Le psittacisme pour permettre l'oubli motivé : implémentation et vérification du lambda lifting pour le compilateur CakeML

Rapport de stage par

Léo Andrès

22 août 2018

Master 1 Jacques Herbrand

école normale supérieure paris–saclay

Sous la direction de

Scott Owens

et de

Hugo Férée

Effectué à

University of Kent

Résumé

Ce document est un rapport de mon stage *Le psittacisme pour permettre l'oubli motivé : implémentation et vérification du lambda-lifting dans le compilateur CakeML*, effectué dans le cadre de ma formation en Master 1 Jacques Herbrand à l'École Normale Supérieure Paris-Saclay. Ce stage s'est déroulé sous la direction conjointe de Scott Owens et de Hugo Férée.

Table des matières

1 Contexte du stage 2
1.1 L'École Normale Supérieure Paris-Saclay . . . 2
1.2 L'Université du Kent . . . 2
1.3 L'équipe PLAS . . . 2

2 CakeML 2
2.1 Les compilateurs . . . 3
2.2 Sémantique et assistants de preuve . . . 4

3 Le lambda lifting 4
3.1 Les optimisations du compilateur . . . 4
3.2 Analogie avec la langue française . . . 5
3.3 Un exemple informatique . . . 6

4 Les algorithmes existants pour le lambda lifting 7
4.1 Différences . . . 7
4.2 Points communs . . . 7
4.2.1 Analyse de portée . . . 7
4.2.2 Désanonymisation . . . 8
4.2.3 Modification des définitions et appels de fonction . . . 8
4.2.4 Block floating . . . 9

5 La question du langage 9

6 Analyse de portée 11
6.1 L'algorithme . . . 11
6.2 Preuves . . . 11
6.2.1 Preuves de terminaison . . . 11
6.2.2 Alpha équivalence . . . 11
6.2.3 Alpha équivalence et analyse de portée . . . 12
6.2.4 Alpha équivalence et évaluation . . . 12

7 Notre algorithme 12
7.1 Présentation générale . . . 12
7.2 Cas des fonctions mutuellement récursives . . . 13
7.3 Exemple . . . 15

8 Comparaison avec l'état de l'art 18

9 Travaux futurs 18

10 Des conditions optimales 19

A Chaîne de compilation de cakeml 21

B Flat lang 22

C Code source des algorithmes et preuves 23

1 Contexte du stage

1.1 L'École Normale Supérieure Paris-Saclay

Ce stage s'inscrit dans le cadre de ma formation à l'École Normale Supérieure Paris-Saclay. L'École Normale Supérieure Paris-Saclay a, entre autres, pour but de former ses étudiants *par la recherche, pour la recherche*; ainsi, en tant qu'étudiant du Master 1 Jacques Herbrand, il est obligatoire d'effectuer un stage de recherche à l'étranger durant l'été. Il est possible d'effectuer un stage plus long, qui couvre alors aussi le second semestre : c'est le choix que j'ai fait.

1.2 L'Université du Kent

Le stage a eu lieu à l'Université du Kent, au sein de la School of Computing située à Canterbury et constituée de plusieurs équipes de recherche. J'ai travaillé au sein de l'équipe Programming Languages and Systems.

1.3 L'équipe plas

Un des axes de recherche de l'équipe Programming Languages and System est l'étude des langages de programmation, tant du point de vue de l'implémentation des langages, que de leur formalisation afin de prouver diverses propriétés.

2 CakeML

CakeML est l'un des projets menés - en partie - par l'équipe plas. Il s'agit, en résumé, d'un *compilateur vérifié* pour un dialecte ml proche de Standard ML, diverses informations à ce propos sont disponibles sur le site web du Standard ML Family project.

2.1 Les compilateurs

Tout d'abord, afin de donner une idée de ce qu'est un compilateur au lecteur profane, il faut tenter de montrer ce que sont les ordinateurs et les langages de programmation. Pour cela, reposons-nous sur cette citation[Bal12] :

> L'ordinateur est un exécutant parfait. Que des ordres lui soient donnés, il les suit sans sourciller. Il va vite, il ne se trompe pas, il ne connaît pas l'Ennui. Il fait tout ce qui lui est demandé, rien que ce qui lui est demandé, et tout cela sans s'inquiéter des raisons de ses actes.
>
> Et pour cause, l'ordinateur vit dans le *comment*, et ne comprendrait rien au *quoi* ni au *pourquoi*.
>
> Il est parfaitement inutile de lui dire "j'ai une liste de nombres et j'aimerais en connaître le produit". Pour le faire bouger, tout ce qui compte est de lui préciser la manière dont il doit s'y prendre et lui détailler pas à pas tout ce qu'il doit faire, en n'utilisant que les quelques dizaines d'*instructions* de base qu'il comprend, comme multiplier deux nombres entiers ou aller rechercher en mémoire une information préalablement enregistrée.
>
> Chaque ordinateur a ainsi son propre *langage machine*, formé d'une poignée de mots et de la plus pauvre des grammaires, dont les phrases permettent seulement de lui affecter les quelques tâches qu'il sait réaliser. Pour utiliser ce langage, il faut connaître parfaitement les mécanismes de l'ordinateur, et être capable sans le secours d'Ornicar de prévoir en détail les effets d'une jungle de monosyllabes.
>
> Avouons que programmer dans ces conditions est laborieux. Pour simplifier cela nous utilisons des *langages de programmation* au vocabulaire et aux structures grammaticales plus riches.

