

Heuristique d'inlining complexe

Adrien Simonnet

Avril - Septembre 2023

Abstract

Ceci constitue mon rapport de stage de fin de cursus Science et Technologie du Logiciel (STL) à Sorbonne Université réalisé chez OCamlPro au sein de l'équipe flambda. Ce stage a consisté à proposer des heuristiques d'inlining pour le compilateur du langage OCaml, spécialité de l'entreprise. Découvrir et travailler sur un compilateur complexe comme celui-ci n'a pas été jugé envisageable par mes tuteurs de stage, Vincent Laviron et Pierre Chambart, c'est la raison pour laquelle j'ai évolué sur un langage "jouet". Mon stage se terminant après la soutenance, ce rapport rend compte essentiellement de mon travail sur les différentes représentations intermédiaires et analyses nécessaires à l'inlining. J'apporte néanmoins quelques idées d'heuristiques que je vais être amené à étudier en détails d'ici la fin du stage.

Table des matières

Introduction	4
Langage source	4
Lambda-calcul	4
Opérations élémentaires	4
Fermetures (mutuellement) récursives	4
Types Somme et filtrage par motifs	5
Langage utilisé	5
Analyse lexicale	6
Lexique	6
Analyseur lexical	6
Analyse syntaxique	7
Arbre de Syntaxe Abstraite (AST)	7
Identificateurs	7
Filtrage par motif	7
Opérateurs binaires	7
Expressions	7
Analyseur syntaxique	8
Résolution des noms	9
Arbre de Syntaxe Abstraite rafraîchi (AST')	9
Identificateurs	9
Opérateurs binaires	9
Expressions	9
Rafraîchissement	10
Analyse du flot de contrôle	12
Graphe de flot de contrôle	12
Identificateurs	12
Expressions	12
Instructions	12
Type de blocs	13
Conversion CFG	14
Nettoyage des alias	16
Taille	16
Recensement des variables et appels de blocs	16
Spécialisation	16
Algorithme de spécialisation	16
Choix des blocs à spécialiser	17
Analyse de valeurs par interprétation abstraite	17
Points d'allocation	17
Domaines	18
Valeur abstraite	19

Blocs	19
Contexte d'appel	19
Pile	19
Bloc à analyser	20
Abstraction de la pile	20
Algorithme d'analyse	20
Idées de terminaison	21
Propagation	22
CFG exécutable	23
Langage	23
Identifiants	23
Expressions	23
Instructions	24
Bloc	24
Génération du CFG exécutable	24
Taille	25
Recensement des variables et appels de blocs	25
Inlining	25
Choix des blocs à inliner	26
Interprétation	26
Validité des phases de compilation	26
Benchmarks	26
Heuristiques d'inlining	27
Blocs de petite taille	27
Inlining partiel	27
Conclusion	29

Introduction

L'inlining consiste à injecter le corps d'une fonction en lieu et place d'un appel vers celle-ci dans l'objectif d'accélérer l'exécution du code (ou dans certains cas en diminuer sa taille). Néanmoins copier le corps d'une fonction peut faire augmenter la taille du code et conduire à de grosses pertes de performances lorsque certains seuils sont franchis. Vu la difficulté que serait de faire une analyse approfondie du meilleur choix d'inlining l'idée a été de se concentrer sur des heuristiques qui fonctionneront bien la plupart du temps. Cette optimisation est actuellement effectuée dans le compilateur natif par la série d'optimisations `flambda`, qui sera plus tard remplacé par `flambda2` actuellement en développement. Le langage "jouet" sur lequel j'ai travaillé n'est rien d'autre que la partie intéressante d'OCaml pour y appliquer l'inlining.

Langage source

Dans le cadre du stage je n'ai traité que les fonctionnalités d'OCaml nécessaires pour aborder la plupart des cas intéressants de l'inlining et suffisantes pour obtenir un langage Turing-complet. En particulier comme je ne me suis concentré que sur le noyau fonctionnel d'OCaml, les objets, références, exceptions ou autres fonctionnalités du langage ont été ignorés.

Lambda-calcul

Les trois constructions du lambda-calcul que sont les variables, l'application et l'abstraction sont évidemment des fonctionnalités indispensables.

```
let id = fun x -> x in
let x = id y in x
```

Opérations élémentaires

La manipulation des entiers et le branchement conditionnel sont incontournables pour réaliser des programmes dignes de ce nom.

```
if c then x + 1 else x - 1
```

Fermetures (mutuellement) récursives

La prise en compte de la récursivité est fondamentale, premièrement en terme d'expressivité du langage (pour réaliser des tests poussés), deuxièmement il est intéressant de prendre en compte la complexité liée à l'impossibilité d'inliner tous les appels récursifs potentiels. La récursivité mutuelle permet d'ajouter une couche de complexité notamment lors de l'analyse.

```
let rec f = fun x -> g x
and g = fun y -> f y in f 0
```

Types Somme et filtrage par motifs

La possibilité de construire des types Somme est une des fonctionnalités essentielles d'OCaml et permet de représenter quasiment n'importe quelle structure de données. De plus, de la même manière que le branchement conditionnel, le filtrage par motifs se prête particulièrement bien à l'inlining puisque connaître le motif peut permettre de filtrer de nombreuses branches et donc d'alléger un potentiel inlining.

```
type int_list =  
  | Nil  
  | Cons of int * int_list  
  
let rec map = fun f -> fun l ->  
  match l with  
  | Nil -> Nil  
  | Cons (x, ls) -> Cons (f x, map f ls)  
in map (fun x -> x + 10) (Cons (1, Cons (2, Nil)))
```

Langage utilisé

J'ai réalisé le compilateur en OCaml, en cohérence avec le langage source et l'expertise d'OCamlPro.

Analyse lexicale

L'analyse lexicale est la première étape de la compilation et convertit le programme source vu comme une chaîne de caractères en une liste de jetons.

Lexique

Le lexique source est un sous-ensemble de celui d'OCaml pour supporter les fonctionnalités qui m'intéressent. Je n'ai pas jugé pertinent d'indiquer le nom des jetons.

- `(* *)` pour les commentaires
- `()` pour le parenthésage
- `+ -` pour les opérations sur les entiers
- `if then else` pour le branchement conditionnel
- `fun ->` pour la création de fermeture
- `let rec = and in` pour les déclarations (mutuellement récursives)
- `type of | *` pour la déclaration d'un type somme
- `,` pour les constructeurs
- `match with | _` pour le filtrage par motif
- `['0'-'9']+` pour la déclaration d'un entier (positif)
- `['A'-'Z']['a'-'z']['A'-'Z']['0'-'9']['_']*` pour désigner un constructeur
- `['a'-'z']['a'-'z']['A'-'Z']['0'-'9']['_']*` pour désigner une variable
- `[' ' '\t' '\n' '\r']` pour l'indentation et les sauts de ligne/retours chariot

Analyseur lexical

Les jetons de l'analyse lexicale sont générés par OCamllex.

Analyse syntaxique

L'analyse syntaxique est la seconde étape de la compilation et va convertir les jetons en un arbre de syntaxe abstraite.

Arbre de Syntaxe Abstraite (AST)

La grammaire est la même que celle d'OCaml pour l'ensemble des jetons supportés.

Identificateurs

Le nom des variables et le nom des constructeurs sont des chaînes de caractères.

$\mathbb{S} := \text{string}$ (chaînes de caractères)

$\mathbb{V} := \mathbb{S}$ (variables)

$\mathbb{T} := \mathbb{S}$ (tags)

Filtrage par motif

J'ai choisi de ne supporter que l'essentiel pour ce qui est du filtrage par motif. Contrairement à OCaml, qui autorise la déconstruction à chaque déclaration de variable, il n'est possible de déconstruire des termes que dans un filtrage par motif.

