

Rapport:

Compilateur Petit Purescript

Matthieu Boyer, Antoine Groudiev

Ce document vise à détailler les différents choix techniques que nous avons pris lors de la réalisation de notre compilateur, et à guider le lecteur dans la compréhension de notre code source.

1 Exécution

- La commande `make` crée le compilateur, nommé `ppurs`, à la racine du projet.
- La commande `make test` crée et exécute le compilateur sur le fichier par défaut `test.purs`.
- La commande `make testc` crée et exécute le compilateur sur le fichier par défaut `test.purs`, et lance l'exécution avec `gcc`.
- La commande `make testd` est similaire à `make testd` mais active le mode `dbg`, qui *pretty print* l'AST.
- La commande `make tests1` exécute le compilateur sur les jeux de tests de la partie 1 (respectivement 2 et 3).
- La commande `make clean` supprime les fichiers créés par `make`

Une fois créé, le compilateur peut être exécuté en lançant une commande de la forme :

```
./ppurs [options] file.purs
```

Les différentes options disponibles sont listées par la commande `./ppurs -help`, parmi lesquelles les options demandées d'analyse syntaxique ou typage seuls, et un mode de débogage qui affiche l'AST.

2 Choix techniques

Le langage utilisé est OCaml. L'analyse lexicale est faite à l'aide de l'outil `ocamllex`, et l'analyse syntaxique avec Menhir.

2.1 Fichiers généralistes

`ppurs.ml` est le fichier principal du projet. Il traite les arguments de la ligne de commande, appelle successivement les différents analyseurs, et gère les éventuelles erreurs advenant au cours de la compilation.

Le fichier `utility.ml` contient des fonctions utilitaires servant notamment à extraire l'adresse ou le type d'une expression, d'un atome, etc.

2.2 Analyse lexicale

`lexer.mll` contient un analyseur lexical traditionnel, dans le sens où il ne gère pas l'indentation significative.

À la place, `indenter.ml` contient un second analyseur lexical. Son rôle est de traiter correctement l'indentation significative propre à PureScript, en générant à la volée les tokens

factices `SEMICOLON (;)`, `LBRACK ({)` et `RBRACK (})` parmi le flux de lexèmes généré par `lexer.mll`. La fonction principale de ce fichier, `traiter_lexeme`, reproduit le comportement attendu d'un analyseur lexical. Elle a ainsi sensiblement le même type que `Lexer.token`, à l'exception du booléen précisant le type (fort ou faible) voulu.

2.3 Analyse syntaxique

`parser.mly` contient l'analyseur syntaxique. Son comportement est standard et suit majoritairement la grammaire PureScript telle que détaillée dans le sujet. Une exception notable est la règle `tdecl`, qui nous a posé des difficultés à parser. La technique adoptée a été de lire la liste à la main, à l'aide d'une règle `type_list` qui implémente manuellement et plus finement la construction `list` de Menhir. Pour pouvoir localiser précisément les erreurs de typage, les types `expr`, `atom`, et `patarg` sont transformés par le parser en `loc_expr`, `loc_atom`, et `loc_patarg`, qui sont décorés des informations de localisation.

L'arbre de syntaxe abstraite généré par l'analyse syntaxique est défini dans `ast.ml`, comme les deux autres AST (respectivement après le typage et l'allocation des variables.)

2.4 Typage

`typing.ml` contient l'analyseur sémantique. L'algorithme employé est le suivant : on type une à une les déclarations du fichier, en maintenant dans des tables de hachage, qui contiennent les définitions de fonctions, de types, de classes et d'instances. On vérifie à chaque étape que la liste des définitions à l'intérieur d'une déclaration (`fdecl`, `clas`, `data`, `instance` dans le type `decl`) est bien regroupée par nom de fonction, et que celle-ci ne déborde pas de la déclaration. Dans chaque définition (`defn` dans le type `decl`), on vérifie récursivement le type de chaque morceau selon la construction de l'arbre de syntaxe.

A été fait le choix de séparer les différentes erreurs de typage en de nombreuses exceptions, comme le traduit les premières lignes du fichier. Cette séparation est en fait peu utile en pratique pour un compilateur de faible ampleur, mais nous semblait plus propre et plus *scalable*.

2.5 Production de code

Le fichier `compile.ml` est séparé en deux parties : l'allocation sur la pile, et la production du code assembleur.

2.5.1 Allocation

Le schéma de compilation et particulièrement d'allocation est similaire à celui décrit dans le sujet. Les valeurs de retour des fonctions sont cependant stockées sur la pile, et non pas dans le registre `%rax`.

C'est également à cette étape que les éventuels *pattern matching* faisant intervenir plusieurs déclarations de fonctions sont convertis en un `case`, dans la fonction `alloc`. (Ceci aurait dû plus logiquement être fait pendant le typage.)

