Rapport de la partie 1 du projet de compilation

Ryan Lahfa, Julien Marquet

Table des matières

Choix techniques

Le projet a été réalisé en utilisant Rust stable 1.48.0, il s'agit d'une collection de crates Rust qui implémente des parties du projet, les crates relatives à la partie 1 sont :

— `automata` : librairie d'automates
— `parsergen` : librairie de générateurs
— `parser` : l'analyseur syntaxique et sémantique (partie 1)

Quelques éléments d'instructions pour tester le projet

Une façon simple de faire tourner le projet consiste à se munir d'une toolchain stable de Rust pas trop vieille et d'exécuter `cargo build`, cela construira le parser qui sera disponible dans `target/debug/parser`.

Ainsi, une façon simple de reproduire les tests fournis est d'aller dans `contrib/compil` puis d'exécuter `./test.sh -2 ../../target/debug/parser`.

Architecture

Pour l'analyse lexicale et syntaxique, nous avons décidé de créer nos propres outils. Nous aurions pu utiliser des bibliothèques, mais il nous semblait plus intéressant de faire le travail nous même.

Ces outils sont assez généraux, et nous nous en sommes servis pour implémenter l'analyse syntaxique de Petit Julia (ainsi que l'analyse syntaxique d'un composant du projet de Systèmes Numériques).

Nous avons développé un système de macros procédurales, avec une macro pour le *lexer* et une autre pour le *parser*.

Analyse lexicale

Pour l'analyse lexicale, nous nous sommes fondés sur le TD 4. Les lexeurs prennent en entrée une chaîne de caractères et renvoient un itérateur sur les lexèmes. Cet itérateur donne : * le type de lexème; * la position du lexème; * le texte du lexème.

Techniquement, l'itérateur renvoie des `Result<...>`, ce qui nous permet de lexer paresseusement : on ne signale une erreur que lorsque l'on est effectivement en train de lire du texte erroné.

La macro procédurale que nous avons écrite n'est en fait que du sucre syntaxique, nous ne générons pas le lexeur au moment de la compilation, car la génération est assez rapide pour être faite à la volée.

Point-virgule automatique, rejet de "else if"

Le lexeur que nous avons écrit pour Petit Julia travaille en fait avec une classe plus grande de lexèmes : nous détectons aussi les retours à la ligne.

Cela permet d'inclure une passe supplémentaire entre l'analyse syntaxique et l'analyse lexicale qui :

— réalise l'insertion de point-virgules;
— rejette la séquence "else if".

Ces deux fonctinnalités ne nous paraissaient pas réalisables simplement avec un système *lexer-parser* classique.

Techniquement, nous avons un itérateur sur les lexèmes donnés par l'analyseur lexical qui garde en mémoire un état et qui implémente la logique nécessaire à l'insertion de point-virgule et au rejet de la séquence "else if".

Analyse syntaxique

L'analyseur syntaxique a aussi demandé l'écriture d'une macro procédurale. Mais, si nous avons écrit l'analyse lexicale sans trop s'encombrer, l'analyse syntaxique n'a pas été de tout repos.

Nous avons implémenté un parseur LR(1). La difficulté n'a pas été la partie théorique, mais clairement la partie technique. En effet :

Table d'automate

La table de l'automate que nous avons écrit pour Petit Julia est **grande** (dans les 800 états) et est particulièrement longue à générer. Nous avons réussi à faire descendre le temps de génération à moins de deux secondes en demandant au compilateur d'optimiser au maximum le code du générateur. [1]

Sans le optimisations, la génération prenait presque une minute !

Le borrow checker de Rust

Comme nous avons utilisé Rust, nous avons eu le plaisir de nous battre avec le *borrow checker*. Cela est dû à la représentation que nous avons adoptée pour notre arbre de syntaxe (AST), qui demande de garder une référence au fichier source pour des raisons de localisation d'erreurs.

Temps de compilation

Le temps de compilation est en l'état assez long. Cela est dû à la façon dont la macro procédurale fonctionne : elle revient à coder en dur un tableau à 800 entrées qui demande au compilateur de réaliser **beaucoup** d'inférence de types, et ajoute une dizaine de secondes à la compilation.

Nous cherchons à régler ce problème pour la version finale du projet (une option serait de sérialiser la table de l'automate dans un objet qui demande moins de travail au compilateur).

