Transformation de fonctions récursives en boucles dans ocamlpro

Élimination des appels récursifs terminaux pour améliorer l'inlining

Fabrice Le Fessant* ****

* *INRIA Saclay – Île-de-France F-91893 Orsay* ** *OCamlPro SARL – Orsay*
fabrice.le_fessant@inria.fr

Résumé. *Dans ce papier, nous présentons une transformation de fonctions récursives terminales en boucles. Cette transformation est spécialement adaptée aux contraintes des représentations intermédiaires du compilateur Objective-Caml, et permet d'étendre le domaine d'application de son* inliner *sans le modifier. Cette transformation s'applique facilement aux fonctions récursives terminales simples, et s'étend aux fonctions mutuellement récursives quand celles-ci forment un graphe réductible. Cette technique a été implantée avec succès dans le compilateur* ocamlpro*, basé sur le compilateur* ocamlopt *d'Objective-Caml.*

Mots-Clés : *compilation,inlining,récursion terminale*

1. Introduction

L'*inlining* est une technique d'optimisation très importante dans les compilateurs, dont dépend l'efficacité maximale des autres techniques d'optimisation. En effet, en insérant le code d'une fonction à l'endroit de son application, l'inlining permet de modifier ce code pour tenir compte des informations supplémentaires disponibles dans le contexte, et souvent de simplifier ce code, parfois dans des proportions considérables, en conduisant à des gains importants en vitesse d'exécution.

L'inlining des fonctions récursives a toujours été un problème difficile dans les langages fonctionnels, où ces fonctions sont souvent utilisées pour exprimer les boucles, structures de contrôle les plus courantes dans les langages impératifs. En effet, *inliner* le code d'une fonction récursive à l'endroit de son appel fait apparaître un autre appel à cette fonction, qu'il conviendrait aussi d'inliner, conduisant à un processus qui, d'une part, ne termine pas, et, d'autre part, ne permet pas de simplification. Certains compilateurs, comme Bigloo [7] ou GHC [2], utilisent notamment deux stratégies pour limiter ce problème :

– **L'élimination des appels terminaux récursifs :** l'élimination des appels terminaux consiste à remplacer, dans la représentation intermédiaire d'un programme, les appels terminaux récursifs en des branchements inconditionnels (voir la figure 1). Cette technique s'applique particulièrement bien quand le langage intermédiaire du compilateur est impératif et non structuré, mais elle est difficilement applicable quand le langage cible est plus structuré.

– **L'inlining des définitions récursives :** l'inlining des fonctions récursives consiste à détecter les fonctions récursives qui présentent un paramètre invariant dans la boucle, et à inliner non pas la fonction à son point d'appel, mais la définition de la fonction quand le paramètre invariant est connu (voir la figure 2).

Nous avons appliqué ces deux stratégies au sein du compilateur natif du langage Objective-Caml[5]. Bien que complexe à implanter, la seconde technique – l'inlining des définitions récursives – ne présente pas de difficulté nouvelle dans ce contexte. Aussi, dans ce papier, nous ne présentons que la mise en œuvre partielle de la première technique – l'élimination des appels récursifs terminaux – dans le contexte particulier du langage intermédiaire de ce compilateur, fortement struturé.

Les appels terminaux sont par définition des appels de fonction, dont la valeur de retour est retournée par la fonction courante. De tels appels sont très intéressants, car le contexte de la fonction appelante ne sera pas utilisé au retour de l'appel, et son emplacement sur la pile peut donc être récupéré immédiatement. Pour une fonction récursive, cela signifie souvent qu'elle s'exécute en taille de pile constante, et ne coûte donc pas plus qu'une boucle impérative. Les fonctions récursives ter-

```
let rec f n x = if n then g (x+1) else h (x-1)
    and g x   = if x mod 2 = 0 then f (x<>4) x else h (x+3)
    and h y   = if y > 10 then f true y else g (2*y)
```

en OCaml devient en C :

```
f:    if(f_n){ g_x=f_x+1; goto g; } else { h_y=f_x-1; goto h;}
g:    if(g_x % 2 == 0){ f_n=(g_x!=4); f_x=g_x; goto f; }
      else { h_y = g_x+3; goto h;}
h:    if(h_y>10){ f_n=1; f_x=h_y; goto f; }
      else { g_x=2*h_y; goto g; }
```

Figure 1 – L'élimination des appels terminaux, dans le cas de trois fonctions mutuellement récursives.

minales sont donc souvent utilisées par les programmeurs fonctionnels pour remplacer les boucles impératives.