Une fois éclairé par cette explication, il est plus simple de comprendre le rôle d'un compilateur. Il s'agit simplement de l'outil qui transforme un programme généralement écrit par un humain dans un langage de programmation *riche* vers un autre langage de programmation plus *pauvre*, celui compréhensible par l'ordinateur. On parle généralement plutôt de langage *bas niveau* pour désigner un langage proche de ce que comprend l'ordinateur et de langage *haut niveau* pour un langage proche de ce que comprend l'humain.

Plusieurs questions se posent alors. Comment fait-on un compilateur ? En fait, le compilateur est lui-même un programme. Comme tout programme, il est courant qu'il ne soit pas parfait et qu'il contienne de nombreux bugs. Cela est problé-

matique. Si j'écris un programme, que je l'exécute et qu'il ne se comporte pas de la façon attendue, c'est peut-être parce que j'ai fait une erreur en écrivant le programme, ou peut-être parce que le compilateur a fait une erreur en transformant mon programme.

2.2 Sémantique et assistants de preuve

La solution à cela, est de faire une preuve mathématique qui garantit que le compilateur ne fera pas d'erreur. Pour cela, il faut définir formellement ce que signifie *ne pas faire d'erreur*. Un des moyens d'y arriver, est de définir mathématiquement la *sémantique* de notre langage. Une fois cela fait, on peut prouver que, pour n'importe quel programme, le compilateur préserve la sémantique. On a alors la garantie que le résultat produit par le compilateur se comportera exactement comme il devrait. Cela a pour conséquence qu'un programme correctement écrit se comportera exactement comme on l'attend.

Il est possible d'effectuer cette formalisation et ces preuves, non pas à la main sur une feuille de papier, mais au moyen de ce que l'on appelle des *assistants de preuve*. Cela revient à écrire les définitions et les preuves sous forme de programme.

La sémantique de CakeML a été formalisée au moyen de Lem. Lem permet d'exporter une sémantique donnée vers des définitions compréhensibles par différents assistants de preuve. Dans le cadre du projet CakeML, les preuves sont réalisées en utilisant HOL.

Une fois que la sémantique de notre langage a été formalisée sous forme de définition compréhensible [1] par un assistant de preuve, il ne reste plus qu'à écrire notre compilateur et à prouver que les transformations qu'il effectue préservent la sémantique.

3 Le lambda lifting

3.1 Les optimisations du compilateur

En réalité, le compilateur ne se contente généralement pas de transformer le programme naïvement. Il effectue des modifications qui permettent d'optimiser

1. Elle le sera aussi pour un humain particulièrement bien entraîné.

le résultat, afin d'obtenir un programme plus rapide, moins coûteux à la fin. Chacune de ces modifications est appelée une *optimisation*. Il pourrait par exemple s'agir de ne pas effectuer un calcul lorsque l'on utilise une opération avec son élément neutre, le code suivant par exemple :

```
let add x y =
  0 + x + y
```

Serait transformé ainsi :

```
let add x y =
  x + y
```

3.2 Analogie avec la langue française

Le lambda lifting est une optimisation possible. On peut en donner une intuition en passant par la langue française. Faisons pour cela appel au *Maître*, qui nous fournit la phrase suivante :

> On cherchait à l'impressionner, à devenir son préféré.

Les mots *l'* et *son* font ici référence à une chose qui n'est pas contenue dans la phrase. Il faut nous référer au contexte dans lequel est placée la phrase pour en saisir tout le sens. Et en effet, si l'on regarde quelques lignes plus haut dans l'œuvre :

> Elle répondait au nom de Bella. [...] On cherchait à l'impressionner, à devenir son préféré.

On s'aperçoit qu'il est ici question de *Bella*. En mathématiques et en informatique, lorsque l'on s'intéresse à une phrase — on parlera plutôt de *terme* —, si un mot — on parlera plutôt de *variable* — fait référence à une chose qui n'est pas définie au sein même du terme, on dit que la variable est *libre*. Sinon, elle est dite *liée*. Lorsque toutes les variables sont liées dans un terme, on dit qu'il est *clos*. Autrement, on parle de terme *ouvert*.

Le lambda-lifting consisterait tout simplement à éliminer toutes les variables libres. Il s'agirait par exemple de transformer notre exemple afin d'obtenir cela :

> On cherchait à impressionner Bella, à devenir son préféré.

Dès lors, puisque le terme est clos, on pourrait le changer de place dans le texte, son sens serait le même puisqu'il n'est plus dépendant du contexte.

En réalité, il s'agit plus exactement d'éliminer les variables libres de chaque *fonction*. Dès que l'on parle de fonction, il est plus simple de s'appuyer sur un langage de programmation, or il est difficile d'écrire un cours de programmation qui permette de comprendre tout ce qui va suivre en seulement quelques paragraphes. Le langage utilisé pour les exemples sera OCaml.

3.3 Un exemple informatique

Pour illustrer le lambda lifting, prenons l'exemple de la fonction `fold_right` suivante :

```
let fold_right f acc l =
  let rec aux = function
    | [] -> acc
    | h::t -> f h (aux t)
  in aux l
```

On remarque que les variables `acc` et `f` sont libres dans la fonction `aux`. Du point de vue du programmeur cela a l'avantage d'expliciter le fait qu'au cours des différents appels récursifs successifs de `aux`, les valeurs de ces variables restent les mêmes que celles que l'on avait lors de l'appel initial à `fold_right` ; cela permet aussi de réduire la taille du code écrit.

