Projet de Compilation

Petit Purescript

Gabriel Desfrene

3 janvier 2024

Organisation

Ce projet utilise dune[1] comme moteur de production. Le code du compilateur Petit Purescript est séparé en deux parties :

— Une bibliothèque PetitPureScript, dont les sources se trouvent dans le répertoire `lib`.

— Le compilateur ppurse, dont les sources sont localisées dans le répertoire `bin`.

Compilateur ppurse

Ce programme accepte les options suivantes :

`--help` affiche l'aide pour l'utilisation de ce compilateur.

`--parse-only` stoppe le compilateur après l'analyse syntaxique du fichier d'entrée.

`--type-only` stoppe le compilateur après l'analyse sémantique du fichier d'entrée.

`--permissive` autorise la définition de fonction avec des motifs de filtrages quelconques sur tous les arguments.

Le compilateur termine avec le code de sortie 0 lorsque aucune erreur n'est remarquée. Un code de sortie 1 signale une erreur lors de l'analyse du fichier d'entrée. Un code de sortie 2 marque une erreur interne du compilateur.

Bibliothèque PetitPureScript

Cette bibliothèque s'occupe d'implémenter et de définir toutes les structures nécessaires à l'implémentation du compilateur ppurse.

Analyse lexicale

L'analyse lexicale est opérée à l'aide de l'outil ocamllex[2] au sein du fichier `Lexer.mll`. Aucun problème n'a été rencontré lors de cette partie. Dans la suite de ce rapport ainsi qu'au sein du code de ce projet, les lexèmes renvoyés par l'analyseur lexical sont appelés *pretokens*. Ce sont des lexèmes étiquetés par une position.

L'ajout des lexèmes factices `{`, `}` et `;` est réalisé au sein du fichier `PostLexer.ml`. Ce fichier implémente l'algorithme proposé dans le sujet. Une fonction `last_pos` est implémenté afin d'obtenir la position du dernier lexème retourné.

1. `https://dune.build/`
2. `https://v2.ocaml.org/manual/lexyacc.html`

Analyse syntaxique

L'analyse syntaxique est opérée à l'aide de l'outil Menhir[3]. Le fichier `Tokens.mly` défini les lexèmes ainsi que leurs priorités. Le fichier `Parser.mly` s'occupe de construire l'arbre de syntaxe abstraite (AST) défini au sein du fichier `Ast.ml`.

La grammaire des types a été légèrement modifiée afin de supprimer des conflits *reduce-reduce*. De plus, puisque les classes de type et les instances, qui partagent la même grammaire, ont un sens différent des types, un nouveau symbole a été introduit : `cio_decl`[4]. On retrouve cette différence de type au sein de l'AST.

Mis à part ces quelques remarques, aucune difficulté majeure n'a été rencontrée.

Analyse sémantique

L'analyse sémantique est opérée au sein des fichiers du dossier `lib/typing`. Lors de cette analyse, l'arbre de syntaxe abstraite produit à l'étape précédente est transformé en un nouveau, défini au sein du fichier `TAst.ml`.

La déclaration des types est réalisée de la manière suivante :

```
type ttyp =
  | TQuantifiedVar of TQVar.t
  | TVar of { id : TVar.t; mutable def : ttyp option }
  | TSymbol of string * ttyp list
```

Les variables de types sont séparés en deux possibilités :

— Les variables de type universellement quantifiées : `TQuantifiedVar`. Ce sont les variables introduites lors des déclarations de fonctions, de classes de types, d'instances ou de symbole de types.

— Les variables de type unifiable : `TVar`. Ce sont des variables qui doivent être unifiée après leur introduction lors des appels de fonction ou l'application de constructeurs.

L'analyse sémantique et le regroupement des déclarations est effectuée au sein du fichier `Typing.ml`. L'analyse des expressions est effectuée dans le fichier `ExpressionTyping.ml`. Les motifs de filtrages sont analysés et typés dans le fichier `PatternTyping.ml`. Le fichier `CommonTyping.ml` regroupe des déclarations utilisées par plusieurs fichiers.

On remarque qu'en Purescript le code de la figure 1 est correct. Alors que la variable `x` est de type `List Int a` lors de l'appel de la fonction `log` à la ligne 16, ce code est accepté. En effet, le type de la variable `x` est résolu à la ligne suivante, `x` est de type `List Int Boolean`. Ce code illustre que la résolution des instances est réalisée après l'analyse de l'expression des fonctions. Cette résolution est effectuée au sein du fichier `ResolveInstance.ml`.

Le filtrage est compilé en un *case* sur tous les constructeurs possibles du motif au sein du fichier `CompileCase.ml`. Enfin, le fichier `TypingError.ml` contient le code nécessaire à la production des erreurs rencontrée lors de l'analyse sémantique.

3. `http://gallium.inria.fr/~fpottier/menhir`
4. *Class or Instance declaration.*

```purescript
module Main where

import Prelude
import Effect
import Effect.Console

data List a b = Nil a | Cons b (List a b)

instance (Show a, Show b) => Show (List a b) where
    show (Nil x) = "(Nil " <> show x <> ")"
    show (Cons hd tl) = "(Cons " <> show hd <> " " <> show tl <> ")"

main :: Effect Unit
main = let x = Nil 1 in -- x :: List Int a, with a unknown at this point
  do
    log (show x)
    (let y = Cons true x in log (show y)) -- Here we find that a = Boolean, so x :: List Int Boolean
    log (show x)
```

FIGURE 1 – Exemple Purescript.

Compilation :

Les `Effect a` sont des clôtures retournant une valeur de type `a`. Division par 0 et modulo par 0 sont définis comme en PureScript par 0. L'égalité de type `Unit` est toujours vraie, à l'inverse de la différence.