On commence donc par vérifier que la

¹Ce n’est donc pas le cas d’un système 32 bits classique.

valeur stockée par la chaîne est représentable dans un entier signé OCaml en utilisant `int_of_string`, puis on effectue les vérifications nécessaires :

```
1 try
2   let n = int_of_string s in
3   if n > 1 lsl 31 then failwith "";
4   INT n
5 with _ ->
6   raise (Lexing_error ("too big integer constant " ^ s))
```

2 Analyse syntaxique

L'analyseur syntaxique renvoie un arbre de syntaxe abstraite décoré par des informations de localisation, à savoir les positions de début et de fin des lexèmes correspondants, fournies par l'analyseur lexical. Les différentes unités syntaxiques choisies pour représenter le langage sont :

- `file` : le programme complet ;
- `decl` : une déclaration de type, de variable ou de fonction ;
- `field` : un champ d'enregistrement ;
- `stype` : une annotation de type ;
- `param` : un paramètre de fonction ou de procédure ;
- `mode` ;
- `stmt` : une instruction ;
- `access` : un accès à une variable ;
- `binop` : un opérateur binaire ;
- `expr` : une expression ;
- `ident` : un identificateur.

Lors de cette phase, on regroupe le cas des fonctions et des procédures en considérant le type fictif `STunit`. On transforme également certaines listes : par exemple, on considère en sortie qu'un enregistrement est constitué d'une liste associant son type à chaque identificateur. De même, les constructions de `if ... elsif ... else` sont transformées en `if ... else` en utilisant des constructions de blocs.

On effectue la vérification supplémentaire demandée à chaque déclaration de fonction ou de procédure qui se termine par un identificateur grâce à la fonction `check_same_identifiers` :

```
1 let check_same_identifiers (i1, _) o2 = match o2 with
2   | Some (i2, loc) when String.lowercase i1 <>
3                       String.lowercase i2 ->
4       raise (Different_ids (i2, loc))
5   | _ -> ()
```

3 Typage

Le typeur prend en entrée l'arbre produit par l'analyseur syntaxique et renvoie un arbre de syntaxe abstraite décoré par les types, où `stype` est remplacé par `typ`. Les différents types considérés sont les suivants :

```
1 type typ =
2   Tint | Tchar | Tbool | Trecord of tident | Taccess of
    tident | Tnull | Tunit
```

Le type `tident` auquel la définition précédente fait référence est la donnée du nom à proprement dit et d'un entier, son niveau de déclaration (la profondeur d'imbrication de la fonction dans laquelle il est déclaré). Ceci permet d'identifier de manière unique un identificateur dans l'état courant du typeur, en prenant en compte la possibilité qu'il existe en même temps plusieurs noms faisant référence à des objets différents (masquage).

L'environnement de typage est stocké avec les informations suivantes.

- `dec_vars` : associe à chaque variable déclarée dans l'environnement courant son type.
- `dec_types` : associe à chaque variable de type déclarée le type qu'elle dénote. S'il n'y a pas de déclaration anticipée, les structures sont automatiquement déclarées lorsqu'elles sont définies. Les types intrinsèques comme `int`, `character` ou `boolean` sont également gérés de cette manière.

- **def_recs** : associe à chaque variable de type enregistrement définie une table associative qui relie chaque identificateur de champ à son type.
- **def_funs** : associe à chaque fonction définie son type de retour (éventuellement **Tunit** pour une procédure), la liste du type et du mode de chacun de ses arguments.
- **const_vars** : variables constantes. Une expression formée à partir d'une d'entre elles n'est pas une lvalue. Cela concerne les paramètres **In** et les compteurs de boucle **for**.
- **for_vars** : ensemble des compteurs de boucle **for**. Il est utilisé pour détecter les cas erronés comme **for i in 1 .. i**.
- **idents** : associe à chaque identificateur un type de dernière déclaration (parmi déclaration de variable, déclaration de variable de type, définition de structure et définition de fonction) et le niveau de cette déclaration.
- **return_value** : type de retour de la fonction courante (si cela a un sens).
- **level** : niveau d'imbrication courant.
- **nb_incomplete** : nombre de déclarations incomplètes (ie. non suivies d'une définition).

Des environnements fictifs sont régulièrement créés, de manière à simuler correctement le masquage des identificateurs, par exemple quand un identificateur de champ ou de variable masque son type lors de sa déclaration. Il est donc fait intensément usage de la non mutabilité des tableaux associatifs en OCaml.

Le typeur supporte la totalité de Mini Ada, mais est construit dans l'optique d'une généralisation à Ada. Par exemple, on pourrait rajouter sans trop de difficulté le support de types **access** sur des types plus généraux que des enregistrements, ou encore des déclarations anticipées de types **access**. Cependant, des vérifications supplémentaires sont faites,

La règle selon laquelle toute fonction doit se terminer par un `return` m'a également semblé assez agressive par rapport à ce qu'Ada autorise. De nombreuses questions se posent quant au suivi du flot de contrôle, par exemple lorsque la fonction se termine par une boucle ou une condition. Ainsi, dans le cas où les deux blocs d'instructions suivant `if` et `else` se terminent par un `return`, le compilateur ne produit pas d'erreur.

La description des erreurs gagnerait à être encore plus précise : en particulier, l'exception `Undeclared` est utilisée même lorsque l'erreur est en réalité causée par un masquage, ce qui n'est pas le comportement des compilateurs courants.