Projet 1
ENS Lyon — L3IF
Analyse des types

Sébastien Briais

23 octobre 2007

Dans cette partie, vous allez devoir implémenter l'analyseur de types du langage `drei` en vous conformant aux règles de typage disponibles à l'URL suivante : `http://tinyurl.com/32xt96`

À rendre pour le 23 novembre minuit au plus tard.

1 Consignes générales

Votre compilateur devra pouvoir être invoqué à l'aide de la commande `dreic` dans un terminal. Il prendra en argument un paramètre indiquant la phase du compilateur testée suivi du nom de fichier `drei` à "compiler". Un exemple typique d'utilisation est :

```
./dreic -analyser Factorial.drei
```

Les noms des phases sont indiqués ci-après :

Nom de la phase	Nom de l'option
Analyse lexicale	`-scanner`
Analyse syntaxique	`-parser`
Construction de l'arbre	`-printer`
Analyse des types	`-analyser`
Génération de code	`-generator`

En outre, pour le rendu de votre projet,

- Vos sources devront être obligatoirement accompagnées d'un fichier `Makefile` permettant de construire votre projet. *Optionnellement*, vous pouvez aussi joindre un fichier `README` pour y reporter vos remarques éventuelles sur le fonctionnement ou la compilation de votre programme. Avec un peu de chance, il sera lu :p
- Vous devrez rendre vos sources sous forme d'une archive "tar gzippée". Le contenu de votre archive devra se décompresser dans un répertoire portant le nom de votre groupe (i.e. `B1`, `R3`, ...)
- L'archive devra m'être envoyée par mail. Le titre de votre mail sera de la forme "[PROJET] Etape 3"

2 Description de l'étape

Vous devez écrire un vérificateur de types, conforme aux règles de validité du typage de `drei`.

Contrairement aux phases précédentes où l'on se contentait de quitter à la première erreur rencontrée, votre vérificateur de types devra signaler *toutes* les erreurs présentes dans le programme. Les messages d'erreur devront être aussi explicites que possible et indiquer en particulier la position dans le fichier source de la construction fautive.

2.1 Structures de données

Une des premières choses à faire est de définir des structures de données pour représenter les concepts formels utilisés dans le typage.

2.1.1 Types

Les types de `drei` sont :
- Le type des classes `Class(a)` (où `a` est le nom de classe)
- Le type `Int`
- Le type `None` (ou `NoType`)

Il est pratique d'introduire un nouveau type `BadType` pour gérer les expressions mal typées afin de poursuivre la vérification de type.

Vous devrez donc définir un type pour représenter les types *internes* manipulés par l'analyseur de types. Par exemple :

```
type dtype = TClass of string
           | TInt
           | TNone
           | TBad
```

2.1.2 Symboles et portées

Un nom d'identificateur dans un programme `drei` peut désigner :
- un nom de classe
- un nom de champ de classe
- un nom de méthode
- une variable

On associe à chaque nom un *symbole* contenant toutes les informations le concernant nécessaires au compilateur : il y a donc quatre sortes différentes de symbole.

Les *portées* associent à un nom le symbole associé. Elles peuvent être implémentées de manière purement fonctionnelle avec des listes associatives ou des arbres (module `Map` de caml) ou bien de manière impérative par des tables de hachage (module `Hashtbl` de caml).

Pour illustrer mon propos, je vais utiliser la représentation sous forme de liste associative. Une portée de champ sera donc simplement une liste de type :

```
type field_scope = (string * field_symbol) list
```

Les quatres sortes de symbole sont décrits ci-après.

Symbole de classe À partir du symbole de classe, on doit pouvoir calculer facilement

- les parents de la classe ;
- la liste des champs de la classe (avec ceux des super classes) ainsi que leur symbole associé (portée de champ) ;
- la liste des méthodes de la classe (avec celles des super classes) ainsi que leur symbole associé (portée de méthodes).

Symbole de champ de classe Un symbole de champ de classe contient simplement le type du champ.

Symbole de méthode Un symbole de méthodes contient le type de chacun des paramètres formels de la méthode et le type de retour de la méthode.

Symbole de variable Un symbole de variable contient simplement le type de la variable.

en OCaml Une représentation possible des symboles en ocaml est la suivante :

```
type var_symbol =
    { var_type : dtype; }
and field_symbol =
    { field_type : dtype; }
and method_symbol =
    { params_type : dtype list;
      return_type : dtype; }
and class_symbol =
    { parents : string list;
      fields : field_scope;
      methods: method_scope; }
and var_scope = (string * var_symbol) list
and field_scope = (string * field_symbol) list
and method_scope = (string * method_symbol) list
```

Finalement, le type symbole sera simplement un type somme :

```
type symbol =
  | V of var_symbol
  | F of field_symbol
  | M of method_symbol
  | C of class_symbol
```

Comme pour le type `name` de la phase "construction de l'arbre", je vous conseille d'abstraire la création de symbole en définissant quatre fonctions de créations (une par type de symbole). Par exemple :

```
let new_var_symbol t =
  { var_type = t; }
```

Ceci étant motivé par le fait que nous étendrons le type des symboles par la suite pour pouvoir générer le code.

