Rapport de la partie 2 du projet de compilation

Ryan Lahfa, Julien Marquet

Table des matières

Choix techniques **1**
Quelques éléments d'instructions pour tester le projet 2
Architecture 2
Futurs travaux de la partie 1 3
AST typé vers HIR 3
Le grand terrassement 4
Variables intermédiaires et anti-collision 4
Déclarations de structures 4
Dynamic dispatch 4
Expressions globales et point d'entrée 5
Émission d'un bloc 5
Émission d'une valeur 6
Émission d'une expression 6
Émission des `else` ou une (grosse) erreur 7
Conclusion 7
HIR vers LIR 7
LIR vers assembleur 8
Assembleur vers binaire ELF avec un runtime C 8
Futurs travaux 8
Tests 8
Optimisations 9
Ce qui n'a pas été fait 9
Exécution 10
Interprétation 11

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 sont :

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

— `ir`: l'émetteur de représentations intermédiaires — High Level Intermediate Representation [1], Low Level Intermediate Representation [2], assembleur (partie 2)
— `compiler`: le front-end de compilation (partie 2)

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/compiler`.

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

Attention : Le script `test.sh` a été modifié afin d'accommoder notre compilateur qui ne se contente pas de faire `gcc -no-pie $f.asm` mais plutôt `gcc -no-pie $f.o $runtime.o` après un `as $f.asm -o $f.o`. Aucun fichier de test n'a été modifié en revanche.

Un `Makefile` a été introduit pour faciliter les tests:

— `make test-pjuliac`: effectue `-all`
— `make test-pjuliac-verbose`: effectue `-vall` (qui est juste `-v1, -v2, -v3` enchaînés).

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), mais aussi l'analyse syntaxique de notre propre HIR et LIR.

Pour des détails, nous renvoyons au rapport de partie 1, les détails n'ont pas changé. Quant au typage statique, nous renvoyons à la partie sur les futurs travaux de la partie 1 qui explique les différences avec le projet rendu en partie 1.

Ensuite, la production de code s'effectue en quatre phases :

— AST typé vers HIR
— HIR vers LIR
— LIR vers assembleur
— Assembleur vers binaire ELF avec un runtime C

1. Qu'on abrégera HIR.
2. Qu'on abrégera LIR.

Nous allons nous attacher à détailler chaque phase après l'interlude sur les futurs travaux de la partie 1.

Futurs travaux de la partie 1

Nous avions indiqué quelques travaux que nous comptions entreprendre après le rendu de la partie 1.

Nous rendons compte de ce qui a été fait désormais :

- Modéliser un nouvel AST qui sera typé d'office avec des informations de portée et autres: effectivement, `scope` et les types statiques sont `Any` par défaut ;
- Écrire un module permettant de visiter un AST et produire un nouvel AST « efficacement » [3]: quelques tentatives ont été faites, sans grande satisfaction dans `typing/visit.rs`, le visitor pattern ne se prêtant pas trop à Rust en général, nous avons décidé de rester pragmatique et de rendre juste plus lisible notre façon de parcourir les AST, comme dans `typing/fill.rs` ;
- Introduire un module pour gérer des `MultiHashMap` [4]: nous n'avons pas un véritable type `MultiHashMap` mais nous avons mis une surcouche sur nos environnements grâce à `TypingContext` qui est amplement suffisante ;
- Introduire une abstraction sur les contextes d'environnement (ajouter, retirer, élégamment dans un style fonctionnel des variables): effectué comme dit précédemment, il s'agit de `TypingContext` ;
- 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: c'est effectué avec la conception du nouvel AST ;
- Améliorer et harmoniser les erreurs avec des span plus précis sur certains endroits et des messages plus clairs sur d'autres: autrefois, les variables d'environnement n'étaient que des strings, désormais ils sont des `LocatedIdent` donc ils bénéficient d'une ciblage plus précis en cas d'erreur, il demeure certains endroits où le span n'est pas parfait (les lvalues), mais c'est amplement suffisant ;

De plus, les tests de typages passent intégralement (sauf celui concernant l'entier trop grand).

AST typé vers HIR

Nous partons donc d'un AST typé, cela sous entend par exemple que toutes les `LValue` qui se compose d'une expression (la partie gauche) et d'un nom (la

3. Lire: complexité temporelle et spatiale.

4. 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.

partie droite), i.e. `expr.nom` se voient doté d'un type qui ne peut pas être `Any` sous aucun prétexte.

Le grand terrassement

L'HIR est une représentation où les assignations sont aplaties, par exemple:

```
x = 3(2 + 5)
```

est transformé [5] en:

```
__intermediate0 <- 2 + 5
__intermediate1 <- 3 * __intermediate0
```

Variables intermédiaires et anti-collision

Le point précédent a montré l'existence de variables intermédiaires automatiquement émises.

