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 amener à étudier en détails d'ici la fin du stage.

Table des matières

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.

$\mathbb{S} \coloneqq \mathit{string}$ (chaînes de caractères)

$\mathbb{V} \coloneqq \mathbb{S}$ (variables)

$\mathbb{T} \coloneqq \mathbb{S}$ (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.

Binop : $\mathbb{B} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ 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 transformants 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.

$\mathbb{V} \coloneqq \mathbb{N}$ (variables)

$\mathbb{T} \coloneqq \mathbb{N}$ (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.

Binop : $\mathbb{B} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$ 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.

$\mathbb{E}_{ast} \times (\mathbb{V}_{ast} \mapsto \mathbb{V}) \times (\mathbb{T}_{ast} \mapsto \mathbb{T}) \vdash_{\text{ast}'} \mathbb{E} \times (\mathbb{V} \mapsto \mathbb{V}_{ast}) \times (\mathbb{V}_{ast} \mapsto \mathbb{V})$

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

- 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 $\overline{x}$ l'identifiant unique donné à une variable x, e' l'expression générée par l'expression e, $\emptyset$ une table vide, $x \coloneqq y$ 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.

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

$$\frac{\begin{array}{c} e_1\ A\ C \vdash_{\text{ast}'} e_1'\ S_{e_1}\ L_{e_1} \\ e_2\ (A \sqcup L_{e_1})\ C \vdash_{\text{ast}'} e_2'\ S_{e_2}\ L_{e_2} \end{array}}{(\text{Binop}\ \diamond\ e_1\ e_2)\ A\ C \vdash_{\text{ast}'} (\text{Binop}\ \diamond\ e_1'\ e_2')\ (S_{e_1} \sqcup S_{e_2})\ (L_{e_1} \sqcup L_{e_1})} \tag{Binop}$$

$$\frac{e\ (A \sqcup \{x \coloneqq \overline{x}\})\ C \vdash_{\text{ast}'} e'\ S_e\ L_e}{(\text{Fun}\ x\ e)\ A\ C \vdash_{\text{ast}'} (\text{Fun}\ \overline{x}\ e')\ (S_e \sqcup \{\overline{x} \coloneqq x\})\ L_e} \tag{Fun}$$

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

$$\frac{x \notin \mathcal{D}(A)}{(\text{Var}\ x)\ A\ C \vdash_{\text{ast}'} (\text{Var}\ \overline{x})\ \emptyset\ \{x \coloneqq \overline{x}\}} \tag{Var2}$$

$$\frac{\begin{array}{r} e_1\ A\ C \vdash_{\text{ast}}, e_1'\ S_{e_1}\ L_{e_1} \\ e_2\,(A \sqcup L_{e_1} \sqcup \{x \coloneqq \overline{x}\})\ C \vdash_{\text{ast}}, e_2'\ S_{e_2}\ L_{e_2} \end{array}}{(\text{Let}\ x\ e_1\ e_2)\ A\ C \vdash_{\text{ast}}, (\text{Let}\ \overline{x}\ e_1'\ e_2')\,(S_{e_1} \sqcup S_{e_2} \sqcup \{\overline{x} \coloneqq x\})\,(L_{e_1} \sqcup L_{e_2})} \quad \text{(Let)}$$

$$\frac{\begin{array}{r} e_1\ A\ C \vdash_{\text{ast}}, e_1'\ S_{e_1}\ L_{e_1} \\ e_2\,(A \sqcup L_{e_1})\ C \vdash_{\text{ast}}, e_2'\ S_{e_2}\ L_{e_2} \\ e_3\,(A \sqcup L_{e_1} \sqcup L_{e_2})\ C \vdash_{\text{ast}}, e_3'\ S_{e_3}\ L_{e_3} \end{array}}{(\text{If}\ e_1\ e_2\ e_3)\ A\ C \vdash_{\text{ast}}, (\text{Binop}\ e_1'\ e_2'\ e_3')\,(S_{e_1} \sqcup S_{e_2} \sqcup S_{e_3})\,(L_{e_1} \sqcup L_{e_1} \sqcup L_{e_3})} \quad \text{(If)}$$