Deconstructor : $\mathbb{T} \times \mathbb{V}^* \mapsto \mathbb{M}$ pour filtrer un tag.

Joker : $\mathbb{V} \mapsto \mathbb{M}$ pour filtrer tous les cas restants.

Opérateurs binaires

Les opérateurs binaires se limitent pour l'instant aux opérations sur les entiers.

Add : \mathbb{B} correspond à l'addition.

Sub : \mathbb{B} correspond à la soustraction.

Expressions

Les expressions constituent la base d'un langage fonctionnel. Je ne réalise aucune vérification de typage, c'est pour cela que je traite les types comme de simples chaînes de caractères.

Var : $\mathbb{V} \mapsto \mathbb{E}$ désigne une variable.

Fun : $\mathbb{V} \times \mathbb{E} \mapsto \mathbb{E}$ fabrique une fermeture.

App : $\mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ applique une fermeture à son argument.

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ associe le résultat d'une expression à une variable.

LetRec : $(\mathbb{V} \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$ définit des fermetures récursives (uniquement des fermetures).

Int : $\mathbb{Z} \mapsto \mathbb{E}$ génère un entier.

Binary : $\mathbb{B} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ représente une opération binaire.

If : $\mathbb{E} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ est le branchement conditionnel.

Type : $\mathbb{S} \times (\mathbb{T} \times \mathbb{S})^* \times \mathbb{E} \mapsto \mathbb{E}$ fabrique un type (ses constructeurs).

Constructor : $\mathbb{T} \times \mathbb{E}^* \mapsto \mathbb{E}$ fabrique une valeur taggée à partir d'un constructeur.

Match : $\mathbb{E} \times (\mathbb{M} \times \mathbb{E})^* \mapsto \mathbb{E}$ filtre une valeur taggée.

Analyseur syntaxique

L'AST est généré par Menhir à l'aide des jetons générés par l'analyse lexicale.

Résolution des noms

La résolution des noms est la troisième étape de la compilation.

Arbre de Syntaxe Abstraite rafraîchi (AST')

L'arbre de syntaxe abstraite rafraîchi est construit à partir de l'AST en transformant les noms et les types (pour l'instant limités aux constructeurs) en identificateurs frais et uniques.

Identificateurs

Le nom des variables et le nom des constructeurs sont désormais représentés par des entiers naturels uniques.

$\mathbb{V} := \mathbb{N}$ (variables)

$\mathbb{T} := \mathbb{N}$ (tags)

Opérateurs binaires

Les opérateurs binaires se limitent toujours aux opérations sur les entiers.

Add : \mathbb{B} correspond à l'addition.

Sub : \mathbb{B} correspond à la soustraction.

Expressions

Pour toutes les expressions, à l'exception du filtrage par motif, seul le type des identificateurs change. Les noms des constructeurs ont reçus un index relatif à leur position dans la déclaration du type qui sera par la suite utilisé dans les filtrages par motifs.

Var : $\mathbb{V} \mapsto \mathbb{E}$ désigne une variable.

Fun : $\mathbb{V} \times \mathbb{E} \mapsto \mathbb{E}$ fabrique une fermeture.

App : $\mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ applique une fermeture à son argument.

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ associe le résultat d'une expression à une variable.

LetRec : $(\mathbb{V} \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$ définit des fermetures récursives (uniquement des fermetures).

Int : $\mathbb{Z} \mapsto \mathbb{E}$ génère un entier.

Binary : $\mathbb{B} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ représente une opération binaire.

If : $\mathbb{E} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ est le branchement conditionnel.

Constructor : $\mathbb{T} \times \mathbb{E}^* \mapsto \mathbb{E}$ fabrique une valeur taggée à partir d'un tag.

Match : $\mathbb{E} \times (\mathbb{T} \times \mathbb{V}^* \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$ filtre une valeur taggée.

Rafraîchissement

Toutes les variables sont rafraîchies et conservées dans une table des symboles. Les variables libres dans le programme sont autorisées, également rafraîchies et ajoutées dans une table à part. L'acceptation de variables libres dans le programme permet à mes yeux de faciliter la gestion du non-déterminisme et d'éviter toute ambiguïté lors de l'analyse. En effet les entrées-sorties peuvent être vues comme des variables libres qui ne sont connues qu'au moment de l'exécution du programme, ce qui permet de s'assurer par exemple qu'un affichage sur la sortie ne serait pas optimisé.

$$\mathbb{E}_{ast} \times (\mathbb{V}_{ast} \mapsto \mathbb{V}) \times (\mathbb{T}_{ast} \mapsto \mathbb{T}) \vdash_{ast} \mathbb{E} \times (\mathbb{V} \mapsto \mathbb{V}_{ast}) \times (\mathbb{V}_{ast} \mapsto \mathbb{V})$$

$$e \ A \ C \vdash_{ast} e' \ S \ L$$

- e est l'expression AST à compiler ;
- A est la table des abstractions existantes dans l'environnement de e ;
- C est la table des constructeurs dans l'environnement de e ;
- e' est l'expression générée ;
- S est la table des variables substituées dans e' ;
- L est la table des variables libres de e .

Par la suite, je note \bar{x} l'identifiant unique donné à une variable x , e' l'expression générée par l'expression e , \emptyset une table vide, $x := y$ une entrée de table dont l'étiquette est x et la valeur est y et $X \sqcup Y$ l'union de deux tables. Deux tables dont les clés sont des identificateurs uniques ou des variables libres étant nécessairement disjointes, leur union est définie de manière naturelle comme l'union de leurs clés avec leurs valeurs associées.

Il est important d'injecter les variables libres dans la table des abstractions pour éviter de détecter plusieurs fois la même variable libre.

$$\frac{}{(\text{Int } i) \ A \ C \vdash_{ast} (\text{Int } i) \ \emptyset \ \emptyset} \quad (\text{Int})$$

$$\frac{\begin{array}{c} e_1 \ A \ C \vdash_{ast} e'_1 \ S_{e_1} \ L_{e_1} \\ e_2 \ (A \sqcup L_{e_1}) \ C \vdash_{ast} e'_2 \ S_{e_2} \ L_{e_2} \end{array}}{(\text{Binary } \diamond \ e_1 \ e_2) \ A \ C \vdash_{ast} (\text{Binary } \diamond \ e'_1 \ e'_2) \ (S_{e_1} \sqcup S_{e_2}) \ (L_{e_1} \sqcup L_{e_2})} \quad (\text{Binary})$$

$$\frac{e \ (A \sqcup \{x := \bar{x}\}) \ C \vdash_{ast} e' \ S_e \ L_e}{(\text{Fun } x \ e) \ A \ C \vdash_{ast} (\text{Fun } \bar{x} \ e') \ (S_e \sqcup \{\bar{x} := x\}) \ L_e} \quad (\text{Fun})$$

$$\frac{x \in \mathcal{D}(A)}{(\text{Var } x) \ A \ C \vdash_{ast} (\text{Var } A(x)) \ \emptyset \ \emptyset} \quad (\text{Var1})$$

$$\frac{x \notin \mathcal{D}(A)}{(\text{Var } x) \ A \ C \vdash_{ast} (\text{Var } \bar{x}) \ \emptyset \ \{x := \bar{x}\}} \quad (\text{Var2})$$

$$\frac{e_1 A C \vdash_{\text{ast}} e'_1 S_{e_1} L_{e_1} \quad e_2 (A \sqcup L_{e_1} \sqcup \{x := \bar{x}\}) C \vdash_{\text{ast}} e'_2 S_{e_2} L_{e_2}}{(\text{Let } x \ e_1 \ e_2) A C \vdash_{\text{ast}} (\text{Let } \bar{x} \ e'_1 \ e'_2) (S_{e_1} \sqcup S_{e_2} \sqcup \{\bar{x} := x\}) (L_{e_1} \sqcup L_{e_2})} \quad (\text{Let})$$