Leur émission est simple, il y a un compteur local par fonction (réinitialisé entre chaque émission de fonction), celui-ci vérifie qu'il n'émet jamais une variable intermédiaire qui puisse être doublon dans le contexte local.

Par exemple, si `__intermediate0` existe déjà dans le code utilisateur (paramètre, variable locale ou variable globale), le compteur augmente et cherche la prochaine occurrence disponible.

Une alternative aurait été le renommage côté utilisateur, mais ça nous paraissait trop coûteux de reparcourir les blocs à la recherche des occurrences.

Déclarations de structures

Les structures sont émises directement car elles sont supportées en HIR, mais sans leur types, donc cette information est jetée.

Elle pourrait être utilisée à des fins d'optimisation, mais nous n'avons pas eu le temps d'examiner comment.

Dynamic dispatch

Le dynamic dispatch est écrit à ce moment, l'on aurait pu disposer d'une fonction `typeof` dans la langage de Petit Julia et écrire la surcouche directement en Petit Julia pour se faciliter la vie, mais, nous avons décidé de le faire directement au niveau de l'HIR.

Concrètement, on se donne une paire `(nom, fonctions possibles)`, puis on renomme toutes les fonctions possible dans un ordre quelconque [6].

Puis ensuite, on les compile vers le HIR, les rajoute dans les déclarations.

5. Transformations non contractuelles.
6. Mais qui reste le même pendant la compilation.

Enfin, on émet le dispatcher, pour ce faire, on procède en plusieurs temps :

- On calcule des informations de sélectivité (poids) des fonctions possibles en fonction de leur signature ;
- On s'assure qu'il n'existe pas plus d'une fonction dite « générique », i.e. de poids nul, donc tous les types dans sa signature sont `Any` ;
- On trie ensuite par sélectivité croissante et on fold des blocs conditionnels en partant de la fin, i.e. le cas où aucune fonction possible n'a été choisie, donc une erreur de dispatch dynamique est émise [7] et en chaînant les constructions `if-else` jusqu'à arriver au premier `if` sur la condition la plus sélective ;
- Le bloc qu'on donne à chacune de ces conditions contient: un appel de fonction à la version renommée et une assignation à une variable de retour locale
- À la fin du dispatcher, on retourne la variable de retour locale

Expressions globales et point d'entrée

Les expressions globales sont traitées en les englobant dans un point d'entrée `__start$k` où `$k` est un entier pour empêcher les collisions en cherchant la liste des fonctions produites après parcours de toutes les fonctions (et après émission des dynamic dispatch).

Ensuite, toutes les assignations sont marquées comme variables globales, `nothing` de type `Nothing` est ajouté dans le contexte global.

Par ailleurs, afin d'éviter les collisions au niveau local et global, nous effectuons un renommage de toutes les variables globales et nous réutilisons l'information de portée émise par le typage pour déterminer si tantôt un accès est au niveau local ou au niveau global et nous émettons directement l'accès renommé [8]. Ainsi, on peut lire sur le HIR lorsqu'une variable globale est utilisée grâce au préfixe `_g$nom$k` avec `$k` l'entier anti-collision et `$nom` le nom original de la variable.

Enfin, le point d'entrée est émis `__start$k` avec une signature vide sans retour, dont le corps est la concaténation de toutes les expressions globales émises.

Émission d'un bloc

Un bloc peut être émis avec l'autorisation ou non de traiter les retours implicites en cas de l'absence d'un point-virgule à la fin.

Dans ce cas, le bloc est émis en tant que bloc à valeur et un retour explicite est ajouté avec la valeur de retour émise. Dans le cas échéant, le bloc est émis en tant que bloc sans valeur.

7. D'ailleurs, à ce moment, on utilise une fonction « native » panic qui sera détaillé ultérieurement.
8. Ou corrigé?

Un bloc à valeur c'est l'émission du bloc sans valeur formé par toutes les expressions sauf la dernière et l'émission de la valeur de la dernière expression. Un bloc sans valeur c'est l'émission des déclarations formé par toutes les expressions qu'il contient.

Émission d'une valeur

L'émission d'une valeur c'est la paire `(déclarations requises pour créer la valeur, valeur)`, par exemple, pour émettre la valeur constante `1`, `([], hir::Val::Cst(hir::Type::Int64, 1 as u64))` suffit.

C'est ici donc que l'aplatissement se fait précisément, en déroulant les expressions tout en émettant des variables intermédiaires internes.