Les compilateurs des langages fonctionnels tentent de traiter ces appels avec efficacité : pour cela, comme illustré par la figure 1, ils remplacent les appels récursifs par des branchements inconditionnels. Si le langage intermédiaire du compilateur est peu structuré, cette transformation peut avoir lieu très tôt. Au contraire, et c'est le cas du compilateur `ocamlopt`, si le langage cible reste très structuré – ce qui permet d'effectuer d'autres analyses plus efficacement – cette transformation a lieu très tard, quasiment à la génération de code, et les autres optimisations effectuées dans le compilateur ne peuvent pas en profiter. C'est en particulier le cas pour le changement des représentations des données lors des appels de fonction (unboxing) ou de la détection de l'échappement de certaines valeurs.

La transformation que nous présentons dans ce papier permet, pour certaines fonctions, d'effectuer cette conversion au tout début de la compilation, et donc dans bénéficier dans les phases ultérieures. Pour montrer son efficacité, nous présenterons quelques exemples de code, et parfois l'assembleur correspondant, pour une architecture AMD 64 bits Linux (syntaxe `gas`).

La définition

```
let rec map f = function
  [] -> []
| x :: xs -> (f x) :: map f xs
;;
let succs = List.map
            (fun x -> x+1) list
```

devient :

```
let succs =
  let rec map = function
    [] -> []
  | x :: xs -> (x+1) :: map xs
  in
  map list
```

Figure 2 – L'inlining de définitions, dans le cas de la fonctions `List.map`, après inlining et suppression du paramètre invariant `f`.

2. Rappel sur l'*inlining* dans Objective-Caml

Le compilateur natif d'Objective-Caml effectue peu d'optimisations sur le code qu'il génère. Le choix a été de privilégier la simplicité du compilateur pour garantir sa correction, tout en mettant en œuvre uniquement les optimisations jugées les plus efficaces.

La compilation d'un programme passe par plusieurs Représentations Intermédiaires du Code (*RIC*) :

Parsetree	Arbre de syntaxe non typé
Typedtree	Arbre de syntaxe typé
Lambda	RIC non typée, sans filtrages ni classes
Clambda	RIC après explicitation des fermetures, propagation de constantes et inlining.
Cmm	RIC après remplacement des primitives, simplification des primitives et spécialisation des représentations des valeurs scalaires.
Mach	RIC prenant en compte les spécificités de l'architecture visée (mode d'adressage, opérations).
Linearized	RIC sans structures de contrôle de flots autres que les branchements conditionnels et inconditionnels.

L'inlining est effectué lors de la traduction du `Lambda` code en `Clambda` code, en même temps que la conversion des fermetures, la

```
type lambda =
    Lvar of Ident.t
  | Lconst of structured_constant
  | Lapply of lambda * lambda list * Location.t
  | Lfunction of function_kind * Ident.t list * lambda
  | Llet of let_kind * Ident.t * lambda * lambda
  | Lletrec of (Ident.t * lambda) list * lambda
  | Lprim of primitive * lambda list
  | Lswitch of lambda * lambda_switch
  | Lstaticraise of int * lambda list
  | Lstaticcatch of lambda * (int * Ident.t list) * lambda
  | Ltrywith of lambda * Ident.t * lambda
  | Lifthenelse of lambda * lambda * lambda
  | Lsequence of lambda * lambda
  | Lwhile of lambda * lambda
  | Lfor of Ident.t * lambda * lambda * direction_flag * lambda
  | Lassign of Ident.t * lambda
  | Lsend of meth_kind * lambda * lambda * lambda list
```

Figure 3 – La définition de la représentation intermédiaire du code en `Lambda` code. Contrairement à beaucoup de représentations intermédiaires, celle-ci est structurée, permettant des analyses plus fines et plus rapides, et non sous forme de blocs de code.

propagation de constantes et la spécialisation des applications de fonctions connues en appels directs.