$$\frac{e_1 A C \vdash_{\text{ast}} e'_1 S_{e_1} L_{e_1} \quad e_2 (A \sqcup L_{e_1}) C \vdash_{\text{ast}} e'_2 S_{e_2} L_{e_2} \quad e_3 (A \sqcup L_{e_1} \sqcup L_{e_2}) C \vdash_{\text{ast}} e'_3 S_{e_3} L_{e_3}}{(\text{If } e_1 \ e_2 \ e_3) A C \vdash_{\text{ast}} (\text{Binary } e'_1 \ e'_2 \ e'_3) (S_{e_1} \sqcup S_{e_2} \sqcup S_{e_3}) (L_{e_1} \sqcup L_{e_1} \sqcup L_{e_3})} \quad (\text{If})$$

$$\frac{e_1 A C \vdash_{\text{ast}} e'_1 S_{e_1} L_{e_1} \quad e_2 (A \sqcup L_{e_1}) C \vdash_{\text{ast}} e'_2 S_{e_2} L_{e_2}}{(\text{App } e_1 \ e_2) A C \vdash_{\text{ast}} (\text{App } e'_1 \ e'_2) (S_{e_1} \sqcup S_{e_2}) (L_{e_1} \sqcup L_{e_2})} \quad (\text{App})$$

$$\frac{e A (C \sqcup \{t_i := i, i \in 1 \dots n\}) \vdash_{\text{ast}} e' S_e L_e}{(\text{Type } s(t_i, s_i)_{i=1}^{i=n} \ e) A C \vdash_{\text{ast}} e' S_e L_e} \quad (\text{Type})$$

$$\frac{e_1 A C \vdash_{\text{ast}} e'_1 S_{e_1} L_{e_1} \quad \dots \quad e_n \left(\bigsqcup_{i=1}^{i=n-1} L_{e_{i-1}} \sqcup A \right) C \vdash_{\text{ast}} e'_n S_{e_n} L_{e_n}}{(\text{Constructor } s(e_i)_{i=1}^{i=n}) A C \vdash_{\text{ast}} (\text{Constructor } (C(s)) (e'_i)_{i=1}^{i=n}) \left(\bigsqcup_{i=1}^{i=n} S_{e_i} \right) \left(\bigsqcup_{i=1}^{i=n} L_{e_i} \right)} \quad (\text{Constructor})$$

$$\frac{e_1 (\{x_i := x'_i, i \in 1 \dots n\} \sqcup A) C \vdash_{\text{ast}} e'_1 S_{e_1} L_{e_1} \quad \dots \quad e_n \left(\bigsqcup_{i=1}^{i=n-1} S_{e_{i-1}} \sqcup \{x_i := x'_i, i \in 1 \dots n\} \sqcup A \right) C \vdash_{\text{ast}} e'_n S_{e_n} L_{e_n}}{(\text{LetRec } (x_i, e_i)_{i=1}^{i=n} \ e) A C \vdash_{\text{ast}} (\text{LetRec } (x'_i, e'_i)_{i=1}^{i=n} \ e') \left(\bigsqcup_{i=1}^{i=n} S_{e_i} \right) \left(\bigsqcup_{i=1}^{i=n} L_{e_i} \right)} \quad (\text{LetRec})$$

$$\frac{(\text{todo})}{(\text{Match } e(m_i, e_i)_{i=1}^{i=n}) A C \vdash_{\text{ast}} \left(\text{Match } e' \left(t'_i, (a_i^j)_{j=1}^{j=m_i}, e'_i \right)_{i=1}^{i=n} \right) \left(\bigsqcup_{i=1}^{i=n} S_{e_i} \sqcup S_e \right) \left(\bigsqcup_{i=1}^{i=n} L_{e_i} \sqcup L_e \right)} \quad (\text{Match})$$

Analyse du flot de contrôle

La conversion CPS/CFG transforme l'AST' en un ensemble de basic blocs. A l'origine il s'agissait d'une conversion CPS, mais l'explicitation des variables libres et la décontextualisation des blocs fait qu'aujourd'hui elle ressemble davantage à une conversion vers un CFG. L'idée est de perdre le moins d'informations possible du programme source tout en ayant sous la main une représentation intermédiaire qui permette une analyse simple et puissante.

Graphe de flot de contrôle

La différence notable avec l'AST' est l'apparition des blocs (avec explicitation des variables libres) et l'absence d'expressions imbriquées.

Identificateurs

On retrouve les identificateurs pour les variables et les tags, auxquels s'ajoute un identificateur pour les pointeurs.

$\mathbb{V} := \mathbb{N}$ (variables)

$\mathbb{T} := \mathbb{N}$ (tags)

$\mathbb{P} := \mathbb{N}$ (pointeurs)

Expressions

Les expressions correspondent aux instructions élémentaires qui construisent des valeurs.

$\text{Int} : \mathbb{Z} \mapsto \mathbb{E}$ génère un entier.

$\text{Var} : \mathbb{V} \mapsto \mathbb{E}$ crée un alias de variable.

$\text{Add} : \mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$ additionne deux entiers.

$\text{Sub} : \mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$ soustrait deux entiers.

$\text{Closure} : \mathbb{P} \times \mathbb{V}^* \mapsto \mathbb{E}$ fabrique une fermeture.

$\text{Constructor} : \mathbb{T} \times \mathbb{V}^* \mapsto \mathbb{E}$ fabrique une valeur taggée.

Instructions

Une instruction est soit une déclaration soit un branchement. Une déclaration construit une valeur à partir d'une expression et l'associe à un identifiant unique. Les valeurs ne peuvent être construites qu'à partir de constantes ou identifiants. Un branchement représente le transfert d'un basic block à un autre que ce soit par le biais d'un appel (fermeture), d'un retour de fonction (return) ou d'un saut conditionnel (filtrage par motif). Comme chaque bloc explicite ses variables libres (arguments), celles-ci doivent apparaître dans les branchements. Les variables libres à destination de différents blocs ne sont jamais réunies afin de conserver un maximum d'informations sur leurs origines.

$\text{Let} : \mathbb{V} \times \mathbb{E} \times \mathbb{I} \mapsto \mathbb{I}$ assigne le résultat d'une expression à une variable.

$\text{Call} : \mathbb{V} \times \mathbb{V}^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \mapsto \mathbb{I}$ branche vers le bloc avec l'environnement contenu dans la fermeture puis continue l'exécution avec le contexte spécifié.

$\text{CallDirect} : \mathbb{P} \times \mathbb{V} \times \mathbb{V}^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \mapsto \mathbb{I}$ est identique à Call à l'exception que le pointeur du bloc vers lequel brancher est connu suite à une étape d'analyse (cette instruction n'est par conséquent jamais générée depuis l'AST').

$\text{If} : \mathbb{V} \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$ branche dans le premier bloc si la valeur de la condition est différente de 0, dans le deuxième sinon. Les variables libres (dernier paramètre) sont implicitement passées aux deux branches et seront utilisées par le bloc qui sera exécuté après.

$\text{MatchPattern} : \mathbb{V} \times (\mathbb{T} \times \mathbb{V}^* \times \mathbb{P} \times \mathcal{P}(\mathbb{V}))^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$ branche soit vers un des blocs lorsque la valeur de la condition correspond au motif de l'un d'eux, soit vers le bloc par défaut. Les variables libres (dernier paramètre) sont implicitement passées à toutes les branches et seront utilisées par le bloc qui sera exécuté après.

$\text{Return} : \mathbb{V} \mapsto \mathbb{I}$ renvoie la valeur de cette variable au bloc appelant depuis une fermeture.