Nous n'attribuons pas de valeur, si ce n'est celle de `nothing` donc, à certaines expressions comme l'assignation, la boucle for ou while, mais nous émettons les instructions requises pour créer les effets de bord.

Certaines valeurs sont donc essentiellement des effets de bord.

Le cas de la valeur d'un appel de fonction ou de la création d'une structure est un peu particulier, nous avons explicitement supporté les fonctions et structures à même nom et signature, cependant, nous n'avons aucun moyen de savoir s'il faut appeler l'un ou l'autre dans cette situation.

Ainsi, tout comme Julia, nous faisons le choix d'ignorer et de préférer la fonction à la structure:

```
struct S end
function S() end

s = S()
```

conduit à `typeof(s) == Nothing`.

Une attention particulière a été portée sur l'émission des valeurs résultant d'une opération binaire, en effet, la puissance n'étant pas une instruction assembleur élémentaire dans les jeux classiques, nous avons introduit un mécanisme de runtime permettant d'appeler des fonctions natives.

Ainsi, nous transformons l'opération binaire `a ^ b` en appel « natif » `pow(a, b)`, nous reviendrons sur le runtime et les fonctions natives dans les sections concernant le LIR et l'assembleur.

Émission d'une expression

L'émission d'une expression se résume à émettre les instructions qu'il faut pour créer des effets de bord, les valeurs ici ne sont pas considérés.

Typiquement, se produit ici la transformation d'une boucle `for` en boucle `while` en injectant la création d'un compteur, le calcul des conditions, et la mise à jour

du compteur.

Ou alors, l'assignation dite « composée » ou « simple » (sans membre gauche):

— « simple » : on se contente de l'assigner directement, c'est supporté par l'HIR ;
— « composée » : nous calculons la valeur du membre gauche, l'expression à droite, puis nous vérifions que le membre gauche est bien une structure correctement définie et allouée, et nous émettons l'assignation

Émission des `else` ou une (grosse) erreur

Nous avons opté dans notre AST pour une représentation des `elseif` explicites, elle permet donc une reproduction à l'identique du code original, mais elle souffre du défaut d'introduire beaucoup de cas particuliers pour le parcours des branches `else`.

En particulier, au lieu de se retrouver à traiter les `elseif` comme des `if`, on les traite à part juste avec un peu de différence, et l'on abstrait pas par dessus.

La seule solution est donc de transformer les noeuds `Elseif` en noeuds `If` et les ré-émettre à la volée, cela a été fait dans quelques endroits du typage, mais c'était fastidieux pour des raisons propre à Rust et par manque de temps, nous avons donc décidé d'accepter une violation du « Don't Repeat Yourself » afin d'aller plus vite [9]

Il en demeure pas moins que c'est un défaut de la propreté du code, qui disparait dans les niveaux plus bas de représentations cela dit.

Conclusion

À la fin de tous ces processus, décrit dans `ir/ast_to_hir.rs`, nous obtenons un `hir::Source` qui est la représentation intermédiaire, qu'on peut pretty-print pour examiner ou passer au prochain niveau de compilation, ou optimiser.

Nous allons donc passer au prochain niveau de compilation.

Il est possible d'examiner l'HIR avec le flag `-h` sur le compilateur qui écrit un fichier `$output.hir`.

HIR vers LIR

Dans le HIR, tous les objets contiennent un identifiant de type et une valeur, et les types des structures sont explicites. Dans le LIR, tout est sur 64 bits, et il n'y a aucune notion de structure.

Le but de la compilation du HIR vers le LIR est donc de déconstruire ces deux abstractions.

9. Effectivement, `vim` est pratique pour cette tâche.

Chaque variable du HIR est donc découpée en deux variables du LIR : une pour le type et une pour la valeur.

La compilation prend en compte la propagation dynamique des types : si on a `x <- a + b`, alors on dit que le type de `x` devient `Int64`.

Les structures sont collectées lors de la compilation. On décide un ordonnement des champ des structures (on associe une adresse à chaque champ).

Les fonctions du LIR renvoient deux valeurs, qui composent une seule valeur au sens du HIR.

LIR vers assembleur

Le LIR est conçu pour être quasi-trivialement compilable vers de l'assembleur. La seule difficulté est la gestion des appels de fonctions.

Lors de la compilation, on fait attention à respecter l'ABI System V (avec un effet de bord utile : on peut directement appeler des fonctions écrites en C).

La seule différence avec l'ABI est que les fonction renvoient toutes 2 valeurs : la première dans `rax`, et la deuxième dans `rdx`.

Les appels aux fonctions « natives » (celles du runtime) sont un peu spéciaux car les fonctions de C ne peuvent renvoyer qu'une seule valeur. Pour régler ce problème, avant un appel à une fonction native, on alloue deux quadwords sur la pile, et les deux premiers arguments de la fonction appelée seront les deux pointeurs vers les deux emplacements mémoire que l'on vient d'allouer.