Cependant, si l'on se place du point de vue du compilateur, une variable libre signifie qu'il faudra créer ce que l'on appelle une *fermeture* — ou *closure* en anglais. Une fermeture est stockée dans un endroit de la mémoire appelé le *tas*.

Il existe plusieurs autres endroits où stocker de l'information. Le dilemme étant généralement le suivant : plus l'accès à un endroit permettant de stocker de la mémoire est rapide, plus la quantité d'informations qu'il est possible d'y stocker est faible. Par exemple, les *registres* sont ce qu'il y a de plus rapide, mais il n'est généralement possible d'y stocker que quelques nombres ; à l'opposé, le disque dur d'un ordinateur permet aujourd'hui de stocker une très importante quantité d'informations, mais il est très long d'y accéder. Le tas se situe à mi-chemin entre les registres et le disque dur.

Si l'on évite la création de la fermeture, il est alors possible d'utiliser les registres dans un plus grand nombre de cas ; et donc, d'obtenir un programme plus rapide.

Le but du lambda lifting est donc de se débarrasser des variables libres. Si l'on reprend l'exemple précédent, cela consisterait par exemple à transformer le programme de sorte à obtenir ceci :

```
let fold_right f acc l =
  let rec aux acc f = function
    | [] -> acc
    | h::t -> f h (aux acc f t)
  in aux acc f l
```

4 Les algorithmes existants pour le lambda lifting

4.1 Différences

Plusieurs algorithmes ont été présentés pour le lambda lifting. Lorsque l'on parle d'un algorithme pour le lambda lifting, il est généralement question d'une seule étape parmi toutes celles qui composent le processus de lambda lifting : le calcul des paramètres à ajouter à chaque fonction. Les autres étapes n'étant généralement pas abordées, elles sont en effet relativement simples à implémenter en comparaison. Ainsi, lors de la phase de calcul des paramètres, ils se distinguent par plusieurs caractéristiques :

- l'utilisation d'équations ensemblistes ou bien de graphes
- la complexité
- la nécessité d'une évaluation paresseuse
- l'optimalité dans le sens où l'on veut qu'aucun paramètre inutile ne soit ajouté aux fonctions

Voici un tableau résumant cela :

Algorithme	Méthode	Complexité	Évaluation par. requise	Optimal
[Joh85]	équations	$O(n^3)$	oui	oui
[DS02]	graphes	$O(n^2)$	non	non
[MM06]	graphes	$O(n^3)$	non	oui
[MS08]	graphes	$O(n^2)$	non	oui

4.2 Points communs

4.2.1 Analyse de portée

Chacun de ces algorithmes fait au départ l'hypothèse que tous les identifiants sont uniques, c'est-à-dire qu'il n'existe pas deux fonctions ou variables portant le même nom. Cette hypothèse est généralement fausse, il faut donc procéder à une première transformation du programme, nommée *analyse de portée*[2], dont le but est de rendre les identifiants uniques.

2. *Scope analysis* en anglais.

4.2.2 Désanonymisation

Dans les langages de programmation fonctionnelle, il est généralement possible d'écrire des *fonctions anonymes*. Comme toute fonction, elles peuvent contenir des variables libres, par exemple ici, `x` est libre dans la fonction anonyme :

```
let x = 1 in
List.iter (fun el -> print_int (el - x)) [0; 1; 2]
```

On va donc ajouter une phase de désanonymisation, elles subiront ainsi le même traitement que les autres ; sur notre exemple précédent, cela donnerait le résultat suivant :

```
let x = 1 in
List.iter (let f el = print_int (el - x) in f) [0; 1; 2]
```

4.2.3 Modification des définitions et appels de fonction

Une fois que pour chaque fonction on a calculé l'ensemble des paramètres à lui ajouter, il faut les rajouter à sa définition ainsi qu'à chaque appel fait à cette fonction. Sur notre exemple précédent, on aurait l'ensemble de solutions suivant :

$$\mathrm{sol}_f = \{x\}$$

On rajoute donc `x` à la définition de `f` ainsi qu'à chaque appel à la fonction :

```
let x = 1 in
List.iter (let f x el = print_int (el - x) in f x) [0; 1; 2]
```

Il faut bien noter que l'on place les nouveaux paramètres au début, afin d'éviter des problèmes en lien avec une éventuelle application partielle. [3] Il n'y aura de plus pas de conflits d'identifiants, puisqu'on a au préalable pris soin de les rendre uniques. Une fois cette étape effectuée, notre programme ne contient plus aucune variable libre.

3. Le lecteur curieux pourra essayer de placer `x` après `el` au moment de la définition de la fonction et observer le résultat.

4.2.4 Block floating

Dès lors, chaque fonction est indépendante des autres en termes de portée. On peut alors toutes les mettre au même niveau sous forme de fonctions mutuellement récursives. Si l'on reprend l'exemple de la fonction `fold_right`, on obtiendrait cela :

```
let rec aux acc f = function
  | [] -> acc
  | h::t -> f h (aux acc f t)

and fold_right f acc l =
  aux acc f l
```

Dans l'article original [Joh85], le lambda lifting, avant d'être vu comme une optimisation possible, était surtout un moyen de parvenir à compiler des fonctions imbriquées pouvant posséder des variables libres. L'étape de *block floating* était donc celle qui débarrassait le programme des fonctions imbriquées et où l'on avait donc atteint l'objectif visé. Dans notre cas, cette étape revêt moins d'importance, un équivalent étant déjà implémenté dans CakeML lors de la création des fermetures [4].

