Compilateur Mini Ada

Florentin Guth

9 décembre 2016

A new, a vast, and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible.
Ada.

Table des matières

1 Lexer et Parser

1.1 Lexer

On utilise un table de hachage qui associe à une chaîne de caractères en minuscules le `token` correspondant au mot-clé en question. Tous les identifiants sont ainsi convertis en minuscules. On ne traite pas `Put` et `New_Line` comme des mots-clés afin de permettre à l'utilisateur de les redéfinir (ça n'aurait pas beaucoup de sens de devoir connaître les noms de toutes les fonctions d'une bibliothèque que l'on utilise pour ne pas avoir de conflits...). Ils sont donc traités exactement comme des procédures déclarées par l'utilisateur.

`Character'Val` n'est pas traité à part non plus : il est converti en la liste de tokens : `[IDENT "character"; QUOTE; IDENT "val"]`.

On prend garde de ne reconnaître que les constantes entières de $[\![0, 2^{31}]\!]$ et les caractères ASCII 7 bits.

Enfin, on a rajouté le mot-clé `all`, qui permet de déréférencer explicitement un type `access` (si `X` est de type `access T`, alors `X.all` est de `type T`).

1.2 Parser

On utilise les tokens « fantômes »(i.e. qui ne sont jamais produits par le Lexer) `OR_ELSE`, `AND_THEN` et `UNARY_MINUS` pour gérer les priorités de manière correcte (car `AND THEN` n'a pas la priorité de `THEN` comme `menhir` l'interprète de manière automatique).

Afin de décorer de manière simple les différents éléments de l'arbre de syntaxe abstraite avec la localisation dans le code (à savoir les identifiants, les expressions et les instructions), on utilise la règle d'ordre supérieur suivante :

parser.mly[72-74]

```
72 (* Rule modifier to decorate a
   ↪  production *)
73 %inline decorated(X):
74 | x = X; { decorate x ($startpos,
   ↪  $endpos) }
```

Listing 1.1 – Règle de décoration

parser.mly[77-80]

```
77 ident_desc:
78   | id = IDENT; { id }
79 ident:
80   | id = decorated(ident_desc); { id }
```

Listing 1.2 – Décoration des identifiants

Table 1 – Décoration des nœuds de l'arbre de syntaxe abstraite

On fait les simplifications suivantes :

- On peut lire ϵ lorsqu'on attend un `mode` pour ajouter automatiquement `in`.
- On développe les initialisations ou déclarations simultanées (`X, Y, Z : Integer := 0;` devient 3 déclarations séparées).
- Les fonctions et procédures ne sont pas distinguées dans l'`AST` (à part par la présence ou non de type de retour).
- On remplace le $-x$ unaire par $0 - x$.
- On remplace les `elsif` par des `if then else` imbriqués.

On déplore le fait de ne pas pouvoir séparer les produtions associées aux opérateurs binaires de celles associées aux expressions, car `menhir` refuse le mot-clé `%prec` (pour indiquer une priorité particulière) dans une règle déclarée avec `%inline` (pour garder les bonnes priorités).

On vérifie enfin que le fichier commence bien par les `with` et `use`, et que l'identifiant optionnel suivant un `end` d'une procédure est bien le même que celui de la procédure en question.

On traite le cas de `all` exactement comme l'accès à un champ d'un record nommé `all`, la différence se fera au typage.

Enfin, en ce qui concerne la gestion des erreurs, j'ai généré le fichier `parser.messages` mais ai renoncé à écrire 150 messages d'erreurs pour expliquer que `while` n'était pas un début de fichier valable, ni `with =...`

2 Arbre de syntaxe abstraite

2.1 Foncteur

Comme on considère deux arbres (avant et après le typage) décorés avec des informations différentes (la position dans le code source et les types, respectivement), et afin de bénéficier d'un maximum de modularité, on utilise un foncteur. Celui-ci prend un entrée un module contenant les types avec lesquels l'AST va être décoré, à l'aide du `type node`.

ast.ml[98-105]

```
98  (** The signature used to decorate an AST
    ↪  with *)
99  module type DECO =
100 sig
101   type ident_deco
102   type ident_decl_deco
103   type expr_deco
104   type stmt_deco
105 end
```