Assembleur vers binaire ELF avec un runtime C

Il nous suffit de compiler le code généré et le runtime, et de les lier.

Futurs travaux

Tests

Nous avions indiqué précédemment dans la partie 1 notre souhait de fuzzer notre compilateur end-to-end, nous n'avons pas conçu de pretty-printing d'AST de Julia, mais en revanche, nous avons des pretty printers pour les représentations intermédiaires.

Ainsi, il est possible d'effectuer cette génération aléatoire pour vérifier si notre production de code ne peut provoquer des comportements indéfinis conduisant à des désastres cosmiques.

Par manque de temps, cela ne sera pas fait avant le rendu de ce projet.

Optimisations

Nous allons décrire les optimisations que nous avions envisagé avec ces niveaux intermédiaires.

L'optimisation principale à envisager est de limiter la redondance des valeurs dans les AST des deux langages intermédiaires (par exemple, on passe beaucoup de temps à recopier des noms de variables).

Une première solution est de simplement utiliser des types `Cow` (*Copy On Write*) de Rust.

On aurait aussi pu chercher des optimisations plus intelligentes (en cherchant à utiliser des références pour les objets qui sont susceptibles d'être souvent réutilisés) mais cela aurait induit beaucoup trop de *borrow fighting* et nous avons préféré nous abstenir.

HIR Le HIR est prévu pour permettre d'introduire du Single Static Assignment.

LIR Le LIR est prévu pour permettre l'optimisation de l'allocation de registres (l'idée du LIR est que la seule différence entre ce langage et l'assembleur est que l'on a des pseudo-registres).

En l'état, on introduit beaucoup de movements entre les registres qui pourraient être évités avec un algorithme d'allocation un tant soit peu efficace.

Runtime Le système de « runtime » que nous avons implémenté permet d'introduire du code en C assez complexe, on aurait donc pu implémenter un *garbage collector*.

Perfect Optimizer 3000 On pourrait décider d'implémenter un interpréteur et d'essayer d'interpréter les programmes au moment de la compilation pour produire des binaires qui se contentent d'afficher le résultat.

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 passons pas le test “exec/int64.jl” : notre compilateur refuse la chaîne `-9223372036854775808` car le nombre `9223372036854775808` n'est pas représentable comme un entier signé sur 64 bits. Normalement la chaîne `-9223372036854775808` représente le nombre `-9223372036854775808`, mais pas ici. En fait, si l'on suit exactement la grammaire de Petit Julia donnée dans le sujet :

> Les constantes obéissent aux expressions regulières et suivantes : [...]
> ::= ^+ [...] Les constantes entières ne doivent pas dépasser 2^63

Cela signifie que `-9223372036854775808` doit être interprété comme `-9223372036854775808` (l'expression régulière qui définit les entiers n'accepte pas de signe `-`).

Pour régler ce problème, on peut essayer de jouer sur l'interprétation de “ne pas dépasser”. Doit-on accepter 2^63 ?

Nous pensons que non, pour deux raisons :

- 2^63 n'est pas représentable comme entier signé de 64 bits. Il aurait fallu soit représenter les `Int64` sur 128 bits, soit les considérer comme non signés. Aucune de ces solutions ne nous semblait pertinente.
- Le mot « dépasser » est même sujet à interprétation. Nous choisissons de définir « dépasser » comme « être plus grand que », et « plus grand » est interprété classiquement (puisque l'on est en France, qui plus est dans l'école de N.Bourbaki) comme au sens de l'ordre naturel, c'est à dire que « plus grand » est équivalent à « supérieur ou égal ».

Donc *`2^{63}` dépasse `2^{63}`*, et donc que l'entier `9223372036854775808` ne doit pas être accepté.

Finalement, notre analyseur syntaxique est bien conforme au sujet...

Analyse sémantique

```
Partie 2
mauvais ..............................
bons .................................
ECHEC sur exec/int64.jl (devrait reussir)
.............................
Partie 2: 88/89 : 98%
```

Exécution

```
Partie 3:
Compilation : 52/53 : 98%
Code produit : 38/53 : 71%
Comportement du code : 35/53 : 66%
```

Concrètement, les majeurs problèmes de la compilation sont:

— l'absence de protection des types à la frontière des appels de fonction: un début d'implémentation se trouve dans notre repo dans `type-guards` ;
— l'absence d'un dynamic dispatch en rapport avec la protection des types des valeurs
— des confusions sur la division et l'opérateur modulo
— des problèmes autour des structures

Interprétation

Aucun interpréteur n'a été écrit.