L'inlining est décomposé en deux étapes :

1) **À la définition d'une fonction :** à la définition d'une fonction, le compilateur choisit si la fonction sera systématiquement inlinée. Ce choix prend deux éléments en jeu : (1) la taille de la fonction, qui doit être inférieure à 10 nœuds par défaut, et (2) l'absence de constructions particulières, telles que la récursion, la définition de sous-fonctions ou de constantes structurées. Le corps de la fonction (en `Clambda` code) est alors enregistré dans un environnement particulier, qui est transmis entre modules lors de la compilation séparée.

2) **À l'appel de la fonction :** à l'appel d'une fonction, le compilateur détecte si le site d'appel peut être spécialisé, c'est-à-dire si la fonction appelée est connue et appliquée à la totalité de ses arguments. Si c'est le cas, et que le corps de la fonction est trouvé dans l'environnement, le compilateur procède à l'inlining de la fonction par substitution des valeurs des arguments aux paramètres dans le corps de la fonction.

Une spécificité du compilateur natif d'OCaml est que ces représentations intermédiaires sont structurées . Ainsi, la figure 3 montre que le `Lambda` code contient des constructions telles que `Lwhile`, `Lfor`, `Lifthenelse` ou `Ltrywith`.

La seule construction s'approchant de l'habituel `goto` des langages non structurés est la construction `Lstaticcatch` : elle permet d'échapper au flot de contrôle dans son corps grâce à l'instruction `Lstaticraise`, qui déplace l'exécution vers le handler du `Lstaticcatch` correspondant, tout en passant un certain nombre de valeurs en arguments. Ces constructions peuvent être imbriquées, chaque `Lstaticcatch` étant associé à un label (un entier), qui est utilisé par `Lstaticraise` pour spécifier le `Lstaticcatch` auquel il se réfère. `Lstaticcatch` et `Lstaticraise` offrent donc un fonctionnement identique à celui de `try ... with ...`, mais sans manipulation de la pile, et donc compilable par un simple saut dans le code.

Cette construction est principalement utilisée pour la compilation du filtrage de motifs (*pattern matching*), mais elle est aussi la seule qui puisse être utilisée pour l'élimination des appels récursifs terminaux. Nous allons voir que cela limite cette élimination aux graphes d'appels réductibles.

3. Transformation de récursion simple

Nous allons commencer par expliquer l'élimination des appels terminaux récursifs dans le cas d'une fonction récursive simple, au niveau du `Lambda` code. Nous allons aussi montrer que cette transformation ne change pas fondamentalement le code généré pour la fonction elle-même, car le compilateur élimine déjà les appels terminaux lors de la

Le code de filtrage suivant :

```
let permute cond a b =
  match cond, a , b with
  (true, x, y) | (false, y, x) -> (x,y)
```

est converti dans le lambda-code suivant :

```
(Llet
  (permute
     (Lfunction cond a b
       (Lstaticcatch
         (Lifthenelse (!= cond 0)
             (Lstaticraise 9 a b) (Lstaticraise 9 b a))
        with (9 x y) -> (makeblock 0 x y)))))
```

Figure 4 – La construction Lstaticcatch est en particulier utilisée dans le filtrage de motifs, quand les motifs d'un choix contiennent des variables [3].

transformation du code `Cmm` en code `Mach`, même si cette élimination a lieu trop tard pour permettre l'inlining de ces fonctions.