2.5.2 Compilation

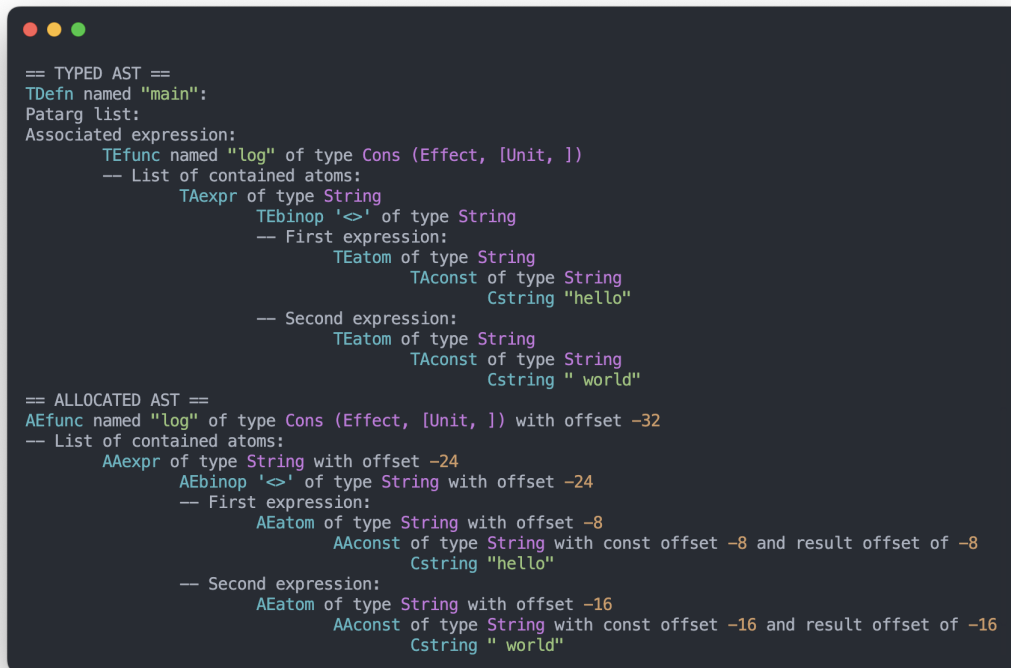
Au fichier `x86_64.ml` a été rajouté deux constructions : l'instruction `enter` de `x86-64`, pour une gestion de la pile plus aisée, et une instruction `comment`, permettant d'insérer des commentaires dans le code, très utile pour le débogage.

Certaines fonctions ont naturellement été écrites directement en assembleur. C'est le cas des fonctions d'affichage d'entiers, de booléens, de division et modulo, de concaténation de chaînes,

mais aussi des fonctions prédéfinies de PureScript, telles `log`, `not` ou `pure`. À noter que ces fonctions ne sont ajoutées au code que si potentiellement utilisées.

2.6 *Pretty printer*

Le fichier `pretty.ml` contient un *pretty printer* pour les arbres après le typage et après l'allocation. Il affiche, en couleur, la hiérarchie des constructions et les "adresses" des variables et autres données allouées, et s'est avéré très utile lors des nombreuses phases de débogage. Il peut notamment être affiché en activant l'option `--dbg`.



```

== TYPED AST ==
TDefn named "log":
Patarg list:
Associated expression:
  TFunc named "log" of type Cons (Effect, [Unit, ])
  -- List of contained atoms:
    TExpr of type String
    TEbinop '<>' of type String
    -- First expression:
      TEatom of type String
      TAcnst of type String
      Cstring "hello"
    -- Second expression:
      TEatom of type String
      TAcnst of type String
      Cstring " world"

== ALLOCATED AST ==
AFunc named "log" of type Cons (Effect, [Unit, ]) with offset -32
-- List of contained atoms:
  AExpr of type String with offset -24
  AEbinop '<>' of type String with offset -24
  -- First expression:
    AEatom of type String with offset -8
    AAconst of type String with const offset -8 and result offset of -8
    Cstring "hello"
  -- Second expression:
    AEatom of type String with offset -16
    AAconst of type String with const offset -16 and result offset of -16
    Cstring " world"

```

FIGURE 1 – Exemple du *pretty printer* sur `concat.purs`

3 Limitations connues

Notre compilateur ne passe pas tous les tests fournis. Ces limitations sont dues aux difficultés rencontrées lors du typage des instances et du *pattern matching*.

Par ailleurs, la gestion de la pile laisse à désirer. Dans certains cas (`do`, `if`), il aurait été préférable d'allouer les expressions successivement calculées aux mêmes emplacements sur la pile plutôt que de continuer à allouer en haut de celle-ci. Nous avons néanmoins adopté cette dernière option pour simplifier la gestion des compteurs et ainsi faciliter le débogage.