$$\frac{\begin{array}{r} e_1\ A\ C \vdash_{\text{ast}}, e_1'\ S_{e_1}\ L_{e_1} \\ e_2\,(A \sqcup L_{e_1})\ C \vdash_{\text{ast}}, e_2'\ S_{e_2}\ L_{e_2} \end{array}}{(\text{App}\ e_1\ e_2)\ A\ C \vdash_{\text{ast}}, (\text{App}\ e_1'\ e_2')\,(S_{e_1} \sqcup S_{e_2})\,(L_{e_1} \sqcup L_{e_2})} \quad \text{(App)}$$

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

$$\frac{\begin{array}{r} e_1\ A\ C \vdash_{\text{ast}}, e_1'\ S_{e_1}\ L_{e_1} \\ \ldots \\ e_n\left(\bigsqcup_{i=1}^{i=n-1} L_{e_{i-1}} \sqcup A\right) C \vdash_{\text{ast}}, e_n'\ S_{e_n}\ L_{e_n} \end{array}}{\left(\text{Constructor}\ s\,(e_i)_{i=1}^{i=n}\right)\ A\ C \vdash_{\text{ast}}, \left(\text{Constructor}\,(C(s))\,(e_i')_{i=1}^{i=n}\right)\left(\bigsqcup_{i=1}^{i=n} S_{e_i}\right)\left(\bigsqcup_{i=1}^{i=n} L_{e_i}\right)} \quad \text{(Constructor)}$$

$$\frac{\begin{array}{r} e_1\,(\{x_i \coloneqq x_i', i \in 1 \ldots n\} \sqcup A)\ C \vdash_{\text{ast}}, e_1'\ S_{e_1}\ L_{e_1} \\ \ldots \\ e_n\left(\bigsqcup_{i=1}^{i=n-1} S_{e_{i-1}} \sqcup \{x_i \coloneqq x_i', i \in 1 \ldots n\} \sqcup A\right) C \vdash_{\text{ast}}, e_n'\ S_{e_n}\ L_{e_n} \end{array}}{\left(\text{LetRec}\,(x_i, e_i)_{i=1}^{i=n}\ e\right)\ A\ C \vdash_{\text{ast}}, \left(\text{LetRec}\,(x_i', e_i')_{i=1}^{i=n}\ e'\right)\left(\bigsqcup_{i=1}^{i=n} S_{e_i}\right)\left(\bigsqcup_{i=1}^{i=n} L_{e_i}\right)} \quad \text{(LetRec)}$$

$$\frac{(\textit{todo})}{\left(\text{Match}\ e\,(m_i, e_i)_{i=1}^{i=n}\right)\ A\ C \vdash_{\text{ast}}, \left(\text{Match}\ e'\left(t_i', \left(a_i^j\right)_{j=1}^{j=m_i}, e_i'\right)_{i=1}^{i=n}\right)\left(\bigsqcup_{i=1}^{i=n} S_{e_i} \sqcup S_e\right)\left(\bigsqcup_{i=1}^{i=n} L_{e_i} \sqcup L_e\right)} \quad \text{(Match)}$$

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.

$\mathbb{V} \coloneqq \mathbb{N}$ (variables)

$\mathbb{T} \coloneqq \mathbb{N}$ (tags)

$\mathbb{P} \coloneqq \mathbb{N}$ (pointeurs)

Expressions

Les expressions correspondent aux instructions élémentaires qui construisent des valeurs.

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

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

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

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

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

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.

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

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'éxécution avec le contexte spécifié.

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').

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.

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.

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

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.

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.

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).

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

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

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.

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

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.

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.

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 réstaurées après (ne doit pas contenir v).
- i est l'expression déjà transpilée au format CFG qui sera éxécutée 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'origne 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 bancale (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 :

- $\overline{e}$ 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.