Un exemple intéressant de fonction récursive que l'on aimerait inliner est la fonction `List.iter` qui permet d'appliquer une fonction à l'ensemble des éléments d'une liste :

```
let rec iter f xs0 =
  match xs0 with
  | [] -> ()
  | x :: xs -> f x; iter f xs
```

Cette fonction est courte, et prend souvent en argument une fonction anonyme, dont l'inlining bénéficierait probablement aux performances du programme. Elle contient un unique appel terminal, qu'il faut transformer à l'aide de l'instruction `Lstaticcatch`. Voici le code de la fonction après notre transformation :

```
let iter f xs0 =
  let xs1 = ref xs0 in
  Lstaticcatch (0) (* catch for return *)
    while true do    (* in case of tailcall, loop *)
      Lstaticcatch (1) (* catch for tailcall *)
        let xs2 = !xs1 in
        Lstaticraise (0) (* if no tailcall, escape to return *)
          (match xs2 with
           | [] -> ()
           | x :: xs -> f x;
              xs1 := xs;
              Lstaticraise (1) ()
          )
      with () -> ()
    done
  with r -> r
```

Comme le montre cet exemple, la transformation requiert d'ajouter deux constructions `Lstaticcatch` dans la fonction :

– **Une construction de fin de boucle :** le premier `Lstaticcatch` dans le code précédent permet de quitter la boucle quand aucune récursion n'a lieu. Le `Lstaticraise` correspondant prend le corps de la fonction comme argument, et retourne donc bien la valeur de la fonction.

– **Une construction de boucle :** le second `Lstaticcatch` permet de revenir au début de la boucle, en cas d'appel terminal récursif. Le `Lstaticraise` correspondant est précédé de l'affectation aux paramètres des arguments de l'appel récursif.

Lors de l'appel terminal, des références (enregistrements modifiables) sont utilisées pour permettre la modification des arguments avant de revenir dans la fonction grâce à la boucle infinie `while true do`.

Deux optimisations supplémentaires permettent de réduire au minimum le coût de cette transformation :

– **La simplification d'invariants :** la transformation détecte quand un appel terminal conserve un argument (l'argument `f` de `List.iter`), et évite en conséquence de modifier la référence correspondante ;

– **La simplification des références :** l'utilisation d'une référence (`ref`) par argument est coûteuse, car elle provoque l'allocation d'une cellule, ainsi qu'une vérification pour le ramasse-miettes à chaque modification. Pour limiter ce coût, le compilateur procède à une simplification des références en variables modifiables (allouées sur la pile) quand celles-ci ne s'échappent pas de la portée de la fonction. Pour améliorer cette simplification, la transformation a ajouté une variable `xs2` qui permettrait, si nécessaire, au paramètre d'échapper de la portée sans que la référence elle-même s'en échappe.

La nécessité de cet ajout est illustré dans le code suivant. Dans le code de la partie gauche, la référence `x` s'échappe dans la fermeture passée en argument à `Array.iteri`. Elle ne peut alors pas être simplifiée par le compilateur. Au contraire, dans la partie droite, la valeur de la référence est remplacée par `xx` dans la fermeture (similairement à `xs2` dans notre cas), et la référence `x` peut donc être simplifiée car elle ne s'échappe plus.

```
let x = ref 0. in
for i = 0 to 10 do
  x := !x +. float i;
  Array.iteri (fun j a ->
    t.(j) <- a +. !x) t;
done
```

```
let x = ref 0. in
for i = 0 to 10 do
  x := !x +. float i;
  let xx = !x in
  Array.iteri (fun j a ->
    t.(j) <- a +. xx) t;
done
```

Ces simplifications permettent d'obtenir, après transformation, un code assembleur extrêmement proche de celui obtenu sans transformation (mais après l'élimination d'appels terminaux qui est effectuée par le compilateur lors de la génération de code dépendant de l'architecture (Mach)) :

```
camlCode__iter1: # avant
    subl    $8, %esp
.L101:
    movl    %eax, %edx
    cmpl    $1, %ebx
    je      .L100
    movl    %ebx, 0(%esp)
    movl    %edx, 4(%esp)
    movl    (%ebx), %eax
    movl    (%edx), %ecx
    movl    %edx, %ebx
    call    *%ecx
    movl    0(%esp), %eax
    movl    4(%eax), %ebx
    movl    4(%esp), %eax
    jmp     .L101
    .align  16
.L100:
    movl    $1, %eax
    addl    $8, %esp
    ret
```

```
camlCode__iter1:  # apres
    subl    $8, %esp
    movl    %ebx, 4(%esp)
    movl    %eax, 0(%esp)
.L101:
    cmpl    $1, %ebx
    je      .L102
    movl    (%ebx), %eax
    movl    0(%esp), %ebx
    movl    (%ebx), %ecx
    call    *%ecx
    movl    4(%esp), %ebx
    movl    4(%ebx), %ebx
    movl    %ebx, 4(%esp)
    jmp     .L101
    .align  16
.L102:
    movl    $1, %eax
    addl    $8, %esp
    ret
```