Listing 2.1 – Signature du module de décoration

ast.ml[83-88]

```
83  (** The type of a decorated type *)
84  type ('desc, 'deco) node =
85    {
86      desc : 'desc;
87      deco : 'deco;
88    }
```

Listing 2.2 – Type des éléments décorés

Table 2 – Structure du module `AST`

Cette manière de procéder permet de garder les mêmes noms de constructeurs pour les arbres en entrée et sortie du typage, de minimiser le copier-coller de code, de ne pas avoir à mettre des types en paramètre partout et de pouvoir changer la décoration en une ligne. On donne un exemple d'utilisation avec le type des expressions.

ast.ml[118-125]

```
118   type expr_desc =
119     | Econst    of const
120     | Eleft_val of left_val
121     | Ebinop    of expr * binop * expr
122     | Enot      of expr
123     | Enew      of ident_decl
124     | Eapp_func of ident * (expr list) (* list is non-empty *)
125   and expr = (expr_desc, D.expr_deco) node
```

Listing 3 – Type des expressions

2.2 Liste des types présents dans l'AST

On donne premièrement les types qui sont indépendants de la décoration (ou qui contiennent celle-ci en paramètre) qui permettent de n'avoir aucune conversion à faire.

`ident_desc` : le type des identifiants, à savoir une chaîne de caractères.

`binop` : simple type somme pour les différentes opérations possibles.

`const` : les différentes constantes possibles.

`typ_annot` : un type annoté par l'utilisateur.

`annotated` : un identifiant dont le type a été annoté par l'utilisateur.

Ci-dessous la liste des types présents dans le `module AST` :

`ident` : un identifiant représentant une variable, une procédure ou une fonction (dans une expression ou une instruction).

`ident_decl` : un identifiant dans une déclaration. Distinct du précédent car ne sera pas annoté par un type lors du typage.

`expr` : représente une expression. On a supprimé le − unaire et inclus l'appel à `character'val` dans les appels de fonction.

`left_val :` valeur gauche, qui est soit un identifiant (qui peut être un appel de fonction à 0 paramètres) soit l'accès à un champ d'un `record`.

`param :` paramètre de fonction, contient le mode (qui est `in` par défaut).

`stmt :` instructions. On a inclus les appels à `new_line` et `put` dans les appels de procédure, remplacé les listes d'instructions par un unique bloc, et remplacé les `elsif` par des `if then else` imbriqués.

`decl :` déclarations, les déclarations de plusieurs variables avec le même type ont été développées en plusieurs déclarations (on ne cherche pas l'efficacité, car on pourrait alors refaire le même calcul d'initialisation plusieurs fois).

`proc_func :` procédure ou fonction, distinguées par la présence d'un type de retour annoté (nom du type vide ou non).

2.3 Décoration

Dans le cas d'un arbre en entrée du typage (tout frais sortit du `Parser`), on décore les `ident`, `ident_decl`, `expr` et `stmt` avec une localisation, qui correspond à un couple (positions de début et de fin) de `Lexing.position`.

En sortie du typage, on ne décore plus que les `ident` et `expr` avec leur type, puisque le type d'une déclaration de variable ou d'une affectation n'a pas beaucoup de sens.

3 Représentation des types

3.1 Structure

Un type est représenté de la manière suivante :

ast.ml[278-298]

```
278  type level = int
279  type t_ident = (ident_desc, level) node
280
281  type typ_def =
282    | Tnull
283    | Tint
284    | Tchar
285    | Tbool
286    | Trecord of t_ident annotated list (* list is non-empty *)
287    | Taccess of t_ident
288    | Tproc_func of (typ * mode) list * typ (* typ = Tnull for a procedure *)
289  and typ =
290    {
291      t_ident : t_ident;
292      def     : typ_def;
293    }
294
295  let null = { t_ident = decorate "typenull"  0; def = Tnull }
296  let int  = { t_ident = decorate "integer"   0; def = Tint  }
297  let char = { t_ident = decorate "character" 0; def = Tchar }
298  let bool = { t_ident = decorate "boolean"   0; def = Tbool }
```