$\text{IfReturn} : \mathbb{P} \times \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$ renvoie la valeur de cette variable au bloc appelant depuis un branchement conditionnel.

$\text{MatchReturn} : \mathbb{P} \times \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$ renvoie la valeur de cette variable au bloc appelant depuis une branche d'un filtrage par motif.

$\text{ApplyBlock} : \mathbb{P} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$ est utilisé par LetRec .

Type de blocs

Chaque bloc est défini différemment en fonction de l'expression à partir de laquelle il est construit (fermeture, retour de fonction ou saut conditionnel), est clos (il explicite les arguments dont il a besoin) et contient une suite de déclarations de variables suivies d'une instruction de branchement. Le dernier argument correspond toujours aux variables libres du bloc (environnement dans le cas d'une fermeture).

$\text{Clos} : \mathbb{V}^* \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ est une fermeture avec ses arguments.

$\text{Return} : \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ sera exécuté après un appel de fermeture avec comme argument son résultat.

$\text{IfBranch} : \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ est un branchement conditionnel avec les variables libres qui seront passées au bloc exécuté ensuite.

$\text{IfJoin} : \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ sera exécuté après un branchement conditionnel avec comme argument son résultat.

$\text{MatchBranch} : \mathbb{V}^* \times \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ est un branchement d'un filtrage par motif avec ses arguments (charge du constructeur ou aucun pour le branchement par défaut) et les variables libres qui seront passées au bloc exécuté ensuite.

$\text{MatchJoin} : \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ sera exécuté après un branchement d'un filtrage par motif avec comme argument son résultat.

$\text{Cont} : \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$ est utilisé par LetRec .

Conversion CFG

La conversion de l'AST' vers le CFG convertit une expression en une suite d'instructions et un ensemble de blocs.

$$\mathbb{E}_{ast'} \times \mathbb{V} \times \mathcal{P}(\mathbb{V}) \times \mathbb{I} \vdash_{\text{cfg}} \mathbb{I} \times \mathcal{P}(\mathbb{V}) \times (\mathbb{P} \mapsto (\mathbb{B} \times \mathbb{I}))$$

$$e \ v \ V \ i \vdash_{\text{cfg}} e' \ V_e \ B_e$$

- e est l'expression sous forme d'AST' à intégrer au CFG ;
- v est le nom de la variable dans lequel conserver le résultat de l'évaluation de e ;
- V est l'ensemble des variables libres (arguments) de i devant être sauvegardées durant l'évaluation de e pour être ré restaurées après (ne doit pas contenir v).
- i est l'expression déjà transpilée au format CFG qui sera exécutée après e . Elle est supposée faire usage de v et chaque variable libre qui y apparaît doit figurer dans V ;
- e' est e transpilée ;
- V_e est l'ensemble des variables libres apparaissant dans e qui ne proviennent pas de V . En théorie, les variables libres de e' sont exactement $V \cup V_e$. À l'origine V_e contenait toutes les variables libres apparaissant dans e' de la même manière que V contient toutes les variables libres de i mais j'ai amendé cela pour simplifier l'implémentation. Cette nouvelle version est néanmoins bancal (e' peut ne pas contenir i mais un appel vers un bloc contenant i et il est impossible de le déduire d'après V_e) c'est pour cela que j'ai prévu de revenir à l'ancienne version quand j'aurai la garantie qu'elle est toujours compatible ;
- B_e est l'ensemble des blocs générés par la transpilation de e .

Par la suite :

- \bar{e} correspond à l'identifiant unique (variable) attribué au résultat de l'évaluation de e .
- \dot{e} correspond à l'identifiant de code unique (pointeur) attribué au bloc qui contiendra e .

$$\overline{(\text{Int } i) \ v \ V \ \epsilon \vdash_{\text{cfg}} (\text{Let } v \ (\text{Int } i) \ \epsilon) \ \emptyset \ \emptyset} \quad (\text{Int})$$

$$\overline{(\text{Var } x) \ v \ V \ \epsilon \vdash_{\text{cfg}} (\text{Let } v \ x \ \epsilon) \ \{x\} \ \emptyset} \quad (\text{Var})$$

$$\frac{e_2 \ v \ V \ \epsilon \vdash_{\text{cfg}} e'_2 \ V_{e_2} \ B_{e_2} \quad V_3 = V_{e_2} \setminus \{x\} \quad e_1 \ x \ (V \cup V_3) \ e'_1 \vdash_{\text{cfg}} e'_1 \ V_{e_1} \ B_{e_1}}{(\text{Let } x \ e_1 \ e_2) \ v \ V \ \epsilon \vdash_{\text{cfg}} e'_1 \ (V_3 \cup V_{e_1}) \ (B_{e_1} \sqcup B_{e_2})} \quad (\text{Let})$$

$$\frac{e_2 \bar{e}_2 (V \cup \{\bar{e}_1\}) (v = \bar{e}_1 \diamond \bar{e}_2; \epsilon) \vdash_{\text{cfg}} e'_2 V_{e_2} B_{e_2} \quad e_1 \bar{e}_1 (V_{e_2} \cup V) e'_2 \vdash_{\text{cfg}} e'_1 V_{e_1} B_{e_1}}{(\text{Binary } \diamond e_1 e_2) v V \epsilon \vdash_{\text{cfg}} e'_1 (V_{e_1} \cup V_{e_2}) (B_{e_1} \sqcup B_{e_2})} \quad (\text{Binary})$$

$$\frac{e \bar{e} \emptyset (\text{Return } \bar{e}) \vdash_{\text{cfg}} e' V_e B_e \quad V_2 = V_e \setminus \{x\}}{(\text{Fun } x e) v V \epsilon \vdash_{\text{cfg}} (\text{Let } v (\text{Closure } \dot{e} V_2) \epsilon) V_2 (B_e \sqcup \{\dot{e} = \text{Clos}(x) V_2 e'\})} \quad (\text{Fun})$$

$$\frac{e_2 \bar{e}_2 V (\text{Ifreturn } \dot{e}_1 \bar{e}_2 V) \vdash_{\text{cfg}} e'_2 V_{e_2} B_{e_2} \quad e_3 \bar{e}_3 V (\text{Ifreturn } \dot{e}_1 \bar{e}_3 V) \vdash_{\text{cfg}} e'_3 V_{e_3} B_{e_3} \quad e_1 \bar{e}_1 (V \cup V_{e_2} \cup V_{e_3}) (\text{If } \bar{e}_1 \dot{e}_2 V_{e_2} \dot{e}_3 V_{e_3} V) \vdash_{\text{cfg}} e'_1 V_{e_1} B_{e_1}}{B_{e_1 e_2 e_3} = \{\dot{e}_1 = \text{Ifjoin } v V \epsilon, \dot{e}_2 = \text{Ifbranch } V_{e_2} V e'_2, \dot{e}_3 = \text{Ifbranch } V_{e_3} V e'_3\}} \quad (\text{If})$$

$$\frac{e_2 \bar{e}_2 (V \cup \{\bar{e}_1\}) (\text{Call } \bar{e}_1 (\bar{e}_2) \dot{e} V) \vdash_{\text{cfg}} \epsilon_2 V_{e_2} B_{e_2} \quad e_1 \bar{e}_1 (V_{e_2} \cup V) \epsilon_2 \vdash_{\text{cfg}} \epsilon_1 V_{e_1} B_{e_1}}{(\text{App } e_1 e_2) v V \epsilon \vdash_{\text{cfg}} \epsilon_2 (V_{e_1} \cup V_{e_2}) (B_{e_1} \sqcup B_{e_2} \sqcup \{\dot{e} = \text{Return } v V \epsilon\})} \quad (\text{App})$$