Localisation des erreurs

Afin d'avoir des messages d'erreur propres, nous avons introduit un système de localisation des fragments de source. Pour cela, nous avons deux types de

1. Ce qui peut paraître bizarre dans le `Cargo.toml`. Mais, à terme, on isolera l'optimisation de code proprement.

structures : `Loc`, qui représente la position d'un caractère dans la source, et `Span`, qui représente la position d'un élément syntaxique dans la source (un `Span` est construit à partir de deux `Loc`s).

Théoriquement, nous aurions pu vouloir avoir plusieurs fichiers source différents. Afin de gérer correctement la localisation des erreurs dans cette éventualité, nous avons done eu besoin de retenir dans `Loc` le nom du fichier que nous utilisons. Afin de ne pas copier inutilement des données, nous avons choisi de garder dans chaque `Span` une référence au nom du fichier lui correspondant.

Or, nous utilisons Rust. Et Rust demande des *lifetimes*, qui sont de types génériques représentant la durée de validité d'une référence.

Il a donc fallu passer ces *lifetimes* dans **chaque objet et chaque fonction** qui pouvait bien directement ou indirectement manipuler des `Span`.

De plus, par souci de généralité, nous nous sommes dit qu'il était absurde que les objets représentant les analyseurs lexical et syntaxique soient au courant du type concret des `Span`s et des `Loc`s. En effet, un autre utilisateur de notre bibliothèque aurait pu vouloir utiliser des `Span`s et des `Loc`s qui correspondent mieux à son programme.

Nous avons donc décidé de rendre les types `DFA` (automate fini déterministe) et `PDA` (automate à pile) *génériques*.

Nous croyions initialement avoir fait les bons choix.

Faisons une ellipse sur le processus d'écriture de tous ces composants et observons le résultat :

— le code écrit fonctionne comme attendu mais est particulièrement difficile à lire ;
— nos relations initialement cordiales avec le compilateur de Rust ont été grandement dégradées.

Le cas de Petit Julia

Nous avons donc implémenté le parseur de Petit Julia avec la bibliothèque que nous avons écrite.

Tout ce qui touche à la lecture de la source et à la production de l'AST se trouve dans `parser/src/parse.rs`.

Nous n'avons pas rencontré de difficulté pour écrire notre analyseur lexical : il suffisait d'écrire chaque expression régulière.

Pour l'analyseur syntaxique, il aurait théoriquement aussi suffi d'écrire les règles de production de la grammaire de Petit Julia, puisque notre bibliothèque permet d'écrire facilement des parseurs pour des langages LR(1).

Le problème est que Petit Julia n'admet pas de grammaire LR(1) canonique. Il a donc fallu implémenter des méthodes *ad hoc* pour chaque cas d'ambiguïté

rencontré.

Résultat pour l'analyse syntaxique

Nous avons réussi à construire un système assez général d'analyse syntaxique et lexicale.

La partie due compilateur qui s'occupe de la construction des `AST`s nous semble raisonnablement claire : nous avons réussi à nous ramener à l'écriture d'une grammaire LR(1) classique.

Cependant, le code source de notre bibliothèque d'analyse lexicale et syntaxique ne nous paraît pas de la meilleure facture.

A posteriori, il nous semble que, lorsqu'il est question de réaliser un analyseur syntaxique depuis zéro, seuls deux choix se présentent :

Solution 1 : Un générateur d'analyseur syntaxique

Passer de nombreuses heures à écrire un parseur généraliste, à le tester et à le documenter, sachant que toute mauvaise décision en matière d'organisation a un coût considérable en temps de développement.

Cela est d'autant plus difficile lorsque l'on veut des messages d'erreur vraiment éclairants.

À notre connaissance, seul en utilisant Haskell, l'on peut arriver à des résultats convenables comme `megaparsec` et ses messages d'erreurs supérieurs, comme dans `hasql-th`: https://github.com/nikita-volkov/hasql-th

Solution 2 : Écrire un parseur à la main

Comme bien souvent, on le remarque dans les gros projets sérieux, i.e. `gcc` [2], `clang` ou même `julia`.

Analyse sémantique (typage statique)

Ce code se trouve essentiellement dans `parser/src/typing`, il se compose d'un module `expr.rs` qui va analyser les expressions, un module `data.rs` qui donne des utilitaires et des structures de contextes, un module principal qui va effectuer les deux phases d'analyse `main.rs` et enfin un module `returns.rs` de vérification des `return` dans les fonctions.

