

Premier rapport du compilateur Petit Koka

Aghilas Y. Boussaa <aghilas.boussaa@ens.fr>

19 janvier 2025

1 Le compilateur

Le compilateur passe tous les tests d’analyse syntaxique, de typage et de production de code. On a ajouté la possibilité de définir des types algébriques de données et du filtrage par motif sur eux¹.

1.1 Analyse lexicale

Pour l’analyse lexicale, on implémente l’algorithme donné dans l’énoncé, en utilisant une file permettant au *lexer* de renvoyer plusieurs lexèmes en une fois.

Pour l’indentation, on a fait le choix de ne prendre en compte que les espaces pour éviter de choisir une valeur arbitraire pour les tabulations. En particulier, deux espaces suivies d’une tabulation puis d’une autre espace compte comme une indentation de deux.

On a fait le choix de ne plus considérer `>` comme une fin de continuation, pour qu’un lexème `{` soit émis à la fin de la première ligne du code de la figure 1.

```
type option<a>
  None
  Some { x : a }
```

FIGURE 1 – `>` ne peut être une fin de continuation

1.2 Analyse syntaxique

Pour l’analyse syntaxique, on modifie la grammaire donnée pour la rendre LR(1) (en particulier, on n’utilise les directives d’associativité que pour les opérateurs binaires). On a fait ce choix sur une recommandation de François Pottier donnée après un “rapport de bug”². Certaines constructions syntaxiques ont nécessité une attention particulière :

1. le filtrage par motif ne fonctionne que sur les types définis dans le programme, en particulier, il ne fonctionne pas sur les listes de la librairie standards, les entiers, ou les chaînes de caractères, par manque de temps

2. suite à l’utilisation de directives, `menhir` a renvoyé une erreur : `Conflict (unexplicable) [...]` Please send your grammar to Menhir’s developers, il s’agit, d’après la réponse reçu par mail, d’un problème connu que l’on peut éviter en donnant une grammaire LR(1) (de ce que j’ai compris en tout cas)

- les opérateurs binaires doivent traiter à part l’expression la plus à droite car elle peut être de la forme `return 1`, alors que `return 1 + 2` est interprété comme `return (1 + 2)`,
- pour traiter le problème du *sinon pendouillant*, on a une règle à part générant des expressions ne pouvant pas être suivie par des `else`,
- pour traiter l’associativité à gauche des `fn` infixes, on remarque qu’un atome `a`, à droite, soit un atome, soit un atome suivi du mot clé `fn` puis d’une expression se terminant par un bloc (une expression se terminant par un atome ou un bloc); on crée donc une règle pour les expressions se terminant par un bloc, ce qui nous permet de traiter ces deux cas.

1.3 Typage

Pour le typage, on implémente l’algorithme J d’inférence (comme en TD) de type sans la généralisation ni l’instantiation, auquel on ajoute les effets. On a de l’instantiation pour les constructeurs de types. Pour les fonctions de bibliothèques standards qui ont du polymorphisme d’effet, on vérifie que les types ont la bonne forme sans se soucier des effets.

```
fun g(f : () -> <div> ())
  ()

fun f()
  g(f)
  println(42)
```

FIGURE 2 – un exemple nécessitant une variable d’effet

Le code de la figure 2 montre que l’on ne peut se contenter de gérer les effets de manière naïve avec seulement des ensembles, car il faut choisir un type pour `f` lorsqu’on analyse `f`, et le compilateur doit rejeter ce programme puisque `f` a l’effet `console` à cause de la dernière ligne. Ce type d’indétermination n’arrive que pour les fonctions récursives, et seulement pour l’effet `console` (dans Petit Koka, en tout cas), comme une fonction récursive a toujours l’effet `div`. On distingue, donc, deux types d’ensemble d’effets, ceux qui incluent les effets indéterminés de la fonction qu’on est en train d’analyser (constructeur `HasRec`), et les autres (constructeur `NoRec`). On ajoute, alors, au contexte un booléen optionnel (`rec_has_console`) indiquant une éventuelle contrainte sur l’effet `console` issue de l’unification.

Le code de la figure 3 montre que le typage ne peut pas être que linéaire, car à la deuxième ligne, le typeur ne sait pas encore que `f()` est affichable. Pour résoudre ce problème, le typeur peut ajouter des contraintes qui ne seront testées qu’après la première passe de typage (en même temps que l’explicitation des fermetures), aussi bien pour l’affichage que pour les opérations binaires polymorphes. On pourrait se contenter

```

fun f()
  println(f())
1

```

FIGURE 3 – les GADT à la rescousse

de garder des listes globales de variables de types pour chaque “classe de type” et les vérifier à la fin.

Cependant, les informations apportées par ces tests peuvent alléger les phases suivantes si on les garde : on peut, par exemple, remplacer les appels à `println` par des appels à de simples fonctions monomorphes. Pour faire cela, on peut créer un constructeur `CheckConstraint of typed_expr * typed_expr -> typed_expr`, où la fonction permet, par exemple, de transformer l’expression `e` donnée en `println_t(e)` si le type `t` de `e` peut s’afficher.

En revanche, cela ne marche pas pour les opérateurs binaires tels que `++` pour lesquels on voudrait une fonction prenant deux expressions, ou pour de futures extensions. Les GADT nous permettent, enfin, d’avoir un constructeur `CheckConstraint : 'b * ('b -> typed_expr) -> typed_desc`, sans que la variable de type n’apparaisse dans le type des expressions.

1.4 Production de code

Après le typage, on explicite les fermetures et les positions des variables sur la pile. Pour la génération de code on suit l’énoncé à la différence que lorsqu’on appelle une fonction, un pointeur vers la fermeture est placée dans le registre `%r12`. Dès qu’on compile une expression, son résultat se trouve en `%rax`.

Le filtrage par motif est compilé de manière naïve. Tous les constructeurs (même ceux sans argument) sont *boxed*.