Heuristique d'inlining complexe

Heuristique d'inlining complexe

Adrien Simonnet

Sorbonne Université

Avril - Septembre 2023

Table of Contents

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
```

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

$$\textbf{if}\ \ c\ \ \textbf{then}\ \ x + 1\ \ \textbf{else}\ \ x - 1$$

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
```

Langage source

Types Somme et filtrage par motifs

La possibilité de construire des types Somme est une des fonctionnalités essentielles d'OCaml, permet de représenter quasiment n'importe quelle structure de données et d'augmenter la complexité de l'analyse de valeurs. 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, N
```

Langage utilisé

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

Lexique

- (* *) 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

La grammaire est la même que celle d'[OCaml](https://v2.ocaml.org/releases/5.0/manual/language.html) pour l'ensemble des jetons supportés.

Identificateurs

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

Identificateurs

$\mathbb{S} := \textit{string}$
$\mathbb{V} := \mathbb{S}$
$\mathbb{T} := \mathbb{S}$

Filtrage par motif

J'ai choisi de ne supporter que l'essentiel pour ce qui est du filtrage par motif.

Filtrage par motif

Deconstructor : $\mathbb{T} \times \mathbb{V}^* \mapsto \mathbb{M}$

Joker : $\mathbb{V} \mapsto \mathbb{M}$

Opérateurs binaires

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

Opérateurs binaires

Add : $\mathbb{B}$
Sub : $\mathbb{B}$

Expressions

Les expressions constituent la base d'un langage fonctionnel.

Expressions

Var : $\mathbb{V} \mapsto \mathbb{E}$

Fun : $\mathbb{V} \times \mathbb{E} \mapsto \mathbb{E}$

App : $\mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

LetRec : $(\mathbb{V} \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$

Int : $\mathbb{Z} \mapsto \mathbb{E}$

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

If : $\mathbb{E} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

Type : $\mathbb{S} \times (\mathbb{T} \times \mathbb{S})^* \times \mathbb{E} \mapsto \mathbb{E}$

Constructor : $\mathbb{T} \times \mathbb{E}^* \mapsto \mathbb{E}$

Match : $\mathbb{E} \times (\mathbb{M} \times \mathbb{E})^* \mapsto \mathbb{E}$

Analyseur syntaxique

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

Résolution des noms

Identificateurs

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

Identificateurs

$\mathbb{V} := \mathbb{N}$

$\mathbb{T} := \mathbb{N}$

Opérateurs binaires

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

Opérateurs binaires

Add : $\mathbb{B}$

Sub : $\mathbb{B}$

Expressions

Pour toutes les expressions, à l'exception du filtrage par motif, seul le type des identificateurs change.

Expressions

Var : $\mathbb{V} \mapsto \mathbb{E}$

Fun : $\mathbb{V} \times \mathbb{E} \mapsto \mathbb{E}$

App : $\mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

LetRec : $(\mathbb{V} \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$

Int : $\mathbb{Z} \mapsto \mathbb{E}$

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

If : $\mathbb{E} \times \mathbb{E} \times \mathbb{E} \mapsto \mathbb{E}$

Constructor : $\mathbb{T} \times \mathbb{E}^* \mapsto \mathbb{E}$

Match : $\mathbb{E} \times (\mathbb{T} \times \mathbb{V}^* \times \mathbb{E})^* \times \mathbb{E} \mapsto \mathbb{E}$

Définitions

$\mathbb{E}_{ast} \times (\mathbb{V}_{ast} \mapsto \mathbb{V}) \times (\mathbb{T}_{ast} \mapsto \mathbb{T}) \vdash_{\text{ast'}} \mathbb{E} \times (\mathbb{V} \mapsto \mathbb{V}_{ast}) \times (\mathbb{V}_{ast} \mapsto \mathbb{V})$
$e\ A\ C \vdash_{\text{ast'}} e'\ S\ L$

- 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

Conventions

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.

Règles

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

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

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

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

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

Analyse du flot de contrôle

La conversion CPS/CFG transforme l'AST' en un ensemble de basic blocs. À 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.
La différence notable avec l'AST' est l'apparition des blocs (avec explicitation des variables libres) et l'absence d'expressions imbriquées.

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

Identificateurs

$\mathbb{V} := \mathbb{N}$

$\mathbb{T} := \mathbb{N}$

$\mathbb{P} := \mathbb{N}$

Les expressions correspondent aux instructions élémentaires qui construisent des valeurs.

Expressions

Int : $\mathbb{Z} \mapsto \mathbb{E}$
Var : $\mathbb{V} \mapsto \mathbb{E}$
Add : $\mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$
Sub : $\mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$
Closure : $\mathbb{P} \times \mathbb{V}^* \mapsto \mathbb{E}$
Constructor : $\mathbb{T} \times \mathbb{V}^* \mapsto \mathbb{E}$

Une instruction est soit une déclaration soit un branchement.

Instructions

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{I} \mapsto \mathbb{I}$