Ce module est extrêmement peu élégant car il s'agit d'une première version qui permettra une seconde version refactorisée plus élégante et éventuellement plus puissante. [3]

2. Autrefois, il y avait du yacc mais ça a disparu en 3.X
3. Effectuer une traque des dépendances dans leur ordre avec un parcours en profondeur par exemple.

État des lieux

Contrairement à la partie sur l'analyse syntaxique, l'analyse sémantique n'a consisté qu'à adapter les règles d'inférence en utilisant des approches de parcours de l'arbre AST de façon mutable, tout en modifiant cet AST.

Cette approche est relativement insatisfaisante car on se trimbale beaucoup trop de types options et tantôt `None` est synonyme du type `Any`, tantôt il ne l'est pas.

Beaucoup de code passe son temps à unwrapper des types options et à travailler sur eux ou à effectuer des hypothèses.

En l'occurrence, tout le code actuel ne permet que de façon très inexacte de traquer les portées, ce qui est la raison pour laquelle le test sur la ré-assignation de `nothing` dans la portée globale échoue, il était possible de hacker dans le code un cas particulier sur cette variable, mais cela rajouterait de la complexité inutile pour un code qui sera de toute façon refactorisé en profondeur.

À noter qu'une bonne chose dans ce module est qu'une attention a été portée à déplacer toujours autant que possible les données, plutôt que de les copier (les copies sont essentiellement pour des chaînes de caractères idéalement petites). Et que tout est fait de façon mutable.

Aussi, autant que possible, des structures de données adéquates et rapides ont été utilisé: `HashMap` & `HashSet`, cependant elles ne sont pas forcément utilisés de façon optimal et certaines informations sont doublonnés, parfois il est choisi de faire un scan linéaire sur une structure car il est attendu qu'il y ait peu d'éléments dedans, e.g. le nombre de signatures d'une même fonction.

Futurs travaux

Une refactorisation prendrait la forme suivante :

- Modéliser un nouvel AST qui sera typé d'office avec des informations de portée et autres ;
- Écrire un module permettant de visiter un AST et produire un nouvel AST « efficacement » [4] ;
- Introduire un module pour gérer des `MultiHashMap` [5] ;
- Introduire une abstraction sur les contextes d'environnement (ajouter, retirer, élégamment dans un style fonctionnel des variables) ;
- Retirer autant que possible l'abus de type option et d'unwrapping forcé pour conserver un code qui paniquera le moins possible de façon violente ;
- Améliorer et harmoniser les erreurs avec des span plus précis sur certains endroits et des messages plus clairs sur d'autres ;

4. Lire: complexité temporelle et spatiale.

5. Pour le moment, on dispose d'une version eco+ qui consiste à utiliser un vector en valeur, pas très satisfaisant en plus de faire plein d'erreurs avec.

Futurs travaux

Ce qui n'a pas été fait

Analyse syntaxique

```
Test de ../../target/debug/parser


Partie 1
mauvais ..................
bons ..........................................................................................
ECHEC sur exec/int64.jl (devrait reussir)
.............................
Partie 1: 140/141 : 99%
```

Nous ne gérons pas le cas de l'entier 64 bits, car nous n'avions pas compris l'ensemble des valeurs acceptés comme fermé, nous pensions donc que 2^{63} était exclu.

Analyse sémantique

```
Test de ../../target/debug/parser


Partie 2
mauvais ..................
ECHEC sur typing/bad/testfile-nothing-1.jl (devrait échouer)
...........
bons .............................
ECHEC sur exec/int64.jl (devrait reussir)
.............................
Partie 2: 87/89 : 97%
```

Comme indiqué, précédemment, le test sur `nothing` sera fixé dans la refactorisation.

Les tests

Hors des demandes de l'énoncé, il serait intéressant d'ajouter une façon de pretty-print un AST afin de pouvoir procéder à du fuzzing end-to-end du compilateur, pour ce faire, nous procéderions ainsi :

- Un algorithme de génération aléatoire d'AST basé sur la théorie des types algébriques ;
- Pretty-printing des AST ;
- Compilation du code obtenu et fuzzing guidé par la couverture