Bien qu'il soit tentant de comparer l'efficacité de ces deux codes, les différences sont trop subtiles pour engendrer des écarts de performances importants. Ceci nous permet cependant de montrer que notre transformation n'affecte pas, ou peu, le code généré pour la fonction transformée. Par contre, le code `Lambda` après transformation n'est plus récursif, et peut donc être inliné aux sites d'appel par le compilateur pendant la phase suivante d'optimisation.

4. Transformation de fonctions mutuellement récursives

La généralisation de cette transformation en boucle aux fonctions mutuellement récursives est plus complexe. En effet, la transformation présentée en Figure 1 n'est pas possible en utilisant uniquement les

`Lstaticcatch` et `Lstaticraise` (que nous noterons respectivement `catch` et `fail` pour raccourcir les notations). Comme nous avons fait le choix de ne pas modifier le langage intermédiaire du compilateur, notre algorithme est donc restreint aux cas pouvant être encodés au moyen de la construction `Lstaticcatch`.

Une alternative serait de simplement encoder les fonctions mutuellement récursives avec un sélecteur de fonction :

```
while true do
  catch (0)
    match !function with
     Fun_f -> ...
   | Fun_g -> ...
   | Fun_h -> ...
  with ()
done
```

Hélas, un tel encodage n'est pas possible à cause des arguments que prennent les fonctions :

Le passage des arguments dans le sélecteur n'est pas possible, car cela provoquerait l'allocation du sélecteur pour pouvoir y placer les arguments. Or, l'efficacité des appels terminaux est un principe des langages fonctionnels, qui implique l'absence d'allocation dans le tas ;

L'utilisation de références mutables n'est pas possible, car (1) lors de l'appel de la première fonction, les références aux arguments des autres fonctions ne seraient pas initialisées et (2) la pré-initilisation avec de fausses valeurs ou le partage des arguments (utiliser la même variable pour le premier argument de chaque fonction) briserait certaines hypothèses de typage (une variable pourrait alors contenir aussi bien des entiers que des flottants, conduisant à un problème lors de l'allocation de registre pour cette variable).

La technique que nous avons développée pour transformer les fonctions mutuellement récursives en boucles avec `Lstaticcatch` est la suivante :

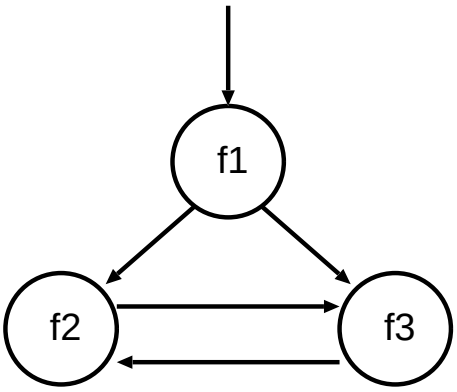
1. Découverte de la fonction principale : nous commençons par étudier quelles fonctions sont utilisées à l'extérieur de la définition. Notre algorithme refuse de continuer si plus d'une seule fonction est utilisée à l'extérieur, ou s'échappe de la portée. Nous appellerons cette fonction la *fonction principale*.

2. Extraction du graphe d'appel : nous calculons pour chaque fonction f, l'ensemble $callers(f)$ des fonctions qui l'appellent, et l'ensemble $callees(f)$ des fonctions qu'elle appelle, parmi les fonctions mutuellement récursives de la définition courante. Ce calcul est généralement indécidable, mais l'hypothèse précédente (une seule fonction s'échappe de la définition) le rend rend décidable dans notre cas.