Call : $\mathbb{V} \times \mathbb{V}^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \mapsto \mathbb{I}$
CallDirect : $\mathbb{P} \times \mathbb{V} \times \mathbb{V}^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \mapsto \mathbb{I}$
If : $\mathbb{V} \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$
MatchPattern : $\mathbb{V} \times (\mathbb{T} \times \mathbb{V}^* \times \mathbb{P} \times \mathcal{P}(\mathbb{V}))^* \times (\mathbb{P} \times \mathcal{P}(\mathbb{V})) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$
Return : $\mathbb{V} \mapsto \mathbb{I}$
IfReturn : $\mathbb{P} \times \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$
MatchReturn : $\mathbb{P} \times \mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$
ApplyBlock : $\mathbb{P} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{I}$

Chaque bloc est défini différemment en fonction de l'expression à partir de laquelle il est construit.

Types de blocs

Clos : $\mathbb{V}^* \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
Return : $\mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
IfBranch : $\mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
IfJoin : $\mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
MatchBranch : $\mathbb{V}^* \times \mathcal{P}(\mathbb{V}) \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
MatchJoin : $\mathbb{V} \times \mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$
Cont : $\mathcal{P}(\mathbb{V}) \mapsto \mathbb{B}$

Définition

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

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

Explications

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

- 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'

Conventions

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

Règles

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

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

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

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

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

Nettoyage des alias

Le code CFG ainsi généré est susceptible de contenir de nombreux alias de variables (créés par la règle (Var)). 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 analyse plus efficace est souhaitable mais difficile à implémenter pour le peu de bénéfice que cela apporterait.

Spécialisation

La spécialisation consiste à dupliquer des blocs.

- Améliore la précision de l'analyse
- Première étape de l'inlining

La copie doit :

- Conserver les invariants du CFG
- Modifier les appels directs

Les blocs sont choisis selon leur taille.

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.
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).

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

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.

Abstraction

Top : $\mathbb{I}$

Singleton : $\mathbb{Z} \mapsto \mathbb{I}$

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

Union

$$x \sqcup y = \begin{cases} \text{Singleton } i \text{ si } x = y = \text{Singleton } i \\ \text{Top sinon} \end{cases}$$

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.

Abstraction

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

L'union de deux fermetures consiste à conserver les entrées distinctes et d'unir les points d'allocations des entrées communes.

Union

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

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.

Abstraction

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

L'union de deux valeurs taggées consiste à conserver les entrées distinctes et d'unir les points d'allocations des entrées communes.

Union

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

Valeur abstraite

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

Abstraction

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 $\mathbb{V} := \mathcal{P}(\mathbb{V})$. 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.
$\mathbb{S} := (\mathbb{P} \times \mathcal{P}(\mathbb{V})^*)^*$ (pile)
Pour assurer 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.

Abstraction de la pile

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.

Exemple

ABCBC a un motif BC de taille 2 et sera remplacée par ABC

Abstraction de la pile

n-CFA

n-CFA consiste à conserver les n derniers contextes de la pile.

Exemple

En 1-CFA, la pile d'appels ABCBC sera remplacée par C

Définitions

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 : $(\mathbb{P} \times \overline{\mathbb{B}} \times \mathbb{S} \times \mathbb{U})^* \times (\mathbb{S} \mapsto \mathbb{S}) \times (\mathbb{P} \mapsto \mathbb{B}) \times (\mathbb{P} \mapsto ((\mathbb{S} \times \overline{\mathbb{B}}) \mapsto \mathbb{U})) \mapsto (\mathbb{P} \mapsto (\overline{\mathbb{B}} \times \mathbb{U}))$
L'analyse d'un bloc se fait par la fonction '$analyse_d\, e_b\, loc$' qui à partir d'un bloc, d'une pile, d'un environnement et d'une usine renvoie une liste de blocs à analyser.
'$env_d\, e_b\, loc$' est une fonction qui génère l'environnement à partir de l'en-tête d'un bloc et les valeurs abstraites qui lui sont passées.

Algorithme

Algorithm 1: Analyse du programme

Input : l pile_abs B U