5 La question du langage

Lorsque l'on écrit un programme, on l'écrit sous forme de texte. Le compilateur commence alors par procéder à l'analyse lexicale et à l'analyse syntaxique du programme. On récupère alors un AST – *Abstract Syntax Tree* [5].

Cet AST n'est autre que notre programme original, représenté sous une forme facilement manipulable, on appelle cette représentation l'AST source.

Le lambda lifting est généralement décrit comme étant une transformation *source to source*, c'est-à-dire, opérant directement sur l'AST source, avant toute autre transformation.

Cependant, on ne peut pas effectuer le lambda lifting avant toute autre transformation. En effet, une des premières étapes, appelée la *vérification de type* [6], dont le but est de vérifier que notre programme est correctement typé, doit être

4. Les fermetures seront donc vides du fait du lambda lifting, mais les fonctions seront toujours indépendantes

5. Le français étant une langue dotée de nombreuses nuances, on a le choix entre *Arbre de Syntaxe absTraite* et *Arbre de Syntaxe abstraiTe* ; voire même *Arbre de SynTaxe abstraite* pour les plus téméraires.

6. *Type checking* en anglais.

effectuée *avant* le lambda lifting, pour une raison simple : un programme qui sera jugé correct par le compilateur ne le sera pas forcément après l'étape de lambda lifting. En voici un exemple ; le code suivant sera jugé comme étant bien typé :

```
let main () =
  let aux x = 42 in
  (aux 1) + (aux true)
let _ =
  print_int (main ())
```

Cependant, si l'on s'intéresse à son équivalent après lambda lifting [7] :

```
let rec aux x = 42
and main () =
  (aux 1) + (aux true)
let _ =
  print_int (main ())
```

On obtiendra l'erreur de type suivante [8] :

```
line 5, characters 17-21: Error: This expression has
type bool but an expression was expected of type int
```

Se pose alors la question suivante : à quel moment est-il judicieux d'effectuer le lambda lifting ?

Premièrement, on sait déjà qu'il est nécessaire de l'effectuer après la vérification de type. Il s'agit ensuite de choisir s'il est préférable de l'effectuer sur le langage source ou bien sur un langage intermédiaire. Le but étant d'avoir à la fois un langage dont la grammaire soit la plus petite possible, afin de minimiser le nombre de cas à traiter lors de l'implémentation [9] mais aussi de choisir un langage possédant une sémantique simple pour ne pas complexifier la preuve.

7. Cela n'a pas été précisé avant, mais l'étape de *block floating* transforme les fonctions imbriquées en fonctions mutuellement récursives.

8. Cette erreur est due à la *value restriction*. En effet, l'inférence de type dans le cas de la récursivité polymorphique est indécidable. Plus de détails sont donnés dans [Hen93].

9. Et donc, indirectement, de minimiser la taille de la preuve.

Un graphique représentant l'ensemble des langages intermédiaires est donné en annexe A. Le premier d'entre eux, appelé *flat lang*, est plus simple que le langage source : il n'y a plus de module, de tuple et de variable et fonction globale, les constructeurs sont simplifiés [10]…De plus, sa sémantique est comparable à celle du langage source.

Le langage intermédiaire suivant, *pat lang*, n'utilise plus des noms pour identifier les variables et les fonctions, mais des indices de DE BRUIJN ce qui rend difficile la manipulation des termes.

Le candidat idéal semble donc être *flat lang*, dont un aperçu est donné en annexe B.

6 Analyse de portée

6.1 L'algorithme

Un lien vers le code de l'algorithme est donné en annexe. L'idée de l'algorithme est simple : on parcourt récursivement notre programme et on maintient une liste d'association faisant correspondre à chaque identifiant son remplaçant. On ne peut utiliser de références, ce qui rend notre algorithme moins lisible ; en effet, dans le cas contraire, il aurait suffit de gérer le compteur dans la fonction `mk_unique_name` en utilisant une fermeture.

6.2 Preuves

6.2.1 Preuves de terminaison

Dès que l'on définit une fonction, si sa terminaison n'est pas évidente pour HOL, il faut en donner une preuve. Cela se fait généralement en utilisant la tactique `WF_REL_TAC` qui prend une relation bien fondée, puis en montrant ce que nous demande HOL par récurrence. Dans la plupart des cas, des relations déjà existantes suffisaient ; cela n'a pas été le cas au moment où l'on a modifié les identifiants puisqu'en effet, la relation usuelle `exp_size` utilise entre autres la taille des identifiants.

6.2.2 Alpha équivalence

L'idée ici a été de définir une relation s'apparentant à l'alpha équivalence. Cependant, dans notre cas, cette relation ne s'applique pas seulement à deux expressions, mais aussi à un environnement. On a ensuite pu prouver diverses propriétés sur cette relation, elle est notamment réflexive et transitive.