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

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

$$\frac{\begin{array}{r} e_2\ v\ V\ \epsilon \vdash_{\text{cfg}} e_2'\ V_{e_2}\ B_{e_2} \quad V_3 = V_{e_2} \setminus \{x\} \\ e_1\ x\ (V \cup V_3)\ e_2' \vdash_{\text{cfg}} e_1'\ V_{e_1}\ B_{e_1} \end{array}}{(\text{Let}\ x\ e_1\ e_2)\ v\ V\ \epsilon \vdash_{\text{cfg}} e_1'\ (V_3 \cup V_{e_1})\ (B_{e_1} \sqcup B_{e_2})} \quad \text{(Let)}$$

$$\frac{\begin{array}{r} e_2\ \overline{e_2}\,(V \cup \{\overline{e_1}\})\,(v = \overline{e_1} \diamond \overline{e_2}; \epsilon) \vdash_{\text{cfg}} e_2'\ V_{e_2}\ B_{e_2} \\ e_1\ \overline{e_1}\ (V_{e_2} \cup V)\ e_2' \vdash_{\text{cfg}} e_1'\ V_{e_1}\ B_{e_1} \end{array}}{(\text{Binop}\ \diamond\ e_1\ e_2)\ v\ V\ \epsilon \vdash_{\text{cfg}} e_1'\,(V_{e_1} \cup V_{e_2})\,(B_{e_1} \sqcup B_{e_2})} \tag{Binop}$$

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

$$\frac{\begin{array}{r} e_2\ \overline{e_2}\ V\ (\text{Ifreturn}\ \dot{e_1}\ \overline{e_2}\ V) \vdash_{\text{cfg}} e_2'\ V_{e_2}\ B_{e_2} \\ e_3\ \overline{e_3}\ V\ (\text{Ifreturn}\ \dot{e_1}\ \overline{e_3}\ V) \vdash_{\text{cfg}} e_3'\ V_{e_3}\ B_{e_3} \\ e_1\ \overline{e_1}\ (V \cup V_{e_2} \cup V_{e_3})\ (\text{If}\ \overline{e_1}\ \dot{e_2}\ V_{e_2}\ \dot{e_3}\ V_{e_3}\ V) \vdash_{\text{cfg}} e_1'\ V_{e_1}\ B_{e_1} \\ B_{e_1e_2e_3} = \{\dot{e_1} = \text{Ifjoin}\ v\ V\ \epsilon, \dot{e_2} = \text{Ifbranch}\ V_{e_2}\ V\ e_2', \dot{e_3} = \text{Ifbranch}\ V_{e_3}\ V\ e_3'\} \end{array}}{(\text{If}\ e_1\ e_2\ e_3)\ v\ V\ \epsilon \vdash_{\text{cfg}} e_1'\ (V_{e_1} \cup V_{e_2} \cup V_{e_3})\,(B_{e_1} \sqcup B_{e_2} \sqcup B_{e_3} \sqcup B_{e_1e_2e_3})} \tag{If}$$

$$\frac{\begin{array}{r} e_2\ \overline{e_2}\,(V \cup \{\overline{e_1}\})\,(\text{Call}\ \overline{e_1}\,(\overline{e_2})\,\dot{e}\ V) \vdash_{\text{cfg}} \epsilon_2\ V_{e_2}\ B_{e_2} \\ e_1\ \overline{e_1}\,(V_{e_2} \cup V)\,\epsilon_2 \vdash_{\text{cfg}} \epsilon_1\ V_{e_1}\ B_{e_1} \end{array}}{(\text{App}\ e_1\ e_2)\ v\ V\ \epsilon \vdash_{\text{cfg}} \epsilon_2\,(V_{e_1} \cup V_{e_2})\,(B_{e_1} \sqcup B_{e_2} \sqcup \{\dot{e} = \text{Return}\ v\ V\ \epsilon\})} \tag{App}$$