3. Suppression des récursions locales : nous maintenons l'invariant, pendant tout l'algorithme, que $f \notin callees(f)$ et $f \notin callers(f)$, en supprimant les liens d'un noeud du graphe vers lui-même. Nous imposons aussi que $callees(f) = \emptyset$ si f est la fonction principale.

4. Fusion des noeuds : quand un noeud du graphe est appelé uniquement par un autre noeud du graphe, nous fusionnons ces deux noeuds, en mémorisant quel noeud est appelé par l'autre.

L'algorithme se termine quand aucune fusion n'est possible sur l'ensemble du graphe. La transformation est possible quand le graphe final est réduit à un seul noeud, correspondant au noeud de la fonction principale. Cet algorithme de suppression des boucles locales et de fusion des noeuds est bien connu : il permet aussi de détecter quand un graphe est *réductible* [1]. Cela montre que notre transformation s'applique à tous les graphes de fonctions mutuellement récursives qui sont réductibles, et uniquement à ceux-là.



Exemple de code menant à un graphe non réductible (dit graphe canonique).

```
let rec f1 cond x =
  if cond then f2 x else f3 x
and f2 x = f3 (x+1)
and f3 x = f2 (x-1)
```

En effet, il est impossible d'imbriquer les fonctions les unes dans les autres de telle sorte que le père puisse être appelé de tous ses fils, mais qu'un fils ne soit appelable que de son père direct.

4.1. *Spécification de l'algorithme*

Nous décrivons maintenant notre algorithme de manière un peu plus formelle : soit G le graphe des fonctions, $body(f)$ le code (en lambda-code) de la fonction $f \in G$. Dans un soucis de simplification, nous ne produirons pas dans les exemples suivant le code associé à la gestion des arguments, i.e. les modifications des variables dans lesquelles sont passées les arguments, ce code se déduisant trivialement de l'encastrement des constructions `catch ... with`.

Nous définissons la *finition* d'une fonction $f \in G$, notée $[|f|]$, le code suivant :

`while true do`
  `catch` $restart_f$
    `fail`$(return, body(f))$ où
                $f\ x_1\ ...\ x_n$ est remplacé par `fail`$(restart_f, (x_1, ..., x_n))$
  `with` $(x_1, ..., x_n) \rightarrow$
`done`

Nous appliquons ensuite itérativement l'algorithme suivant :

Tant que $\exists f \in G$ tel que $callers(f) = \{g\}$, alors

$G \leftarrow G \backslash \{f\}$

$body(g) \leftarrow$

```
catch call_f
    body(g) où f x_1 ... x_n est remplacé par
                                  fail(call_f, (x_1, ..., x_n))
with (x_1, x_2, ..., x_n) →
    [|f|]
```

$callees(g) \leftarrow callees(g) \cup callees(f)$

$\forall h \in callees(f), callers(h) \leftarrow callers(h) \backslash \{f\} \cup \{g\}$

Lorsque cet algorithme termine, deux cas sont possibles :

– Si $G = \{f\}$ où f est la fonction principale, alors nous définissons la fonction f comme une fonction non récursive dont le corps est :

```
catch return
  [| f |]
with r → r
```

– Sinon, la transformation échoue, et les fonctions sont compilées ne manière récursive.

Le lecteur expert des algorithmes de graphe remarquera que cet algorithme n'est pas optimal en complexité pour réduire un graphe réductible. Néanmoins, les définitions mutuellement récursives définissant rarement plus de 4 ou 5 fonctions, la simplicité de cet algorithme est plus importante que sa complexité.