$$\frac{\begin{cases} \epsilon_0, V_0, B_0 = \text{Let } v (\text{Constructor } t (\bar{a}_i)_{i=1}^{i=n}) \epsilon, \{\bar{a}_i \mid 1 \leq i \leq n\}, \emptyset \\ \epsilon_n, V_n, B_n = \epsilon, V \cup V_{n-1}, B \cup B_{n-1} \text{ si } a_n \bar{a}_n (V \cup V_{n-1} \setminus \{\bar{a}_n\}) \epsilon_{n-1} \vdash_{\text{cfg}} \epsilon V B \end{cases}}{(\text{Constructor } t (a_i)_{i=1}^{i=n}) v V \epsilon \vdash_{\text{cfg}} \epsilon_n V_n B_n} \quad (\text{Constructor})$$

$$\frac{d \bar{d} V (\text{Matchreturn } \dot{e} \bar{d} V) \vdash_{\text{cfg}} \epsilon_d V_d B_d \quad \begin{cases} B_0 = \emptyset \\ V_{e_n}, B_n = V, B \cup B_{n-1} \cup \{\dot{e}_n, \text{Matchbranch } (\bar{a}_n^i)_{i=1}^{i=m_n} V_{e_n} V, \epsilon\} \\ \text{si } e_n \bar{e}_n V (\text{Matchreturn } \dot{e} \bar{e}_n V) \vdash_{\text{cfg}} \epsilon V B \\ V_{e_n} = V_{e_n} \setminus \{a_n^1, \dots, a_n^{m_n}\} \end{cases} \quad \frac{V_{e_n} = e_n \bar{e}_n V (\text{Matchreturn } \dot{e} \bar{e}_n V) \vdash_{\text{cfg}} \epsilon V B}{V \setminus \{a_n^1, \dots, a_n^{m_n}\}}}{e \bar{e} \left(\bigcup_{i=1}^{i=n} V_{e_i} \cup V_d \cup V \right) \left(\text{Matchpattern } \bar{e} \left(t_i, \dot{e}_i, (\bar{a}_i^j)_{j=1}^{j=m_i}, V_{e_i} \right)_{i=1}^{i=n} \langle \dot{d}, V_d \rangle V \right) \vdash_{\text{cfg}} \epsilon_1 V_1 B_1} \quad (\text{Match})$$

Nettoyage des alias

Le code CFG ainsi généré est susceptible de contenir de nombreux alias de variables (créés par la règle (Var)) et cela peut nuire à la qualité de l'analyse. La passe de nettoyage supprime tous les alias. Il est plus simple d'éliminer les alias en 2 passes que directement lors de la génération même si cela permettrait de se passer de l'expression Var dans la représentation intermédiaire.

Taille

Déterminer la taille du CFG est nécessaire pour certaines heuristiques. Comme le CFG n'est pas le langage qui sera compilé ou exécuté, le calcul de cette taille est seulement indicatif et n'est en aucun cas précis. La taille du CFG correspond à la somme de la taille de tous ses blocs. La taille d'un bloc correspond au nombre de ses paramètres plus la somme de la taille de ses instructions. La taille d'une instruction dépend principalement du nombre de ses arguments.

Recensement des variables et appels de blocs

Vivacité $<>$ Compte > 0

Plusieurs optimisations ont besoin de connaître le nombre de fois qu'une variable est utilisée ou qu'un bloc est appelé. Un tel algorithme de recensement est trivial mais en fait ne l'est pas du tout. Mon implémentation actuelle ne prend pas en compte la vivacité des blocs, c'est pour cela qu'un bloc appelé par un bloc mort sera considéré vivant. De plus, étant donné qu'il est impossible de déterminer sans une analyse complète quel bloc sera appelé par un appel indirect, je me contente de supposer un appel à chaque création de fermeture. Ces surapproximations doivent être corrigées, mais cela ne peut pas se faire sans une approche globale des optimisations qui suivent, en particulier le nettoyage des variables inutilisées et des blocs morts.

Spécialisation

La spécialisation consiste à copier des blocs. Les appels directs vers les blocs concernés reçoivent un nouveau pointeur vers un bloc fraîchement copié. La copie d'un bloc a pour effet immédiat d'améliorer la précision de l'analyse, et c'est également la première étape de l'inlining.

Algorithme de spécialisation

L'implémentation actuelle de la spécialisation des blocs n'est pas poussée à son maximum. Les blocs à copier sont toujours copiés indépendamment de l'appel. Il serait intéressant d'intégrer la possibilité de choisir les appels à spécialiser, autrement dit de traiter des couples bloc appelant/bloc appelé. Néanmoins un tel couple me paraîtrait insuffisant notamment lorsqu'un même bloc peut être appelé de différentes manières par une même instruction (en théorie c'est le cas du filtrage par motif même si mon implémentation actuelle ne le permet

pas). Une solution possible serait de choisir dynamiquement lors de l'analyse de spécialiser tel ou tel appel. Cette solution apporterait certes des réponses mais apporterait probablement de trop nombreux problèmes.

Lors de la copie d'un bloc il est évidemment indispensable de conserver les invariants qui s'appliquent au CFG, en particulier l'unicité des noms de variable, c'est pour cela que chaque copie s'accompagne d'une nouvelle étape de renommage.

Choix des blocs à spécialiser

Les blocs à spécialiser sont actuellement sélectionnés uniquement selon que leur taille se trouve ou non sous un certain seuil statique. C'est en particulier sur ces choix que je vais être amené à trouver des heuristiques pertinentes d'ici la fin du stage. De telles heuristiques devraient prendre en compte les gains et coûts potentiels apportés par la spécialisation en se basant sur les informations issues de l'analyse.

Analyse de valeurs par interprétation abstraite

L'analyse est l'étape la plus compliquée et probablement la plus importante pour permettre d'inliner de manière efficace. L'objectif principal est de transformer au mieux les sauts indirects (appels de fonctions) en sauts directs afin d'être capable d'inliner de tels sauts. Ensuite, même si ce n'est pas obligatoire, il est intéressant de disposer d'une analyse des valeurs assez précise pour se faire une idée de quand inliner pour obtenir les meilleurs bénéfices. Cette analyse s'effectue au niveau du CFG afin d'exploiter la sémantique du langage (en conservant certaines relations) tout en disposant des informations nécessaires sur les blocs.

Une contrainte importante portée sur l'analyse est la nécessité de pouvoir réaliser plusieurs analyses consécutives afin de pouvoir comparer les performances et résultats d'une seule analyse en profondeur face à plusieurs petites analyses. Cela implique que le langage intermédiaire sur lequel s'effectue l'analyse (en l'occurrence ici le CFG) doit rester le même après analyse. C'est pour cette raison que les résultats éventuels de l'analyse, en particulier les sauts directs, sont intégrés au CFG.

Points d'allocation

Sur conseil de mon tuteur, je réalise une analyse par point d'allocation. Ce choix donne des garanties de terminaison (le nombre de point d'allocation est borné par la taille du programme) tout en permettant une analyse poussée qui autorise par exemple la récursivité lors de la construction des blocs (fondamental pour traiter les listes). Il y a néanmoins certaines limitations à utiliser une telle analyse. La première que j'ai rencontrée était due à la présence de nombreux alias de variables présents dans le code CFG généré ce qui réduisait grandement la précision de l'analyse. Problème corrigé en effectuant une passe de nettoyage des alias avant chaque tour d'analyse.

Point d'allocation Un point d'allocation correspond simplement à une déclaration, c'est à dire une instruction `Let`. Étant donné que chaque valeur créée est déclarée (il n'existe pas de

valeur temporaire) avec un nom de variable unique (garanti par l'alpha-conversion), un point d'allocation peut donc être identifié par le nom de variable utilisé par Let.

Usine L'usine correspond à une table associant chaque point d'allocation aux valeurs qui y ont été produites. Par valeurs il faut comprendre valeur abstraite qui sera définie plus loin.