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

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

$$\overline{(\text{Match}\ e\left(t_i, \left(a_i^j\right)_{j=1}^{j=m_i}, e_i\right)_{i=1}^{i=n} d)\ v\ V\ \epsilon \vdash_{\text{cfg}} (todo)}$$

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

Plusieurs optimisations ont besoin de connaître la vivacité des variables et blocs. L'approche naïve que j'ai implémentée consiste simplement à compter leurs occurences. Cela permet d'éliminer un nombre conséquant de variables et blocs morts avec la garantie de ne jamais éliminer quoi que ce soit de vivant mais ne permet pas d'éliminer tout ce qui est réellement mort. Une analyser plus efficace est souhaitable mais difficile à implémenter pour le peu de bénéfice que cela apporterait.

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. Néammoins 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 si 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'occurence 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 dûe à la présence de nombreux alias de variables présents dans le code CFG généré ce qui réduisait grandement la précision de l'analyse en fusionnant certaines valeurs au lieu de les garder séparées. 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 du CFG. Étant donné que chaque valeur créée est déclarée (il n'existe pas de valeur temporaire) avec un nom de variable unique (garanti par le rafraîchissement), un point d'allocation peut donc être identifé 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.

$\mathbb{U} \coloneqq \mathbb{V} \mapsto \mathbb{A}$ (usine)

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 : $\mathbb{Z} \to \mathbb{I}$ ($\{i\}$)

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.

$\mathbb{F} \coloneqq \mathbb{P} \to \mathcal{P}(\mathcal{P}(\mathbb{V}))$ (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.

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

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.

$\mathbb{C} \coloneqq \mathbb{T} \mapsto \mathcal{P}(\mathbb{V})^*$ (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.

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

Valeur abstraite

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

IntDomain : $\mathbb{I} \mapsto \mathbb{A}$

ClosureDomain : $\mathbb{F} \mapsto \mathbb{A}$

ConstructorDomain : $\mathbb{C} \mapsto \mathbb{A}$

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 $\mathbb{V} \coloneqq \mathcal{P}(\mathbb{V})$. L'union de deux blocs (de même type) est l'union de leurs paramètres.

Abstraction de la pile

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. La pile d'appel est une liste ordonnée de contextes d'appel.

$\mathbb{S} \coloneqq (\mathbb{P} \times \mathcal{P}(\mathbb{V})^*)^*$ (pile)

Pour assure la terminaison de l'analyse, il est nécessaire de disposer d'une fonction qui réduit la taille de la pile dont l'image est finie. Son intérêt est de garantir la terminaison de l'analyse en bornant le nombre de piles possibles par rapport à la taille du programme.

Détection de motif Afin de tenter d'obtenir une précision maximale, je détecte d'éventuels motifs sur la pile en regardant si des contextes d'appels se répètent et le cas échéant je supprime la répétition. Par exemple la pile d'appels `ABCBC` a un motif `BC` de taille 2 et sera remplacée par `ABC`. Néanmoins mes tuteurs m'ont fait comprendre qu'elle ne pouvait pas garantir la terminaison, certains piles peuvent croître indéfiniment sans jamais appercevoir de motif.

n-CFA n-CFA consiste à conserver les n derniers contextes de la pile. Par exemple en 1-CFA, la pile d'appels `ABCBC` sera remplacée par `C`.

Algorithme d'analyse

L'algorithme d'analyse prend une liste des blocs à analyser (pointeur, en-tête, pile et usine), une fonction d'abstraction de la pile, l'ensemble des blocs du programme, l'ensemble des usines pour chaque contexte de chaque bloc déjà analysé et renvoie en-tête et usine pour chaque bloc.

analyse : $(\mathbb{P} \times \overline{\mathbb{B}} \times \mathbb{S} \times \mathbb{U})^* \times (\mathbb{S} \mapsto \mathbb{S}) \times (\mathbb{P} \mapsto \mathbb{B}) \times (\mathbb{P} \mapsto ((\mathbb{S} \times \overline{\mathbb{B}}) \mapsto \mathbb{U})) \mapsto (\mathbb{P} \mapsto (\overline{\mathbb{B}} \times \mathbb{U}))$