10. Ils ne sont plus identifiés par des noms mais par des entiers uniques.

6.2.3 Alpha équivalence et analyse de portée

Une fois cette relation d'alpha équivalence définie, il a été possible de montrer que pour un programme bien formé, c'est-à-dire clos et bien typé, ce programme et son image après analyse de portée sont alpha équivalents. Une version simplifiée du théorème est la suivante :

$$\begin{aligned} &\forall\,(\texttt{exp, exp', env}),\\ &\quad \texttt{is_valid_exp(env, exp)}\\ &\wedge\ \texttt{is_valid_env(env)}\\ &\wedge\ \texttt{compile_exp(env, exp)} = \texttt{(exp')}\\ &\Longrightarrow\ \texttt{alpha_eq_exp(env, exp, exp')} \end{aligned}$$

6.2.4 Alpha équivalence et évaluation

Dès lors, il reste à montrer que si deux programmes sont alpha équivalents, leur sémantique est identique. Pour cela, on montre que leur image par la fonction `eval` est la même :

$$\begin{aligned} &\forall\,(\texttt{exp, exp',env, env', result, result'}),\\ &\quad \texttt{alpha_eq_env(env, env')}\\ &\wedge\ \texttt{alpha_eq_exp(}\varnothing\texttt{, exp, exp')}\\ &\wedge\ \texttt{evaluate(env, exp)} = \texttt{result}\\ &\wedge\ \texttt{evaluate(env', exp')} = \texttt{result'}\\ &\Longrightarrow\ \texttt{result_eq(res, res')} \end{aligned}$$

7 Notre algorithme

7.1 Présentation générale

Notre algorithme utilise tout d'abord des équations ensemblistes, puis utilise momentanément des graphes pour traiter le cas des équations mutuellement récursives, puisque l'on ne dispose pas de l'évaluation paresseuse.

La complexité de l'algorithme est supérieure à celle des autres algorithmes. En effet, l'implémentation pour les opérations ensemblistes et sur les graphes utilise des listes et des listes d'associations. Il n'y a pas d'implémentation vérifiée des tables de hachage en HOL et il est de manière générale plus facile de raisonner sur les listes, d'où ce choix.

En revanche, notre algorithme est optimal, dans le sens où il ne rajoute aucun argument qui ne soit nécessaire.

Actuellement, la quasi-totalité des preuves de terminaison est réalisée. Il reste à prouver la correction de l'algorithme, pour l'instant seules quelques propriétés ont été prouvées.

On utilise un algorithme récursif sur les expressions. La plupart des cas sont simples à traiter, c'est pourquoi seul le cas des fonctions mutuellement récursives sera détaillé.

On maintient en permanence ces trois ensembles :

$$\begin{aligned} \mathtt{oldVars} &:= \text{l'ensemble des variables dans le scope courant} \\ \mathtt{oldFuns} &:= \text{l'ensemble des fonctions dans le scope courant} \\ \mathtt{oldSol}_g &:= \text{si } g \in \mathtt{oldFuns}\text{, son ensemble de solutions, sinon } \varnothing \end{aligned}$$

Ils sont au départ tous vides.

7.2 Cas des fonctions mutuellement récursives

Nos fonctions mutuellement récursives ne sont dotées que d'un seul argument [11]. On a ainsi quelque chose de cette forme :

$$\begin{aligned} &\text{let rec } f_1\ x_1 = e_1 \\ &\quad \text{and } f_2\ x_2 = e_2 \\ &\qquad \dots \\ &\quad \text{and } f_n\ x_n = e_n \\ &\text{in } e \end{aligned}$$

On pose alors :

$$\begin{aligned} \mathtt{introducedFuns} &:= \{f_1, \dots, f_n\} \\ \mathtt{introducedVars} &:= \varnothing \\ \mathtt{accessibleFuns} &:= \mathtt{oldFuns} \cup \mathtt{introducedFuns} \\ \mathtt{accessibleVars} &:= \mathtt{oldVars} \cup \mathtt{introducedVars} \end{aligned}$$

Notre langage ne dispose pas de variables mutuellement récursives comme c'est par exemple le cas en OCaml, d'où l'ensemble vide. Dans le cas contraire,

11. Dans le cas d'une fonction à plusieurs arguments, on aura tout simplement quelque chose de la forme `let rec f x = fun y -> ...`

il suffirait d'ajouter ces variables à l'ensemble idoine pour que l'algorithme fonctionne tout de même.

Puis, pour tout $i \in \{1, \dots, n\}$, on note $\mathtt{sol}_i$ l'ensemble de solutions de la fonction f_i. On pose alors :

$$\mathtt{sol}_i := \mathtt{localSol}_i \cup \mathtt{globalSol}_i$$

Pour un i donné, l'ensemble $\mathtt{localSol}_i$ ne dépend pas des autres fonctions appartenant à $\mathtt{introducedFuns}$. On commencera donc par calculer cet ensemble pour chacune des fonctions de $\mathtt{introducedFuns}$.

On note $\mathtt{freeID}_i$ l'ensemble des identifiants libres de f_i. On pose :

$$\mathtt{localDeps}_i := \mathtt{oldFuns} \cap \mathtt{freeID}_i$$

On a alors la première partie de la solution donnée par :

$$\mathtt{localSol}_i = (\mathtt{accessibleVars} \cap \mathtt{freeID}_i) \cup (\bigcup_{g \in \mathtt{localDeps}_i} \mathtt{oldSol}_g)$$

On pose :

$$\mathtt{globalDeps}_i := (\mathtt{introducedFuns} \cap \mathtt{freeID}_i) \setminus \{f_i\}$$

On a alors :

$$\mathtt{globalSol}_i = \bigcup_{g_j \in \mathtt{globalDeps}_i} \mathtt{sol}_j$$

C'est lors du calcul de cette partie de la solution que l'évaluation paresseuse est utilisée dans l'article original par [Joh85]. Cependant, cette technique n'est pas disponible dans notre cas ; c'est pourquoi l'on va utiliser des graphes, en nous appuyant sur l'approche de [DS02].