$\mathbb{U} := \mathbb{V} \mapsto \mathbb{D}$ (usine)

Paramètres Chaque paramètre de bloc est identifié par un ensemble de point d'allocation correspondant aux endroits d'où peut avoir été déclarée et initialisée sa valeur.

$\mathbb{P} := \mathcal{P}(\mathbb{V})$ (paramètre)

Domaines

Actuellement seulement trois domaines abstraits sont nécessaires pour représenter toutes les valeurs du langage.

Entiers Les entiers sont représentés de la manière la plus simple qui soit, c'est à dire des singletons munis de Top.

Top : $\mathbb{I}(\mathbb{Z})$

Singleton : $\mathbb{Z} \rightarrow \mathbb{I}(\{i\})$

L'union de deux entiers donne toujours Top sauf lorsqu'il s'agit de deux singletons de même valeur.

$$x \sqcup y = \begin{cases} \{i\} & \text{si } x = y = \{i\} \\ \mathbb{Z} & \text{sinon} \end{cases}$$

Fermetures Le domaine pour les fermetures est un environnement d'identifiant vers contexte, où l'identifiant correspond au pointeur de fonction, et le contexte correspond aux variables libres. Étant donné que les pointeurs de fonctions ainsi que les contextes (ensemble de points d'allocations) sont des ensembles bornés par la taille du programme, l'union de deux fermetures est garantie de converger.

$\mathbb{F} := \mathbb{P} \rightarrow \mathcal{P}(\mathcal{P}(\mathbb{V}))$ (fermeture)

L'union de deux fermetures consiste à conserver les entrées distinctes et d'unir les points d'allocations des entrées communes. Deux entrées communes, c'est à dire ayant comme clé le même pointeur, sont censées avoir le même environnement, dans le cas contraire il s'agit d'une erreur d'implémentation.

$$x \sqcup y = \forall z \in \mathcal{D}(x) \cup \mathcal{D}(y), \begin{cases} \{x(z)(i) \cup y(z)(i), i \in x(z) \text{ et } i \in y(z)\} & \text{si } z \in \mathcal{D}(x) \text{ et } z \in \mathcal{D}(y) \\ x(z) & \text{si } z \in \mathcal{D}(x) \\ y(z) & \text{si } z \in \mathcal{D}(y) \end{cases}$$

Unions taggées Le domaine pour les unions taggées est un environnement d'identifiant vers contexte, où l'identifiant correspond au tag, et le contexte correspond au contenu de l'union. Étant donné que les tags ainsi que les contextes (ensemble de points d'allocations) sont des ensembles bornés par la taille du programme, l'union de deux unions taggées est garantie de converger.

$\mathbb{C} := \mathbb{T} \rightarrow \mathcal{P}(\mathbb{V})^*$ (union taggée)

L'union de deux valeurs taggées consiste à conserver les entrées distinctes et d'unir les points d'allocations des entrées communes. Deux entrées communes, c'est à dire ayant comme clé le même tag, sont censées avoir le même contenu, dans le cas contraire il s'agit d'une erreur d'implémentation.

$$x \sqcup y = \forall z \in \mathcal{D}(x) \cup \mathcal{D}(y), \begin{cases} (x(z)_i \cup y(z)_i)_{i=1}^{i=n} & \text{si } z \in \mathcal{D}(x) \text{ et } z \in \mathcal{D}(y) \text{ et } |x(z)| = |y(z)| = n \\ x(z) & \text{si } z \in \mathcal{D}(x) \\ y(z) & \text{si } z \in \mathcal{D}(y) \end{cases}$$

Valeur abstraite

Une valeur abstraite est un entier, une fermeture ou une valeur taggée.

$\text{IntDomain} : \mathbb{I} \rightarrow \mathbb{A}$

$\text{ClosureDomain} : \mathbb{F} \rightarrow \mathbb{A}$

$\text{ConstructorDomain} : \mathbb{C} \rightarrow \mathbb{A}$

Blocs

Les blocs utilisés pour l'analyse sont les même que ceux du CFG à l'exception des paramètres qui sont désormais des ensembles de points d'allocation au lieu d'un nom de variables, c'est à dire \mathbb{B}_{cfg} avec $\mathbb{V} := \mathcal{P}(\mathbb{V})$. L'union de deux blocs (de même type) est l'union de leurs paramètres.

Contexte d'appel

Un contexte d'appel correspond à un étage de la pile, c'est à dire le pointer vers un bloc qui sera exécuté au prochain retour d'appel avec les paramètres qui ont été sauvegardés.

$\mathbb{F} := \mathbb{P} \times \mathcal{P}(\mathbb{V})^*$ (contexte d'appel)

Pile

La pile d'appel est une liste ordonnée d'appels.

$\mathbb{S} := \mathbb{F}^*$ (pile)

Bloc à analyser

Un bloc à analyser est identifié par son pointeur, ses arguments, la pile d'appel et enfin à l'usine du moment où il est appelé.

$$\mathbb{BB} := \mathbb{P} \times \text{cont}_{type} \times \mathbb{S} \times \mathbb{U}$$

Abstraction de la pile

Une fonction qui réduit la taille de la pile dont l'image est finie.

$$\text{stack}_{reduce} := \mathbb{S} \rightarrow \mathbb{S}$$

Contexte d'appel Le contexte d'appel contient la pile d'appel et les arguments d'appel.

Il est important de noter que si l'on suppose la pile d'appel comme ayant une taille finie (garantie après réduction), la dimension d'un contexte est bornée, ce qui est essentiel pour garantir la terminaison.

$$\text{context} := \mathbb{S} \times \text{cont}_{type}$$

Contextes des usines Une table des usines associe un contexte à une usine. La dimension d'un contexte étant bornée, les contextes sont dénombrables (on peut donc implémenter la table d'association correspondante).

$$\text{allocontexte} := \text{context} \rightarrow \mathbb{U}$$

Table de contextes des usines Une table de contextes de blocs associe chaque bloc à ses contextes d'allocation.

$$\text{blocontexte} := \text{pointer} \rightarrow \text{allocontexte}$$

Appel analysé Un appel analysé est une paire associant les paramètres passés avec leurs allocations correspondantes.

$$\text{callanalysed} := \text{cont}_{type} \times \text{allocations}$$

Programme analysé Un programme analysé (càd le résultat de l'analyse) est une table associant chaque bloc à son appel analysé.

$$\text{analysis} := \text{pointer} \rightarrow \text{callanalysed}$$

Algorithme d'analyse

$$\text{analysis} : \mathbb{BB}^* \times \text{stack}_{reduce} \times \text{blocks} \times \text{blocontexte} \rightarrow \text{analysis}$$

Algorithm 1: Analyse du programme

Input : B reduce prog map

```
1 if B est vide then
2   return map telle que pour chaque bloc sa valeur soit l'union des usines et paramètres
   de chaque contexte de pile de ce bloc.
3 else
4   k, block', stack'', allocations  $\leftarrow$  hd(B)
5   B'  $\leftarrow$  tl(B)
6   stack  $\leftarrow$  reduce(stack'')
7   if k  $\in \mathcal{D}(\textit{map})$  then
8     oldcontexts  $\leftarrow$  map(k)
9     if (stack, block')  $\in \mathcal{D}(\textit{oldcontexts})$  then
10      oldallocations  $\leftarrow$  oldcontexts((stack, block'))
11      newallocations  $\leftarrow$  oldallocations  $\cup$  allocations
12      if newallocations = oldallocations then
13        else
14          block, expr  $\leftarrow$  prog(k)
15          nextconts  $\leftarrow$  analysiscont(expr, stack'', (blockenv block block'),
            newallocations)
16          return analysis(B'nextconts, reduce, prog, add k (add (stack, block')
            newallocations oldcontexts) map)
17        end
18      else
19      end
20    else
21    end
22 end
```

Idées de terminaison

Les noms de variables, par extension les points d'allocations, étant en nombre fini dans le programme, l'ensemble identifiant les valeurs est également fini. De la même manière, un contexte d'appel (ensemble fini de valeurs correspondant aux arguments et pointeur de bloc), est un ensemble fini étant donné que le nombre de blocs dans le programme est également borné. Reste la question épineuse de comment garantir que la pile d'appels ne croît pas infiniment. Afin de tenter d'obtenir une précision maximale, je détecte d'éventuels motifs sur la pile en regardant si 1..N contextes d'appels se répètent et le cas échéant je supprime la répétition. Par exemple la pile d'appels $A::B::A::B::C::[]$ sera remplacée par $A::B::C::[]$. Après avoir implémenté cette méthode, mes tuteurs m'ont rapidement fait comprendre qu'elle ne pouvait pas garantir la terminaison. Pour la suite du stage je vais certainement devoir durcir la détection de motifs en passant à 0-CFA ou 1-CFA.