Algorithm 1: Analyse du programme

Input : l pile_abs B U

```
// Il ne reste plus aucun bloc à analyser.
if l est vide then
    return U telle que pour chaque bloc sa valeur soit l'union des usines et paramètres de chaque contexte de pile de ce bloc. // Tous les contextes de pile sont fusionnés.
else
    p, b, s, u ← hd(l) // Le premier bloc à analyser.
    l' ← tl(l) // Les autres blocs à analyser.
    s̄ ← pile_abs(s) // Abstraction de la pile.
    c ← (s̄, b) // Le contexte (pile et en-tête).
    // Ce bloc a déjà été analysé.
    if p ∈ D(U) then
        U_p ← U[p] // Les usines associées à ce bloc.
        // Ce contexte a déjà été analysé pour ce bloc.
        if c ∈ D(U_p) then
            u_c ← U_p[c] // L'usine associée à ce contexte.
            u_2 ← u_c ∪ u // Union de l'ancienne et la nouvelle usine.
            // L'ancienne usine et la nouvelle ont convergé.
            if u_2 = u_c then
                // La pile d'appels est vide.
                if s est vide then
                    return analyse(l', pile_abs, B, U) // On continue avec les prochains blocs.
                else
                    p_2, args ← hd(s) // Le contexte vers lequel renvoyer.
                    s_3 ← tl(s) // Les autres blocs à analyser.
                    b_2 ← Return(∅, args) // Le bloc de retour recevra Bottom.
                    return analyse((p_2, b_2, s_3, u_2) :: l', pile_abs, B, U) // On analysera le bloc appelant avec Bottom comme résultat.
                end
            else
                b_2, i ← B(p) // L'en-tête et le corps de ce bloc.
                l_2 ← analyse_de_bloc(i, s, env_de_bloc(b_2, b), u_2) // On analyse ce bloc et on récupère les prochains à analyser.
                U_p[c] ← u_2 // On met à jour l'usine pour ce contexte.
                U[p] ← U_p // On met à jour les usines pour ce bloc.
                return analyse(l'@l_2, pile_abs, B, U) // Récursion sur les nouveaux blocs à analyser.
            end
        else
            b_2, i ← B(p) // L'en-tête et le corps de ce bloc.
            l_2 ← analyse_de_bloc(i, s, env_de_bloc(b_2, b), u) // On analyse ce bloc et on récupère les prochains à analyser.
            U_p ← ∅ // On crée un nouvel ensemble de contextes.
            U_p[c] ← u // On initialise l'usine pour ce contexte.
```

Terminaison Les identifiants de variable ou de bloc (pointeur) sont en nombre fini dans le code et ne sont jamais générés lors de l'analyse, leur nombre est donc borné par la taille du programme. Chaque point d'allocation étant une variable, un ensemble de points d'allocations est par conséquent également borné par la taille du programme. L'union de deux entiers converge évidemment vers un point fixe. Étant donné que les pointeurs, les tags ainsi que les ensembles de points d'allocations sont des ensembles bornés par la taille du programme, l'union de deux fermetures ou de deux valeurs taggées est garantie de converger vers un point fixe. Il en est de même pour l'union de deux usines qui pour un ensemble fini de point d'allocation attribue des valeurs dont l'union converge. L'abstraction de la pile d'appels quant à elle est censée garantir que celle-ci ne croisse pas infiniment, ce qui borne le nombre de contexte d'appel pour chaque bloc. Même s'il ne s'agit pas d'une preuve formelle, tous ces arguments semblent confirmer que l'analyse est garantie de terminer.

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 éxécutable (c'est à dire générer le CFG éxécutable 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 éxécutable

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 éxécutable car les modifications apportées peuvent casser la sémantique d'appel ce qui doit être représenté au niveau de la pile.

Langage

Le CFG éxécutable 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.