Listing 4 – Représentation des types du Mini Ada

L'annotation par des `level` permet de savoir à quel niveau l'identifiant en question fait référence. Cette structure permet de traiter les cas comme :

Exemple 1

```
1  procedure P is
2    tye R is record Foo :
     ↪  Integer; end record;
3    type T1 is access R;
4    type T2 is access R;
5    x : T1 := new R;
6    y : T2 := new R;
7  begin
8    if x = y --Type error
9    ...
10 end;
```

Listing 5.1 – Access nommés

Exemple 2

```
1  procedure P is
2    type R is record A: Character; end record;
3    procedure Q is
4      type Character is access R;
5      X : R;
6    begin
7      X.A := 'c';
8    end;
```

Listing 5.2 – Shadowing

Table 5 – Exemples de déclarations de type en Mini Ada

3.2 Possibilités supplémentaires

En plus des constructions du Mini Ada, on autorise la création de type `access` sur n'importe quel type (sauf `type A is access A`), ainsi que la copie de types (`type T is new Integer`), ce qui a pour effet de pouvoir transformer une valeur de type `Integer` en une valeur de `type T` sans que l'inverse soit possible.

4 Typage de l'arbre de syntaxe abstraite

4.1 Description de l'environnement

On utilise lors du typage un `module Env` qui encapsule presque toute la logique du typage, à savoir :

- connaître le type d'un identifiant ou la définition d'un type à un niveau de déclaration donné,
- détecter la présence de deux identifiants identiques à un même niveau,
- connaître les variables qui ne peuvent être modifiées,
- déterminer si `type T2` est un sous-type de `type T1` (i.e. avec `new` ou avec un `access` anonyme par exemple),
- vérifier que tous les types ont été déclarés avant un changement de niveau.

Ainsi, le `module Env` a la signature suivante :

typer.ml[32-64]

```
32 module Env : sig
33   type t
34
35   val new_env : t
36   (** Returns a new environment with builtins *)
37
38   val get_id_type : t -> ?lvl:level -> ident -> typ
39   (** Types an identifier (variable or procedure/function) at a given level
40       (default is current one) *)
41   val get_typ_ident_type : t -> ?lvl:level -> ident -> typ
42   (** Types a typ_ident, i.e. a name of a type *)
43   val get_typ_annot_type : t -> ?lvl:level -> typ_annot -> typ
44   (** Takes into account whether typ_annot is an access or not *)
45
46   val set_id_type : t -> ?lvl:level -> ?force:bool -> ident -> typ -> t
47   (** Sets the type of an identifier (variable or procedure/function), at the given
48       level, and force (defaults to false) prevents the env to check for multiple
49       definitions of the same variable (useful for for-loop-counter-variables) *)
50   val set_typ_ident_type : t -> ?force:bool -> ident -> typ -> t
51   val set_param : t -> ?force:bool -> param -> t
52   val set_undefined : t -> ?force:bool -> ident -> t
53   val set_subtypes : t -> typ -> typ -> t
54   val set_not_const : t -> ident_desc -> t
55
56   val ensure_all_def : t -> loc -> unit
57   val ensure_not_const : t -> ident -> unit
58   val ensure_subtype : t -> typ -> typ -> loc -> unit
59
60   val level : t -> level
61   val incr_level : t -> t
62   val deco_level : t -> ident_desc -> t_ident
63   (** Decorates the given identifier with the current level *)
64 end = struct
```

Listing 6 – Signature de l'environnement

Le type d'un environnement (qui est caché en-dehors du module `Env` pour plus de modularité) est le suivant :

typer.ml[65-83]

```
65  type constr =
66    | Ta (* type_annot *)
67    | Id (* variable or procedure/fonction *)
68  type obj = constr * typ
69
70  module Sset = Set.Make(String)
71  module Smap = Map.Make(String)
72  module Tset = Set.Make(struct type t = typ * typ
73      let compare = Pervasives.compare end)
74
75  type map = (obj * level) list Smap.t
76  type t =
77    {
78      objs     : map;    (* Maps an identifier to its type *)
79      consts   : Sset.t; (* Set of variables that shouldn't be modified *)
80      notdef   : Sset.t; (* Set of types that haven't been defined yet *)
81      subtypes : Tset.t; (* Contains (t1, t2) pairs where t2 is a subtype of t1 *)
82      current_level : level;
83    }
```

Listing 7 – Type de l'environnement

L'environnement est initialisé avec les `builtins`, comme les opérateurs (qui sont typés comme des fonctions), `Put`, `New_Line`, `Character'Val`, et les types primitifs, qui sont tous traités comme des déclarations de niveau 0. La procédure principale du programme est donc de niveau 1, et ses déclarations de niveau 2.