```
// Il ne reste plus aucun bloc à analyser.
1 if l est vide then
2     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.
3 else
4     p, b, s, u ← hd(l) // Le premier bloc à analyser.
5     l' ← tl(l) // Les autres blocs à analyser.
6     s̄ ← pile_abs(s) // Abstraction de la pile.
7     c ← (s̄, b) // Le contexte (pile et en-tête).
      // Ce bloc a déjà été analysé.
8     if p ∈ 𝒟(U) then
9         U_p ← U[p] // Les usines associées à ce bloc.
          // Ce contexte a déjà été analysé pour ce bloc.
```

Terminaison

Quelques idées justifiant la terminaison de l'analyse.

- Identifiants jamais générés
- Un point d'allocation est un identifiant
- L'union de deux valeurs converge
- L'union de deux usines converge
- Abstraction de la pile d'appels

Propagation

La propagation modifie le CFG pour y faire apparaître les résultats de l'analyse.

- Expressions transformées en constantes
- Éliminations de branches
- Appels indirects transformés en appels directs

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.
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).

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

Identifiants

$$\mathbb{V} := \mathbb{N}$$
$$\mathbb{P} := \mathbb{N}$$
$$\mathbb{F} := \mathbb{P} \times \mathbb{V}^*$$
$$\mathbb{S} := \mathbb{F}^*$$

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.

Expressions

Const : $\mathbb{Z} \mapsto \mathbb{E}$
Pointer : $\mathbb{P} \mapsto \mathbb{E}$
Var : $\mathbb{V} \mapsto \mathbb{E}$
Add : $\mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$
Sub : $\mathbb{V} \times \mathbb{V} \mapsto \mathbb{E}$
Tuple : $\mathbb{V}^* \mapsto \mathbb{E}$
Get : $\mathbb{V} \times \mathbb{N} \mapsto \mathbb{E}$

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.

Instructions

Let : $\mathbb{V} \times \mathbb{E} \times \mathbb{I} \mapsto \mathbb{I}$
ApplyDirect : $\mathbb{P} \times \mathbb{V}^* \times \mathbb{S} \mapsto \mathbb{I}$
ApplyIndirect : $\mathbb{V} \times \mathbb{V}^* \times \mathbb{S} \mapsto \mathbb{I}$
If : $\mathbb{V} \times (\mathbb{N} \times \mathbb{P} \times \mathbb{V}^*)^* \times (\mathbb{P} \times \mathbb{V}^*) \times \mathbb{S} \mapsto \mathbb{I}$
Return : $\mathbb{V} \mapsto \mathbb{I}$

CFG exécutable

Bloc

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

Bloc

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

Définitions

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.

Règles

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

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

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

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

$$\frac{}{\left(\quad\right)\ i \vdash_{\text{cfg'}} \quad} \quad \text{(IfJoin)}$$

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 occurences des variables et blocs sont comptées afin de supprimer le code mort mais aussi pour l'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).

C'est le CFG exécutable 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.

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.