$\mathbb{V} \coloneqq \mathbb{N}$ (variables)

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

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

$\mathbb{S} \coloneqq \mathbb{F}^*$ (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.

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

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

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

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

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

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

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.

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

Génération du CFG éxécutable

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

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

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

- 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.

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

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

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

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

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

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

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

Taille

La taille du CFG éxécutable 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 occurences des variables et blocs sont comptées afin de supprimer le code mort mais aussi pour l'inlining.

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 éxécutable que j'interprête 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 apparaîssent pour des programmes plus longs.

Benchmarks

Durant l'interprêtation j'enregistre diverses informations intéressantes, comme le nombre de sauts, de variables lues ou écrites. Celles-ci sont affichées après éxécution 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.

Inlining partiel

Une technique 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 technique 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 technique 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 technique 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

J'ai beaucoup travaillé sur les représentations intermédiaires essentielles pour à la fois réaliser la meilleure analyse possible et me permettre d'avoir tous les outils en main pour inliner. J'ai commencé avec 2 représentations, l'AST et la forme CPS (désormais le CFG) puis j'en ai créé 2 nouvelles, l'AST rafraîchi et le CFG exécutable pour alléger les différentes transformations. Toutes ont beaucoup évoluées pour me rapprocher de la sémantique la plus simple et la plus natuelle, ce qui a demandé beaucoup de choix pas toujours évidents. Le fait de les avoir formalisées dans ce rapport m'a permis de corriger certaines erreurs qui auraient été difficile de détecter dans le code.

J'ai évidemment passé beaucoup de temps sur l'interprétation abstraite et rencontré de nombreuses difficultés dont certaines n'ont pas été surmontées. En particulier certaines erreurs rencontrées avec les abstractions de pile n-CFA n'apparaissant pas avec l'abstraction par motifs laissent supposer que je n'exploite pas toujours de la bonne manière la pile abstraite dans mon algorithme. J'ai également des erreurs qui peuvent apparaître lorsque je réalise au moins 2 analyses consécutives mais uniquement avec une étape de propagation entre les deux. Exceptées ces deux problèmes rencontrés, l'analyse tient la plupart de ses promesses pour ce que j'attends d'elle, c'est à dire me donner de manière la plus exhaustive possible les blocs candidats pour les appels indirects, et d'assez bonnes précisions sur les valeurs taggées. Il est néanmoins très important de noter que durant ce stage j'ai fait totalement abstraction de la complexité de mon analyse. J'essaye de garantir de mon mieux qu'elle termine mais il est fort probable qu'elle soit exponentielle et la toute petite taille de mes tests ne me permet de m'en faire une idée.

Concernant l'inlining, d'après les premiers tests que j'ai réalisés, il apparaît que spécialiser systématiquement les blocs de très petite taille (de l'ordre d'1 instruction) permet à la fois de réduire grandement la taille du code tout en diminuant les instructions exécutées lors de l'évaluation. Augmenter légèrement ce seuil (de l'ordre de 2-3 instructions) permet généralement de gagner quelques instructions exécutées mais augmente aussi la taille du code, ce qui fait que les bénéfices sont durs à déterminer. Au delà de quelques instructions, sans autre forme d'heuristiques comme par exemple l'élimination de branchements en connaissant le motif, il semblerait que cela n'ait plus aucun ou très peu d'impact sur le nombre d'instructions exécutées tout en augmentant considérablement la taille du code, ce qui n'est évidemment pas souhaitable.

Pour conclure je suis assez satisfait de ce que j'ai appris et produit durant ce stage, à la fois d'un point de vue théorique et pratique. J'ai sous la main un ensemble de phases de compilations cohérentes pour un petit langage certes mais très expressif et je pense être capable sans trop de difficultés d'implémenter et tester de nouvelles heuristiques d'inlining. Pour mener au mieux de telles expérimentations il me sera par contre nécessaire de disposer d'une base de tests exhaustifs et d'en assurer l'automatisation, ce qui en soit ne sera probablement pas la chose la plus aisée de ce stage.