Petit Purescript

Octave Mortain, Théo Goix

1 Lexing

1.1 Fichiers

`lexer.mll`: Pas grand chose de particulier. Les tokens émis ici sont récupérés par `indentlexer.ml`.

`indentlexer.ml`: La fonction `token` qui est envoyée au parser utilise une file pour les lexèmes qu'elle doit envoyer. Lorsque cette file est vide, elle appelle `find_token` qui prend un token donné par `Lexer.token` et l'ajoute à la file, ainsi que d'autres tokens au besoin, suivant l'algorithme décrit dans le sujet.

Décommenter l'avant-dernière ligne de `Indentlexer.token` permet d'afficher tous les tokens émis.

1.2 Problème

Les chaînes de caractères interagissent bizarrement avec l'indentation significative, le code suivant donne par exemple une erreur de syntaxe:

```
main::String->Int
main str = case str of "hello" -> 42
                       x -> 1
```

alors que celui là compile normalement

```
main::String->Int
main str = case str of ("hello") -> 42
                       x -> 1
```

2 Parsing

2.1 Fichiers

`parser.mly`: Le parser reprend à la lettre la grammaire hors contexte donnée dans le sujet, a quelques exceptions:

- Certaines écritures ont été développées, en particulier dès qu'il y avait une liste avec plus d'un seul token par élément de la liste.

- La déclaration du type des fonctions a été écrite sous une forme récursive plutôt qu'avec deux listes pour éviter les conflits shift-reduce.
- Un `type` (renommé en `btype` pour éviter le conflit avec le mot-clé Caml) ne peut pas s'écrire comme un `ntype`, puisque les `ntype` représentent les instances et non les types construits.

`ast.ml:` L'AST a une forme très proche de la grammaire hors-contexte. Tous les types sauf `binop`, `constant` et `bind` sont décorés avec leur localisation (sous forme d'un couple début-fin) pour placer les erreurs.

Le case prend une liste d'entrées, et chaque branche à une liste de patterns, même si ces listes ont un seul élément dans la syntaxe, pour simplifier la transformation des différentes équations de fonctions en une seule avec un case (pas encore implémentée).

2.2 Éléments manquants

Les classes et les instances ne sont pour l'instant pas implémentées. Le parser renvoie simplement des unit à ces endroits, et les constructeurs `DefClass` et `DefInstance` n'ont pas d'arguments. La liste d'instance dans la déclaration du type d'une fonction est une liste de units.

3 Typage

3.1 Fichiers

`grouping.ml:` Le fichier est initialement une liste de déclarations `Ast.decl`. Il est ensuite transformé en une liste de déclarations groupées `Ast.gdecl` par la fonction `group_fun`. Cette dernière se charge de vérifier que les équations qui définissent une fonction se trouvent bien immédiatement après la déclaration de son type, et regroupe alors le type et les différentes équations de la fonction en une seule `gdecl`.

`group_fun` sera aussi chargé de regrouper les définitions de classes avec leurs fonctions membres et les déclarations d'instances avec leurs équations.

`typing.ml:` Basé sur le TD sur l'algorithme W. Les définitions sont mises dans `typing_def.ml` pour pouvoir les inclure dans `ast.ml`.

Les types construits `Tdata` sont représentés par leur nom et leur liste de paramètres. Les types de base (`Tint`, `Tbool`, `Tstring` et `Tunit`) sont codés de manière intrinsèque pour l'instant, plutôt qu'en utilisant un `Tdata`, ce qui changera peut-être. Les types produits `Tproduct` sont toujours là mais ne servent à rien. Les fonctions (et les constructeurs) sont typés à l'aide de `Tarrow` qui prend une liste de types en entrée plutôt qu'un seul, puisqu'il n'y a dans tout les cas pas d'évaluation partielle.

L'algorithme W est implémenté dans la fonction `w_expr`, qui transforme les expressions `Ast.expr` en expressions typées `Ast.texpr`, à l'aide de beaucoup de tests d'unifications plus ou moins intéressants.

Pour le typage du `case`, on ne fait actuellement que vérifier que les types des patterns sont cohérents avec ce sur quoi on match, et que toutes les branches renvoient bien le même type. On utilise pour cela la fonction `type_pattern` qui renvoie le type d'un pattern en ajoutant à l'environnement des nouvelles variables de types pour les variables rencontrées dans le pattern. Chaque branche créée donc un nouvel environnement pour typer l'expression qui en résulte.

Le typage est commencé par la fonction `type_file` qui ajoute les fonctions prédéfinies à l'environnement et appelle `type_decl` qui parcourt la liste des déclarations groupées. Elle ajoute les types construis à `datas` et leur constructeurs à `env`, et elle type les fonctions pour vérifier que le type de retour est bien celui attendu et les ajoute à l'environnement.

3.2 Éléments manquants et problèmes

- Le typage ne fait évidemment rien pour les classes et les instances.
- L'algorithme F n'est pas encore implémenté, on ne vérifie donc pas si le pattern-matching est exhaustif et l'AST typé du `case` n'est pas prêt pour la production de code.
- Lorsqu'une expression est simplement une variable, on ne vérifie pas qu'il s'agisse bien d'une variable et pas d'une fonction, par exemple:

  ```
  f::Int->Int
  f x = x + 1

  main::Int->Effect Unit
  main x = f
  ```

  Renvoie une erreur de typage (c'est bien) mais simplement parce que `main` renvoie un `Int->Int` au lieu d'un `Effect Unit`.
- Au contraire, les variables globales ne sont pas bien gérées puisque par exemple;

  ```
  n::Int
  n = 42
  ```

  définit une variable `n` dans l'environnement qui a le type `Tarrow([], Tint)` au lieu de `Tint`.
- Le typage ignore les paterns qui ne sont pas des variables dans les arguments des déclarations de fonctions (Il faut faire la transformation en case des équations de fonction).