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'arbre de syntaxe abstraite manipulés par les autres composants, `exceptions.ml` déclare les exceptions et `main.ml` coordonne le tout. La compilation est assurée par `ocambuild`, qui est appelé par `make`.

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 convertis en minuscules. On a veillé tout particulièrement 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 des 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 valeur[1]. On commence donc par vérifier que la 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
```

[1] Ce n'est donc pas le cas d'un système 32 bits classique. On aurait par exemple stocker un littéral entier positif x par $-x$ afin de permettre artificiellement au compilateur de fonctionner correctement sur les machines en complément à 2 sur 32 bits.

```
with _ ->
  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` : un programme complet Mini Ada ;
- `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.

`decl`, `stype`, `stmt`, `access`, `expr` et `ident` admettent une forme `nom_loc` qui est en fait un alias de types pour `nom * loc`.

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 pour chaque déclaration de fonction ou de procédure qui se termine par un identificateur grâce à la fonction `check_same_identifiers` :

```
let check_same_identifiers (i1, _) o2 = match o2 with
  | Some (i2, loc) when String.lowercase i1 <>
                        String.lowercase i2 ->
    raise (Different_idents (i2, loc))
  | _ -> ()
```

3 Typage

Le typeur prend en entrée l'arbre produit par l'analyseur syntaxique et renvoie un arbre de syntaxe abstraite décoré par des types. De plus, toutes les références à des types annotés par l'utilisateur `stype` sont remplacées par `typ`. Les différents types considérés sont les suivants :

```
type typ =
  | Tint | Tchar | Tbool           (* Types intrinseques *)
  | Trecord of tident                 (* Enregistrement *)
  | Taccess of tident                          (* Acces *)
  | Tnull                                   (* typenull *)
  | Tunit   (* Type fictif pour les retours de procedure *)
```

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 de caractériser de manière unique un identificateur dans l'environnement courant du typage, en prenant en compte la possibilité que subsistent 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 declarée dans l'environnement courant son type.
- `dec_typs` : associe à chaque variable de type declarée le type qu'elle dénote. S'il n'y a pas de declaration anticipée, les structures sont automatiquement délarées 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. À titre d'exemple, l'ajout d'un identificateur se déroule de la manière suivante :

```
(*
 * Ajoute dans idents un identificateur (i, loc) dont la
 * declaration est de type dec_type dans le niveau level.
 *)
let add_ident (i, loc) dec_typ level idents =
  try
    (* S'il n'existe pas, on l'ajoute *)
    let (prev_dec_typ, prev_level) = Smap.find i idents in
    (* Si i est dans la liste des identificateurs
       reserves (put et new_line) *)
    if List.mem i reserved_idents then
      raise (Reserved_ident (i, loc))
    (* Si il y a un masquage strict, on ajoute *)
    else if prev_level < level then
      raise Not_found
    (* Si c'est une structure declaree puis definie, on
       ajoute *)
    else if prev_dec_typ = Dtyp_typ_dec &&
            dec_typ = Dtyp_rec_def then
      raise Not_found
    (* Sinon, ca fait deux identificateurs de meme nom au
       meme niveau *)
    else
      raise (Already_declared (i, loc))
  with Not_found ->
     Smap.add i (dec_typ, level) idents
```

Il est donc fait intensément usage de la non mutabilité des tableaux associatifs — module `Map` de 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 déclaration anticipées de types `access`.

Conformément au comportement d'Ada, deux types enregistrements ne sont considérés comme identiques — et sont donc compatibles, au sens des opérations qu'on leur applique — que s'ils correspondent à la même déclaration — et pas s'ils contiennent les mêmes champs, par exemple.

La règle selon laquelle toute fonction doit se terminer par un `return` m'a 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 tous les blocs d'instructions suivant `if .. elsif .. 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. De plus, certaines informations sont perdues dès les premières étapes de l'analyse du compilateur : par exemple, la casse des identificateurs dans les exceptions ne correspond pas nécessairement à celle utilisée dans le programme[2].

[2]Une amélioration consisterait donc à représenter un identificateur dans l'arbre de syntaxe abstraite en sortie de l'analyseur syntaxique comme un triplet : son nom, son nom en minuscules et sa localisation. Le précalcul du deuxième élément permet de réduire le coût d'une conversion en minuscules à chaque utilisation du nom dans le typeur.