4.2 Gestion des sous-types

En ce qui concerne les sous-types, on traite cela d'une manière pour le moins dénuée de finesse : on représente les relations $T_2 \subseteq T_1$ (T_2 est un sous-type de T_1) par un ensemble de couples (T_1, T_2). Afin d'assurer la transitivité, on effectue les ajouts suivants lors de l'ajout de (T_1, T_2) :

$$\forall T_3,\ T_1 \subseteq T_3 \Rightarrow T_2 \subseteq T_3$$
$$\forall T_4,\ T_4 \subseteq T_2 \Rightarrow T_4 \subseteq T_1$$

4.3 *Shadowing everywhere*

On applique les règles suivantes (qui sont toutes tirées du fonctionnement de `gnat`) :

- la variable d'une boucle `for` *shadow* dès l'évaluation des bornes, et remplace tout objet déjà présent (même s'il était déclaré au même niveau),
- une variable déclarée *shadow* son initialisation (i.e. on interdit `X : Integer := X + 1;` même si `X` était déclaré un niveau plus haut),
- la déclaration d'un `type record` *shadow* ses champs, même si un `type R` d'un niveau précédent était bien formé,
- les paramètres d'une fonction ou procédure sont au même niveau que ses déclarations,
- une fonction ou une procédure est *shadow* par ses arguments.

4.4 Vérifications supplémentaires

Les vérifications restantes sont les valeurs gauches (traitées comme dans le sujet) et la présence de `return` dans le corps d'une fonction (avec *warning* en cas de code situé après un `return`), qui sont traitées à part sans grandes difficultés.

On vérifie de plus que la forme `(new R).X` n'apparaît jamais (ni en tant que valeur gauche, ni en tant que valeur droite) car soit il s'agit de le lecture d'un champ qui n'est pas initialisé (de manière certaine), soit il s'agit de la modification d'un champ qui n'est plus accessible dès la fin de l'instruction.

Enfin, quand on type la déclaration d'une variable `X`, on prend garde d'enlever un éventuel `X` de l'environnement marqué comme constant (car le nouveau `X` *shadow* l'ancien et a donc le droit d'être modifié). Pour les autres vérifications de non-modification, quand on type l'affection d'une valeur gauche, on remonte récursivement les champs, et l'on s'arrête lorsqu'on tombe sur une expression de type `e.f` avec `e` de type `access R` (puisqu'on ne modifie pas `e` mais la valeur sur laquelle elle pointe).

4.5 Cas du `.all`

Lors du typage de `e.f`, on teste premièrement si `f` est `all`. Dans ce cas, on vérifie que `e` est de type `access T` et on type `e.f` avec `type T`. Ceci permet d'avoir des valeurs gauches assez intéressantes :

Exemple 3

```
1   procedure P is
2     type R is record X : Integer; end record;
3     X : access R := new R;
4     function F return access R is
5     begin
6       return X;
7     end;
8     Y : R;
9   begin
10    F.all := Y;
11  end;
```

Listing 8 – Fonctions comme valeurs gauches (à peu de choses près)

5 Annexe : rôle de chaque fichier

Nom du fichier	Contenu
`ast.ml`	Types associé à l'arbre de syntaxe abstraite et au typage, fonctions d'affichage associées
`lexer.mll`	Lexer
`main.ml`	Parsing de la ligne de commande et compilation du fichier
`parser.mly`	Parser
`printer.ml`	Fonctions générales de pretty-printing
`typer.ml`	Typage d'un arbre de syntaxe abstraite et décoration
`utils.ml`	Fonctions usuelles utiles dans tout le projet

TABLE 9 – Liste des fichiers