Propagation

La propagation modifie le CFG pour y faire apparaître les résultats de l'analyse. Avec le recul, il est fort probable que cette optimisation ne soit pas strictement nécessaire à cette étape de la compilation, mais se fasse plutôt lors de la conversion vers le CFG exécutable (c'est à dire générer le CFG exécutable en ayant sous la main les résultats de l'analyse au lieu de modifier le CFG sur lequel il existe des contraintes sémantique fortes).

Constantes Lorsque les paramètres sont connus, certaines expressions comme l'addition ou la soustraction sont transformées en constantes.

Branchements conditionnels Lorsque les valeurs possibles de la condition d'un if ou d'un filtrage par motif sont connues, certaines branches sont supprimées. Néanmoins pour respecter la sémantique des blocs, il n'est pas possible durant cette phase de transformer les branchements en appels directs lorsqu'il ne reste qu'une seule branche possible.

Appels indirects Les appels indirects sont transformés en appels directs lorsqu'il y a exactement 1 fermeture candidate.

CFG exécutable

L'objectif principal de ce langage intermédiaire est d'avoir une représentation bas-niveau stable et facile à interpréter sur laquelle effectuer des benchmarks. L'inlining a lieu sur le CFG exécutable car les modifications apportées peuvent casser la sémantique d'appel ce qui doit être représenté au niveau de la pile.

Langage

Le CFG exécutable est très similaire au CFG, si ce n'est que quasiment tous les traits de langage propres à OCaml ont été concrétisés. À chaque construction de valeur du langage OCaml est associée une structure de données, la plupart d'entre elles devenant des n-uplets. Tous les types de blocs fusionnent en un seul en fixant la sémantique des sauts (passage de l'environnement comme argument) et chaque type de branchement est transformé en un saut (direct ou indirect) avec la possibilité d'ajouter des contextes d'appel sur la pile (seule l'instruction d'appel ajoute un contexte lors de cette transformation).

Identifiants

On retrouve ici les identifiants du CFG auxquels s'ajoutent de nouveaux identifiants pour gérer les contextes de pile.

$\mathbb{V} := \mathbb{N}$ (variables)

$\mathbb{P} := \mathbb{N}$ (pointeurs de blocs)

$\mathbb{F} := \mathbb{P} \times \mathbb{V}^*$ (contexte d'appel)

$\mathbb{S} := \mathbb{F}^*$ (pile)

Expressions

De la même manière que pour les identifiants, on retrouve ici la plupart des expressions du CFG. Le principal changement est la transformation des constructeurs et fermetures qui deviennent des n-uplets.

$\text{Const} : \mathbb{Z} \mapsto \mathbb{E}$ génère un entier.

$\text{Pointer} : \mathbb{P} \mapsto \mathbb{E}$ génère un pointeur vers un bloc.

$\text{Var} : \mathbb{V} \mapsto \mathbb{E}$ crée un alias de variable.

$\text{Add} : \mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$ additionne deux entiers.

$\text{Sub} : \mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$ soustrait deux entiers.

$\text{Tuple} : \mathbb{V}^* \mapsto \mathbb{E}$ fabrique un n-uplet.

$\text{Get} : \mathbb{V} \times \mathbb{N} \mapsto \mathbb{E}$ accède à un champs d'un n-uplet.

Instructions

Les instructions sont assez similaires, si ce n'est que les branchements permettent désormais d'ajouter des contextes de piles. Le filtrage par motif et le if classique fusionnent en une seule instruction.

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{I} \mapsto \mathbb{I}$ assigne le résultat d'une expression à une variable.

ApplyDirect : $\mathbb{P} \times \mathbb{V}^* \times \mathbb{S} \mapsto \mathbb{I}$ réalise un saut direct vers le bloc associé.

ApplyIndirect : $\mathbb{V} \times \mathbb{V}^* \times \mathbb{S} \mapsto \mathbb{I}$ réalise un saut indirect au bloc associé au pointeur contenu dans la variable.

If : $\mathbb{V} \times (\mathbb{N} \times \mathbb{P} \times \mathbb{V}^*)^* \times (\mathbb{P} \times \mathbb{V}^*) \times \mathbb{S} \mapsto \mathbb{I}$ réalise un saut vers le bloc associé à la valeur de la condition ou vers le bloc par défaut.

Return : $\mathbb{V} \mapsto \mathbb{I}$ renvoie le résultat contenu dans la variable.

Bloc

Il n'existe plus de sémantique pour chaque type de bloc, l'ordre des arguments est maintenant fixé.

$\mathbb{B} := \mathbb{V}^* \times \mathbb{I}$ (bloc)

Génération du CFG exécutable

La transpilation d'un bloc CFG peut générer plusieurs blocs concrétisés.

$\mathbb{B}_{cfg} \times \mathbb{I} \vdash_{cfg} \mathbb{B} \times (\mathbb{P} \mapsto \mathbb{B})$

$a \ i \vdash_{cfg} a' \ i' \ B$

- b est l'entête du bloc à transpiler ;
- i est l'instruction du bloc à transpiler (son contenu) ;
- a' est l'entête transpilée (les arguments du bloc transpilé) ;
- i' est le contenu transpilé ;
- B est l'ensemble des blocs générés au passage.

Dans la suite, les ensembles de variables correspondant aux arguments des blocs sont considérés comme ordonnés (ordre lexicographique par exemple) de manière déterministe.

$$\overline{\left(\text{Cont } (a_i)_{i=0}^{i=n} \right) i \vdash_{cfg} (a_i)_{i=0}^{i=n} \ i \ \emptyset} \quad (\text{Cont})$$

$$\overline{\left(\text{Return } a_0 \ (a_i)_{i=1}^{i=n} \right) i \vdash_{cfg} (a_i)_{i=0}^{i=n} \ i \ \emptyset} \quad (\text{Return})$$

$$\frac{\begin{cases} i_0 = \text{ApplyDirect } p(a_{n+1}, \dots, a_m, a_1, \dots, a_n) () \\ i_n = (a_n = \text{Get } e \ n; i_{n-1}) \end{cases}}{(\text{Clos } (a_i)_{i=0}^{i=n} (a_i)_{i=n+1}^{i=m}) i \vdash_{\text{cfg}}, (e, a_{n+1}, \dots, a_m) i_n \{p = (a_{n+1}, \dots, a_m, a_1, \dots, a_n), i\}} \quad (\text{Clos})$$

$$\frac{}{(\text{IfBranch } (a_i)_{i=0}^{i=n} (a_i)_{i=n+1}^{i=m}) i \vdash_{\text{cfg}}, (a_i)_{i=0}^{i=m} i \emptyset} \quad (\text{IfBranch})$$

$$\frac{}{(\text{IfJoin } a_0 (a_i)_{i=1}^{i=n}) i \vdash_{\text{cfg}}, (a_i)_{i=0}^{i=n} i \emptyset} \quad (\text{IfJoin})$$

$$\frac{\begin{cases} i_0 = i \\ i_n = (a_n = \text{Get } e \ n; i_{n-1}) \end{cases}}{(\text{MatchBranch } (a_i)_{i=0}^{i=n} (a_i)_{i=n+1}^{i=m} (a_i)_{i=m+1}^{i=o}) i \vdash_{\text{cfg}}, (e, a_{n+1}, \dots, a_m, a_{m+1}, \dots, a_o) i_n \emptyset} \quad (\text{MatchBranch})$$

$$\frac{}{(\text{MatchJoin } a_0 (a_i)_{i=1}^{i=n}) i \vdash_{\text{cfg}}, (a_i)_{i=0}^{i=n} i \emptyset} \quad (\text{MatchJoin})$$

Taille

La taille du CFG exécutable est uniquement intéressante d'un point de vue performances et connaître l'impact des différentes optimisations. Le calcul se fait de la même manière que pour le CFG.