On va partir du graphe d'appel au sein de notre ensemble de fonctions mutuellement récursives. Les nœuds de notre graphe seront donc l'ensemble des $f_i \in \mathtt{introducedFuns}$. Puis, pour un nœud f_i donné, pour chaque $f_j \in \mathtt{globalDeps}_i$, on trace une arrête orientée allant de f_i à f_j.

On calcule alors l'ensemble des composantes fortement connexes de notre graphe. On utilise l'algorithme de Kosaraju[Sha81], dans l'optique de simplifier les preuves, plutôt que celui de Tarjan[Tar72].

Il s'agit alors de regrouper les composantes fortement connexes de notre graphe original. L'idée principale étant de remarquer que si deux fonctions appartiennent à la même composante fortement connexe, elles auront le même ensemble de solutions.

On obtient alors un graphe acyclique. On peut alors, pour un nœud donné, calculer l'ensemble des solutions des fonctions qui le composent en utilisant un algorithme récursif sur les voisins extérieurs de ce nœud, qui terminera puisqu'il n'y a plus de cycles. Il faut bien avoir pris soin de ne pas avoir un nœud ayant une arrête le reliant à lui-même.

Autrement dit, si notre nœud A contient les fonctions $\{g_1, \dots, g_m\}$ et qu'il a pour voisins extérieurs l'ensemble des nœuds $\mathrm{outNeighbours}$, on pose :

$$\begin{aligned} \mathrm{SCCSol}_A &= \mathrm{localSCCSol}_A \cap \mathrm{globalSCCSol}_A \\ \mathrm{localSCCSol}_A &= \bigcup_{i \in \{1,\dots,m\}} \mathrm{localSol}_i \\ \mathrm{globalSCCSol}_A &= \bigcup_{X \in \mathrm{outNeighbours}} \mathrm{SCCsol}_X \end{aligned}$$

On a alors :

$$\forall f_i \in A, \mathrm{sol}_i = \mathrm{SCCSol}_A$$

Il ne reste plus qu'à poursuivre notre algorithme récursivement sur chacun des e_i avec $i \in \{1, \dots, n\}$ et sur e.

7.3 Exemple

Prenons le code suivant [12] :

```
let f x =

  let rec g ... =
    ... x ... a ... b .... c ... d ...
  and     a ... =
    ... b ...
  and     b ... =
    ... c ... d ...
  and     c ... =
    ... g ...
  and     d ... =
    ...
  in

  ...
```

12. Inspiré d'un exemple utilisé dans [MS08].

Au moment de traiter l'ensemble des fonctions mutuellement récursives, on a les ensembles suivants :

$$\begin{aligned} \mathtt{oldVars} &= \{x\} \\ \mathtt{oldFuns} &= \varnothing \\ \mathtt{oldSol} &= \varnothing \end{aligned}$$

Si $\mathtt{f}$ était récursive, on aurait $\mathtt{oldFuns} = \{f\}$. On commence alors par calculer les ensembles suivants :

$$\begin{aligned} \mathtt{introducedFuns} &= \{g, a, b, c, d\} \\ \mathtt{introducedVars} &= \varnothing \\ \mathtt{accessibleFuns} &= \{g, a, b, c, d\} \\ \mathtt{accessibleVars} &= \{x\} \end{aligned}$$

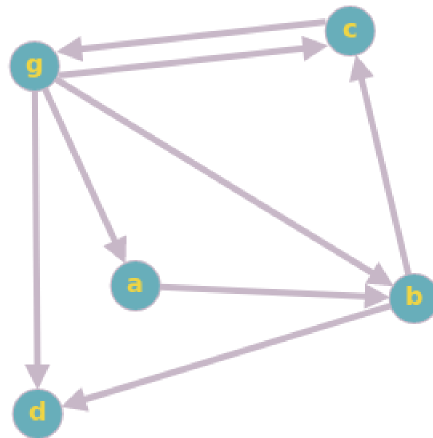
Puis :

$$\begin{aligned} \mathtt{freeID}_g &= \{x, a, b, c, d\} \\ \mathtt{freeID}_a &= \{b\} \\ \mathtt{freeID}_b &= \{c, d\} \\ \mathtt{freeID}_c &= \{g\} \\ \mathtt{freeID}_d &= \varnothing \end{aligned}$$

Ici, $\mathtt{oldFuns} = \varnothing$, donc $\forall i, \mathtt{freeID}_i = \mathtt{localDeps}_i$. D'où :

$$\begin{aligned} \mathtt{localSol}_g &= \{x\} \\ \mathtt{localSol}_a = \mathtt{localSol}_b = \mathtt{localSol}_c = \mathtt{localSol}_d &= \varnothing \end{aligned}$$

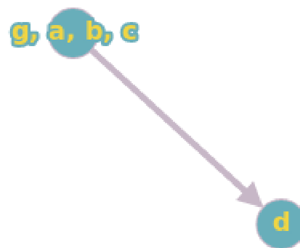
Il faut maintenant passer à une représentation utilisant des graphes. On aura le graphe d'appel initial suivant :



Notre calcul des composantes fortement connexes nous donne l'ensemble suivant :

$$\{\{g, a, b, c\}, \{d\}\}$$

On reconstruit le graphe d'appel entre les composantes fortement connexes et en retirant les arrêtes d'un nœud vers lui-même :