2.2 Modification du type `name` dans les AST

Durant l'analyse de types, nous allons modifier l'arbre pour attribuer les symboles idoines aux identificateurs.

Ainsi, nous allons modifier le type `name` (utilisé dans les AST) afin qu'une valeur de type `name` contienne non seulement le nom de l'identificateur mais aussi le symbole associé.

Comme les symboles ne sont pas connus avant de procéder à l'analyse de type, le symbole sera stocké dans une référence (ou un champ mutable).

En caml, vous pouvez par exemple représenter les noms ainsi :

```
type name = { name : string;
              mutable symbol : symbol option;
            }
```

Vous pourrez modifier la fonction de création des noms en :

```
let mk_name str = { name = str; symbol = None; }
```

Il est pratique de faire une fonction pour initialiser le symbole d'un nom à une valeur. Par exemple :

```
let set_symbol x s =
  assert (x.symbol = None);
  x.symbol <- Some s
```

La présence de l'`assert` est juste un parachute qui facilitera la phase de débogage ; ici, on vérifie simplement que l'on attribue au plus une fois un symbole à chaque nom.

2.3 Analyse des types

La présentation des règles de typages de `drei` suggère la définition de plusieurs fonctions pour mener cette tâche à bien. Voici la liste des fonctions à implémenter :

Bon typage des programmes Les règles de la forme $P \diamond$ pourront être implémentées par :

```
val analyse_program : program -> unit
```

Insertion dans les portées Les règles de la forme $\Gamma_c \vdash D \Rightarrow \Gamma'_c$ pourront être implémentées par :

```
val analyse_class : class_scope -> decl -> class_scope
```

À noter que comme la portée des classes est globale à l'analyse, vous pourrez préférer la représenter de manière impérative (table de hachage) dans une valeur globale. Ici, j'ai plutôt choisi la représentation fonctionnelle.

Vérifications des membres Les règles de la forme $\Gamma_c \vdash D \diamond$ pourront être implémentées par :

```
val analyse_classdef : class_scope -> decl -> unit
```

Bon typage des membres Les règles de la forme $\Gamma_c; b \vdash d \diamond$ pourront être implémentées par :

```
val analyse_member : class_scope -> name -> member -> unit
```

Bonne formation des types Les règles de la forme $\Gamma_c \vdash T \diamond$ pourront être implémentées par :

```
val analyse_type : class_scope -> dtype -> unit
```

Relation de sous-typage Les règles de la forme $\Gamma_c \vdash T <: T$ pourront être implémentées par :

```
val is_subtype : class_scope -> dtype -> dtype -> bool
```

Typage des expressions Les règles de la forme $\Gamma_c; \Gamma_v \vdash t : T$ pourront être implémentées par :

```
val analyse_expression : class_scope -> var_scope -> expression -> dtype
```

Typage des énoncés Les règles de la forme $\Gamma_c; \Gamma_v \vdash S \Rightarrow \Gamma'_v$ pourront être implémentées par :

```
val analyse_statement : class_scope -> var_scope -> statement -> var_scope
```

2.4 Signalement des erreurs

L'analyseur de types va vérifier que toutes les prémisses des règles d'inférences sont correctes (dérivable en des axiomes) sur le programme analysé. À partir de la règle «Program», l'analyseur procédera à une descente récursive à travers l'AST, guidé par les règles de typage, jusqu'à atteindre des axiomes. Si l'une ou l'autre règle ne peut vérifier ses prémisses, une erreur sera générée (mais n'oubliez pas que l'analyse doit continuer quand même).

Afin de pouvoir continuer l'analyse des types sur un programme mal typé, nous avons introduit le type `TBad` qui sera attribué aux *expression*s mal typées. Afin d'éviter de signaler de fausses erreurs de typage, le type `TBad` doit bien se comporter avec tout le monde. En particulier, le type `TBad` est sous-type de tous les types et il est également sur-type de tous les types (définissez la fonction `is_subtype` en conséquence !).

De même, pour éviter de propager des erreurs de type inutilement, il est important de remarquer que le type de certaines expressions ne dépend pas du type des sous-expressions la composant.

Ainsi, un noeud `Binop(o,e1,e2)` est toujours de type `TInt` que les expressions `e1`, `e2` soient bien typés ou non et quelle que soit la valeur de l'opérateur binaire `o`.

3 Extensions

Pour les groupes de type B, vous *pouvez* modifier les règles de typage de `drei` afin que les portées des variables correspondent aux portées utilisées dans la sémantique de `drei` et implémenter la version modifiée.

Par exemple, le programme suivant est mal typé avec les règles actuelles alors que moralement, il ne devrait pas.

```
class Vector {
  val x : Int;
  val y : Int;
  def scale(x:Int) : Vector = new Vector(x * this.x, x * this.y);
  def printLn() {
    printInt(this.x);
    printChar(44);
    printInt(this.y);
    printChar(10);
  }
}

{
  var x : Int = 1;
  {
    var x : Vector = new Vector(1,2);
    do x.printLn();
    do x.scale(2).printLn();
  }
  printInt(x);
  printChar(10);
}
```

En revanche, le programme suivant devrait toujours rester mal typé :

```
class Vector {
  val x : Int;
  val y : Int;
}

{
  var x : Int = 1;
  var x : Vector = new Vector(1,2);
}
```