Recensement des variables et appels de blocs

De la même manière que lors de la phase CFG, les variables et les appels de blocs sont recensés afin de supprimer les variables inutilisées et les blocs morts.

Inlining

Inliner un bloc consiste à intégrer son contenu dans le bloc appelant à la place de la dernière instruction (branchement) lorsqu'il s'agit d'un appel direct uniquement. Chaque argument du bloc inliné est remplacé par la variable qui lui a été assignée lors du branchement par le bloc appelant. Pour l'instant seuls les appels directs peuvent être inlinés. Si lors de l'appel des contextes étaient empilés sur la pile, alors le branchement du bloc inliné en tiendra compte. En particulier, si le branchement du bloc inliné est un retour de fonction (et si la pile n'est pas vide) celui-ci dépilera la pile et deviendra un saut direct. Dans les autres cas les contextes du bloc appelant sont empilés sur les contextes du bloc appelé, ce qui permet d'avoir des sauts vers l'intérieur d'une fonction.

Choix des blocs à inliner

Sont actuellement inlinés tous les blocs appelés exactement 1 fois (les blocs spécialisés lors de l'analyse du flot de contrôle sont ainsi tous concernés).

Interprétation

C'est le CFG exécutable que j'interprète pour dans un premier temps m'assurer de la validité de toutes les transformations et dans un second temps réaliser des benchmarks.

Validité des phases de compilation

L'interprétation m'a permis de corriger de nombreuses erreurs à partir de divers tests focalisés. Néanmoins les tests en question sont très courts (moins d'une dizaine de lignes) et il est possible que d'autres erreurs apparaissent pour des programmes plus longs.

Benchmarks

Durant l'interprétation j'enregistre diverses informations intéressantes, comme le nombre de sauts, de variables lues ou écrites. Celles-ci sont affichées après exécution du programme pour identifier la pertinence de certaines optimisations.

Heuristiques d'inlining

Les phases de compilation étant prêtes et à mes yeux suffisamment expressives pour permettre toute forme d'inlining, je détaille ici quelques idées d'heuristiques pour la suite du stage. De nombreux tests complets seront à réaliser pour à la fois vérifier la pertinence des heuristiques implémentées mais également en détecter de nouvelles.

Blocs de petite taille

Inlining partiel

Une heuristique possible sur laquelle je vais me pencher sera l'inlining partiel, c'est à dire potentiellement inliner seulement les premiers blocs d'une fonction. La manière dont je représente le programme me permet de sauter à l'intérieur d'une fonction, ce qui n'est actuellement pas possible avec flambda2. La question est donc de savoir s'il existe des cas où cette heuristique pourrait être intéressante, que ce soit à la fois en terme d'optimisations de la taille ou du temps d'exécution. Une première idée de situations intéressantes qui me vient à l'esprit est le grand nombre de fonctions en OCaml qui filtrent au début un de leurs arguments ce qui, à ma connaissance, se représente une fois compilé comme quelques instructions suivies d'un saut. Le filtrage en lui même n'est pas une opération spécialement coûteuse en terme d'espace, on transformerait ici un appel de fonction en un filtrage vers des blocs à "l'intérieur" de celle-ci (avec évidemment les opérations sur la pile qu'il convient de faire comme un appel classique). De plus l'inliner peut permettre de gagner en informations sur le motif, ce qui peut rendre possible de transformer le filtrage en appel direct, les gains seraient considérables.

Un exemple qui me pousse à croire qu'une telle heuristique peut être prometteuse est fourni sur le site de flambda :

```
let f b x =  
  if b then x else ... big expression ...
```

```
let g x = f true x
```

En appliquant les transformations pour faire apparaître les blocs, les branchements et les manipulations de la pile on obtient ce pseudo-code sous forme de basic blocks (sans distinguer les appels directs des appels indirects) :

```
let rec f b x stack =  
  if b then f0 x stack else f1 stack  
and f0 x stack =  
  f2 x stack  
and f1 stack =  
  f2 (... big expression ...) stack  
and f2 x (p::stack) =  
  p x stack  
and g x stack =  
  f true x (g0::stack)
```

```
and g0 x (p::stack) =
  p x stack
```

Dans le cas présent la fonction `f` peut être vue comme un bloc d'une seule instruction. L'idée est donc d'inliner systématiquement ce bloc (pas la fonction complète, d'où la notion d'inlining partiel), et on obtient le code suivant où l'appel vers `f` est inliné dans `g` :

```
let rec f b x stack =
  if b then f0 x stack else f1 stack
and f0 x stack =
  f2 x stack
and f1 stack =
  f2 (... big expression ...) stack
and f2 x (p::stack) =
  p x stack
and g x stack =
  if true then f0 x (g0::stack) else f1 (g0::stack)
and g0 x (p::stack) =
  p x stack
```

Cet inlining n'est possible que si toutes les branches du `if` empilent les mêmes contextes sur la pile, ce qui est normalement toujours le cas quand le code est bien construit. Avec cet inlining, indépendamment de la valeur de `b`, on économise un saut et on se donne davantage d'informations sur le contexte dans le bloc inliné qui pourraient être utiles à d'éventuelles simplifications. Après simplifications, en supposant que le premier bloc de `f` est systématiquement inliné pour chaque appel et en inlinant les appels terminaux, on obtient le code suivant :

```
let rec f0 x (p::stack) =
  p x stack
and f1 (p::stack) =
  p (... big expression ...) stack
and g x (p::stack) =
  p x stack
```

Évidemment au stade actuel cette heuristique ne reste qu'une hypothèse et il faudra vérifier à la fois sa faisabilité et son efficacité. Il est par ailleurs fort probable qu'elle ne soit pas possible à mettre en place dans l'évaluateur de bytecode.

Conclusion

Ce stage m’a permis de mettre en pratique un grand nombre de savoirs acquis lors du Master pour réaliser les grandes étapes d’un compilateur. J’ai à de nombreuses reprises été amené à faire des choix pas toujours évidents et l’expérience de mes responsables de stage m’a grandement aidé. Avec le recul les avantages et inconvénients du choix d’évoluer sur un langage “jouet” au lieu du compilateur officiel sont apparues plus claires à mes yeux. En particulier, ayant côtoyé durant mon stage d’autres stagiaires STL s’impliquant en équipe sur des projets concrets, je peux regretter d’avoir été amené à travailler en totale autonomie et ne pas avoir pu exploiter le stage pour me former en profondeur au monde réel de l’entreprise. Néanmoins, le temps qu’aurait été nécessaire pour comprendre le fonctionnement de ce compilateur complexe et prenant en compte ce que j’ai appris en partant de zéro fait que ce choix en valait la peine.