Projet de Compilation

Petit Purescript

Gabriel Desfrene

21 janvier 2024

Le compilateur `ppurse` ici présenté, traite et supporte la totalité du sujet. De plus, ce compilateur supporte quelques extensions :

— Le typage des fonctions peut être rendu plus permissif, en autorisant le filtrage sur tous les arguments à l'aide de l'option `--permissive`.

— Les expressions du type `Effect a` sont compilée comme des clôtures afin d'imiter au plus le comportement du compilateur PureScript.

— Les constantes entières négatives sont autorisées dans les motifs de filtrage.

Analyse lexicale et syntaxique

Aucune difficulté majeure a été rencontrée dans cette partie. La grammaire a légèrement été modifiée afin de supprimer des conflits *réduction-réduction*. Un nouveau symbole a été introduit, `cio_decl` [1] afin que les références à des instances ou des classes de type soient différentiée des références à des symboles de types. Enfin, la règle ⟨*patarg*⟩ a été modifié afin d'accepter une constante entière négative.

Analyse sémantique, compilation des filtrages et résolution des instances

La déclaration des types est réalisée de la manière suivante :

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

Les variables de types sont alors séparés en deux possibilités :

— Les variables de type universellement quantifié : `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.

Lors de la vérification de l'exhaustivité d'un filtrage, ce dernier est transformé selon le type de l'expression filtrée :

— Les filtrages sur des expressions booléennes sont transformée en des branchements conditionnels.

1. *Class or Instance declaration.*

— Les filtrages sur les entiers sont distingués des filtrages sur les chaînes de caractères pour faciliter leur compilation dans la suite. Ils sont donc représentés sous la forme de dictionnaire ayant pour clef des entiers ou des chaînes de caractères. Cette solution n'est pas la plus plaisante, car presque redondante, mais elle permet d'assurer l'uniformité du type des constantes filtrant l'expression.

— Les filtrages sur des constructeurs de type sont transcrits par un dictionnaire associant chaque continuation à un constructeur.

Les instances de types sont résolue après l'analyse sémantique de la fonction dans laquelle la résolution prend lieu. On procède de cette manière afin d'accepter, comme PureScript, le code du test `lazy-inst.purs`.

Le fichier `TypedAst.ml` décrit la représentation d'un programme après l'analyse sémantique.

Simplification du programme

Une passe de simplification est effectuée entre le typage et la suite de la compilation. Cette étape permet de propager les constantes et de procéder à de procéder à de nombreuses substitutions sur les variables introduites lors de la compilation des filtrages et des fonctions. Lors de cette étape, les filtrages sur les constantes (entiers ou chaînes de caractères) sont transformés en des arbres binaires de recherche. De plus, certaines expressions constantes sont, en partie, évaluées par le compilateur. Enfin, lors de cette étape, les opérateurs binaires sont classés dans plusieurs catégories selon le type de leurs arguments et de leur résultat.

Le fichier `SympAst.ml` décrit la représentation d'un programme après la passe de simplification.

Allocation des variables locales

La passe d'allocation des variables locales calcule la taille du tableau d'activation de chaque fonction. De plus, lors de cette passe, les variables locales sont positionnées dans la mémoire (par rapport à `%rbp`). C'est également lors de cette étape que les blocs `do` correspondant à la construction d'une clôture à partir d'autres clôtures sont traités. On calcule alors l'ensemble des variables libres et des instances locales (celles données en argument à une fonction) utilisées au sein des blocs `do`. Puis, le code correspondant au bloc `do` est placé dans une nouvelle fonction accédant à ses arguments via la clôture toujours placée dans registre `%r12`. On fera attention à sauvegarder cette clôture lors de l'appel à une clôture dans une clôture.

Le fichier `AllocAst.ml` décrit la représentation d'un programme après la passe d'allocation des variables et le traitement des clôtures introduites par les blocs `do`.

Compilation vers x86_64

On respecte le schéma de compilation proposé dans le sujet. On impose également que lors d'un appel à une clôture, cette dernière est placée dans le registre `%r12`. Lors de la construction de nouvelles instances par un schéma de classe de type, on se sert de l'instance construite par le schéma afin de stocker les instances dont elle nécessite. Celles-ci sont placées à la suite des pointeurs vers le code des fonctions déclarées dans la classe de type. Enfin, afin que les fonctions implémentées dans un schéma d'instance de type aient accès aux instances dont elles font usage, on place toujours l'instance dans laquelle elles ont été déclarée sur le tas, avant tous les arguments [2]

Les fonctions de la bibliothèque standard du C utilisées sont encapsulés dans des fonctions respectant la convention d'appel proposée dans le sujet. Ces fonctions ainsi que les fonctions prédéfinies sont toujours ajoutées au code assembleur produit. Quelques remarques sur ces fonctions :

2. Comme une fonction classique.

— La fonction `pure` est compilée comme la fonction retournant une clôture de la fonction identité.

— La fonction `log` est compilée comme la fonction retournant une clôture de la fonction affichant du texte à sur la console.

— La fonction `mod` n'a pas été *inline* au sein du code produit. En effet, cette fonction émule le comportement de la fonction `mod` de PureScript, et nécessite donc plus qu'une poignée d'instruction.

— Une fonction `div` est introduite afin de simuler, de la même manière que `mod`, la division de PureScript.

On remarquera d'ailleurs qu'en PureScript :

$$\forall x.\ \texttt{mod}\,(x, 0) = 0 \text{ et } \texttt{div}\,(x, 0) = 0$$

Remarques

Ce projet fût très intéressant et pleins de rebondissement, en particulier la partie de production de code. Quelques améliorations pourraient cependant être ajoutés au code :

— Une bonne refactorisation du code, en particulier lors de l'analyse sémantique.

— Des meilleurs messages d'erreurs lors d'un filtrage non exhaustif, en énumérant les cas non traités.

— Réaliser une allocation des registres afin de grandement diminuer le nombre d'opérations sur la pile

— Réaliser une meilleure sélection des instructions.

Quelques tests ont été ajoutés afin de vérifier le comportement du code produit par le compilateur. Parmi ceux-ci, on notera :

— `pi-with-bbp.purs` : Ce test approxime π à l'aide de la formule de Bailey-Borwein-Plouffe[3]

— `modulo.purs` : Ce test (généré) vérifie le comportement de la fonction `mod`

— `division.purs` : Même qu'au-dessus pour la division.

— `lazy-inst.purs` : Ce test permet de vérifier la résolution des instances.

— `lazy-bool-op.purs` : Ce test permet de vérifier que les opérations booléennes sont bien paresseuses à l'aide d'une boucle infinie.

— `fibonacci.purs` : Ce test calcule quelques termes de la suite de Fibonacci à l'aide d'une liste.

— `evil_effects.purs` : Ce test vérifie que la compilation des effets est similaire à celle dans PureScript.

— `effect_and_inst.purs` : Ce test vérifie la compilation des schémas d'instances.

— `case4.purs` : Ce test vérifie la compilation du filtrage sur les constantes.

3. Voir : `https://fr.wikipedia.org/wiki/Formule_BBP`