On a alors :

$$\begin{aligned} \mathtt{localSCCSol}_{\{g,a,b,c\}} &= \{x\} \\ \mathtt{localSCCSol}_{\{d\}} &= \varnothing \end{aligned}$$

D'où :

$$\begin{aligned} \mathtt{globalSCCSol}_{\{g,a,b,c\}} &= \{x\} \\ \mathtt{globalSCCSol}_{\{d\}} &= \varnothing \end{aligned}$$

Finalement :

$$\begin{aligned} \mathtt{sol}_{\mathtt{g}} = \mathtt{sol}_{\mathtt{a}} = \mathtt{sol}_{\mathtt{b}} = \mathtt{sol}_{\mathtt{c}} &= \{\mathtt{x}\} \\ \mathtt{sol}_{\mathtt{d}} &= \varnothing \end{aligned}$$

Si l'on introduit les paramètres à l'endroit idoine, il est facile de vérifier à la main que plus aucune fonction ne possède de variable libre, qu'aucune ne possède de paramètre inutile [13] et qu'il ne leur manque aucun paramètre.

8 Comparaison avec l'état de l'art

En comparaison des algorithmes existant, notre algorithme remplit tous les critères intéressants à l'exception de la complexité, qui peut cependant encore être améliorée. En revanche, dans les différents articles cités précedemment, aucun ne donne plus qu'une intuition quant à la correction de l'algorithme. Une preuve est donnée dans [KC12], cependant, l'algorithme n'y est pas optimal [14], seulement l'étape de calcul des paramètres y est traitée et la preuve n'est pas vérifiée par un assistant de preuve.

9 Travaux futurs

Plusieurs points peuvent encore être améliorés ou étudiés, notamment :

- amélioration de la complexité, notamment en implémentant les ensembles et les graphes de façon plus efficace
- une série de *benchmarks* permettant de mesurer l'impact du lambda lifting sur le temps d'exécution
- la gestion de l'analyse de portée pour différents types de *scopes* non utilisés dans CakeML : l'*overloading*, les *namespaces* ...
- la réutilisation de l'analyse de portée dans le cadre du projet Trustworthy Refactoring mené au sein de l'équipe PLAS.
- une comparaison avec la *Skolémisation*, il semblerait en effet que ces deux processus partagent certains aspects

13. $\mathtt{x}$ n'est pas ajouté aux arguments de $\mathtt{d}$
14. Dans le sens où des paramètres inutiles seront ajoutés.

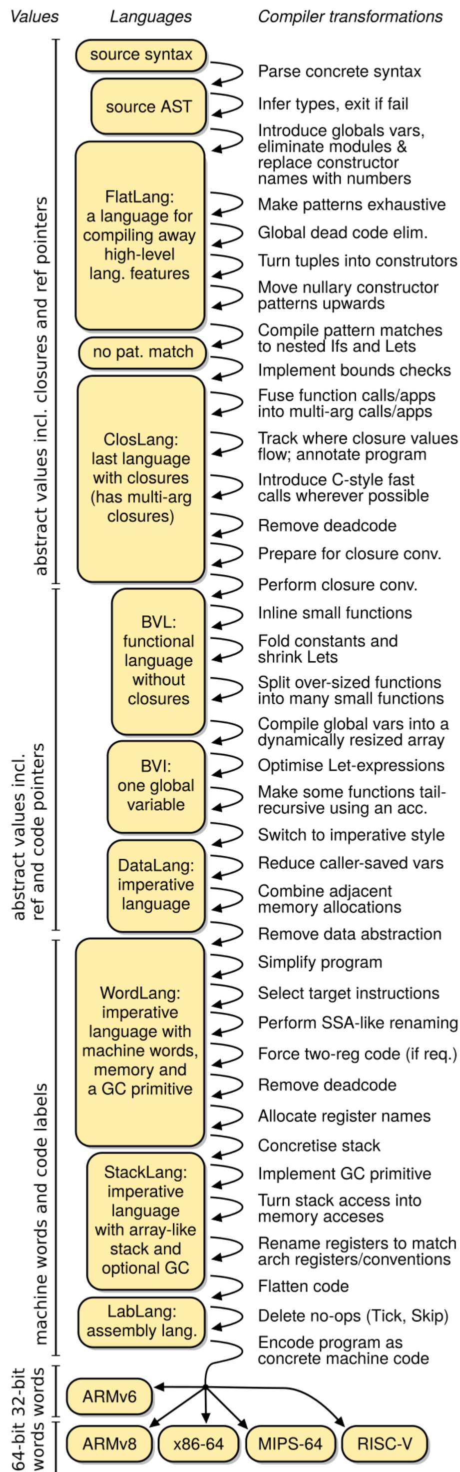
10 Des conditions optimales

Ce stage s'est déroulé sous des conditions optimales ; de quoi se promener, du café, un piano, et la sémantique préservée : que demander de plus ?