4.2. *Expérimentation*

Bien que la limitation de notre algorithme aux seuls graphes de fonctions réductibles puisse sembler importante, cette limitation est en pratique peu contraignante :

– D'une part, elle ne s'applique pas sur la majorité des cas qui nous intéressent, i.e. les fonctions simplement récursives ;

– D'autre part, les fonctions mutuellement récursives sont souvent utilisées pour encoder des automates, en particulier ceux d'analyseurs lexicaux ou grammaticaux rudimentaires. Ces automates forment souvent des graphes réductibles, sur lesquels notre transformation fonc-

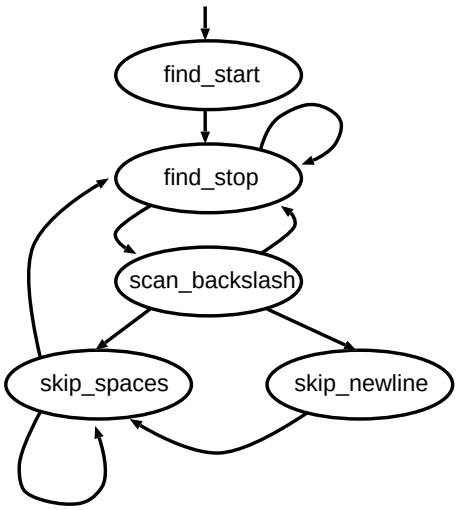


Figure 5 – Le graphe d'appel de la fonction `find_start` du module `Scanf` de la bibliothèque `stdlib` d'Objective-Caml. Cette définition mutuellement récursive est transformée en boucles par notre algorithme, car le graphe est réductible.

tionne parfaitement.

Pour illustrer ce dernier point, nous pouvons prendre l'exemple de la fonction principale `find_start` du module `Scanf` de la bibliothèque standard d'OCaml. Cette fonction survole une chaîne de caractères commençant par des guillements jusqu'aux guillements finaux, en interprétant correctement les caractères d'échappement. Le graphe d'appel de cette fonction est présenté sur la Figure 5, et le code obtenu après transformation est présenté dans une version simplifiée en Figure 6. Il est facile de vérifier que l'encastrement des constructions `catch ... with` permet bien le graphe d'appel correspondant.

Le tableau suivant donne une idée de l'efficacité de notre transformation sur les sources de la distribution Objective-Caml :

	Fonction Récursive			Mutuellement Récursives		
	Succès	Échecs	Taux Succès	Succès	Échecs	Taux Succès
`stdlib`	112	140	44%	10	9	52 %
`ocamlopt`	159	224	41%	17	72	18 %

```
catch (return)
while true do
  catch (restart_find_start_1456)
    catch (call_find_stop_1457)
      BODY(find_start_1456)
    with call_find_stop_1457 ->
      while true do
        catch (restart_find_stop_1457)
          catch (call_skip_spaces_1460)
            catch (call_skip_newline_1459)
              catch (call_scan_backslash_1458)
                BODY(find_stop_1457)
              with call_scan_backslash_1458 ->
                while true do
                  catch (restart_scan_backslash_1458)
                    BODY(scan_backslash_1458)
                  with restart_scan_backslash_1458 -> ()
                done
            with call_skip_newline_1459 ->
              while true do
                catch (restart_skip_newline_1459)
                  BODY(skip_newline_1459)
                with restart_skip_newline_1459 -> ()
              done
          with call_skip_spaces_1460 ->
            while true do
              catch (restart_skip_spaces_1460)
                BODY(skip_spaces_1460)
              with restart_skip_spaces_1460 -> ()
            done
        with restart_find_stop_1457 -> ()
      done
  with restart_find_start_1456 -> ()
done
with return (r) -> r
```

Figure 6 – Une version simplifiée du code généré pour la fonction `find_start` dont le graphe d'appel est présenté en Figure 5.

Le taux de succès sur les fonctions récursives simples (plus de 40%) est nettement supérieur à celui sur les fonctions mutuellement récursives, sauf dans le cas de la bibliothèque standard (`stdlib`) où ces fonctions sont beaucoup plus simples.