Il me tient donc à cœur de remercier plusieurs personnes pour avoir permis cela. Tout d'abord, David Baelde qui a su me diriger vers ce stage malgré mon absence quasi-totale d'idée quant à ce que je voulais faire [15]. Scott Owens pour m'avoir accepté comme stagiaire et tout ce qui s'en est suivi [16]. Hugo Férée, pour avoir pris le temps de me fournir de nombreuses explications sur les assistants de preuves, la culture anglaise et son aide permanente. Tous mes camarades du bureau sw12, qui ont égayé bien des journées : Anouk, Marco, Reuben et Simon [17]. L'ensemble des membres de l'équiple plas (et les quelques autres venus d'ailleurs !) pour ces repas et ces vendredis soirs partagés. Enfin, j'ai eu la chance de croiser le chemin de plusieurs autres personnes qui ont donné une petite touche céleste à ces quelques mois, je ne peux toutes les nommer ici, j'aimerais cependant dire merci de façon particulière à Charlotte et ειρηνη pour les instants à Londres, à Tim et Rebecca pour avoir été d'aussi formidables *housemates* et à Dulcie qui a su envoyer un peu de *cosmic dust* au moyen de sa voix lors de ces nombreuses soirées jazz.

15. "J'aime bien OCaml, la compilation et la logique."
16. C'est-à-dire : beaucoup de choses !
17. Et bien évidemment Hugo une seconde fois !

A Chaîne de compilation de cakeml



B Flat lang

```
(** Langage **)

type op = Add | Sub (* etc. *)
type lit = Int of int | String of string (* etc. *)
type varN = string
type con_typ = (int * (int option)) option

type pat =
  | Pany (* _ *)
  | Pvar of varN (* x *)
  | Pcon of con_typ * (pat list) (* Constructor (pat1, ...) *)
  | Pref of pat

type exp =
  | Raise      of exp
  (* try `exp` with `pat_exp_map` *)
  | Handle     of exp * pat_exp_map
  | Lit        of lit
  | Con        of con_typ * (exp list)
  | Var_local of varN
  | Fun        of varN * exp (* fun `varN` -> `exp` *)
  | App        of op * (exp list)
  (* if `exp` then `exp` else `exp *)
  | If         of exp * exp * exp
  (* match `exp` with `pat_exp_map` *)
  | Mat        of exp * pat_exp_map
  (* Some varN, then:    let `varN` = `exp` in `exp *)
  (* None, then:          `exp`; `exp` *)
  | Let        of (varN option) * exp * exp
  (* let rec `varN` `varN` = `exp` and ... in `exp` *)
  | Letrec     of ((varN * varN * exp) list) * exp
and pat_exp_map = (pat * exp) list

(** Sémantique **)

type v =
  | Lit_v        of lit
  | Con_v        of con_typ * (v list)
  | Closure      of env * varN * exp
  | Rec_closure of env * ((varN * varN * exp) list) * varN
  | Vector_v     of (v list)
and env = (varN * v) list
```

C Code source des algorithmes et preuves

- application successives des différentes étapes composant le lambda lifting
- analyse de portée
- désanonymisation des fonctions
- calcul des paramètres à ajouter
- modification des définitions et des appels de fonction
- implémentation des ensembles
- implémentation des graphes
- preuve de l'analyse de portée

Références

[Bal12] Thibaut Balabonski. « La pleine paresse, une certaine optimalité : partage de sous-termes et stratégies de réduction en réécriture d'ordre supérieur ». 2012PA077198. Thèse de doct. 2012, 1 vol. (368 p.) URL : `http://www.theses.fr/2012PA077198` (cf. p. 3).

[Joh85] Thomas Johnsson. « Lambda Lifting : Transforming Programs to Recursive Equations ». In : *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France : Springer-Verlag New York, Inc., 1985, p. 190-203. ISBN : 3-387-15975-4. URL : `http://dl.acm.org/citation.cfm?id=5280.5292` (cf. p. 7, 9, 14).

[DS02] Olivier Danvy et Ulrik P. Schultz. « Lambda-Lifting in Quadratic Time ». In : *Functional and Logic Programming*. Sous la dir. de Zhenjiang Hu et Mario Rodríguez-Artalejo. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, p. 134-151. ISBN : 978-3-540-45788-6 (cf. p. 7, 14).

[MM06] Marco T. Morazán et Barbara Mucha. « Improved Graph-Based Lambda Lifting. » In : *Software Engineering Research and Practice*. Citeseer. 2006, p. 896-902 (cf. p. 7).

[MS08] Marco T. Morazán et Ulrik P. Schultz. « Optimal Lambda Lifting in Quadratic Time ». In : *Implementation and Application of Functional Languages*. Sous la dir. d'Olaf Chitil, Zoltán Horváth et Viktória Zsók. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 37-56. ISBN : 978-3-540-85373-2 (cf. p. 7, 15).

[Hen93] Fritz HENGLEIN. « Type Inference with Polymorphic Recursion ». In : *ACM Trans. Program. Lang. Syst.* 15.2 (avr. 1993), p. 253-289. ISSN : 0164-0925. DOI : 10.1145/169701.169692. URL : http://doi.acm.org/10.1145/169701.169692 (cf. p. 10).

[Sha81] Micha SHARIR. « A strong-connectivity algorithm and its applications in data flow analysis ». In : *Computers & Mathematics with Applications* 7.1 (1981), p. 67-72 (cf. p. 14).

[Tar72] Robert TARJAN. « Depth first search and linear graph algorithms ». In : *SIAM JOURNAL ON COMPUTING* 1.2 (1972) (cf. p. 14).

[KC12] Gabriel KERNEIS et Juliusz CHROBOCZEK. *Lambda-lifting and CPS conversion in an imperative language*. Rapp. tech. Fév. 2012. URL : https://hal.archives-ouvertes.fr/hal-00669849 (cf. p. 18).