4.3. *Impact sur les autres optimisations*

Il est raisonnable de s'interroger sur l'intérêt de transformer des fonctions mutuellement récursives en boucles, sachant que la fonction générée va probablement dépasser le seuil de taille pour l'inlining (10 nœuds par défaut), et qu'en conséquence, elle ne sera donc jamais inlinée. L'intérêt est pourtant réel, car le compilateur effectue des optimisations importantes sur les boucles, plus difficiles à effectuer sur les appels de fonctions, en particulier pour simplifier la représentation des données.

Ainsi, la fonction `sum_i` suivante, qui effectue la somme des éléments d'une liste d'entiers avec une référence mutable et une récursion, est compilée par la version actuelle du compilateur en allouant une référence, qui est ensuite placée dans la fermeture de la sous-fonction `iter`, elle-même allouée à chaque appel.

La fonction `sum_i` qui calcule la somme des éléments d'une liste d'entiers :

```
let sum_i list =
  let n = ref 0 in
  let rec iter l =
    match l with
      [] -> !n
    | x :: tail ->
        n := !n + x;
        iter tail
  in
  iter list
```

est compilée dans le code assembleur suivant :

Nouveau code généré :

```
camlCode__sum_i_1031:
.L102:
        movq  %rax, %rbx
        movq  $1, %rax
.L101:
        cmpq  $1, %rbx
        je    .L100
        movq  (%rbx), %rdi
        leaq  -1(%rax, %rdi),
                  %rax
        movq  8(%rbx), %rbx
        jmp   .L101
.L100:
        ret
```

Le compteur est stocké dans le registre `%rax`.

Code d'origine :

```
camlCode__sum_i_1030:
        subq  $8, %rsp
.L102:
        movq  %rax, %rsi
.L103:  subq  $48, %r15
        movq  caml_young_limit,
                   %rax
        cmpq  (%rax), %r15
        jb    .L104
        leaq  8(%r15), %rdi
        movq  $1024, -8(%rdi)
        movq  $1, (%rdi)
        leaq  16(%rdi), %rbx
        movq  $3319, -8(%rbx)
        movq  camlCode__iter_1033,
                   %rax
        movq  %rax, (%rbx)
        movq  $3, 8(%rbx)
        movq  %rdi, 16(%rbx)
        movq  %rsi, %rax
        addq  $8, %rsp
        jmp   camlCode__iter_1033
.L104:  call  caml_call_gc
.L105:  jmp   .L103
```

Après transformation en boucle, le code assembleur ne contient plus de référence, mais un simple compteur dans un registre. Ni la référence, ni la fermeture ne sont allouées. Le code est difficilement plus optimisable. L'application de cette fonction 10,000,000 de fois à une liste de longueur 100 prend 3.25 secondes avec la version d'origine, 2.02 secondes avec la version transformées.

L'*unboxing* des flottants[6, 4] peut aussi bénéficier de la transformation en boucle : le compilateur natif d'OCaml ne spécialise pas les conventions d'appel en fonction des types des arguments. Il impose donc que les flottants, par exemple, soient toujours alloués (*boxés*),

quand ils sont passés en argument d'une fonction, comme ils le seraient dans le cas d'une fonction polymorphe.

Ainsi, dans la fonction `sum_f` suivante, qui teste si la somme des éléments d'une liste de flottants est positive, et qui est écrite en pur style fonctionnel, le résultat de la somme `sum +. x` est alloué à chaque itération de la boucle (nous ne présentons pas ce code, trop complexe, dans l'exemple).

La fonction `sum_f` vérifiant qu'une somme est positive :

```
let sum_f list =
  let rec iterf sum l =
    match l with
      [] -> sum > 0.
    | x :: tail ->
       iterf (sum +. x) tail
  in
  iterf 0. list
```

est compilée après élimination de la récursion terminale en :

```
camlCode__sum_f_1038:
        xorpd   %xmm1, %xmm1  # %xmm1 := 0.
.L113:
        cmpq    $1, %rax
        je      .L116
        movq    (%rax), %rbx
        addsd   (%rbx), %xmm1 # %xmm1 := %xmm1 +. [%rbx]
        movq    8(%rax), %rax
        jmp     .L113
.L116:
        xorpd   %xmm0, %xmm0  # %xmm0 := 0
        comisd  %xmm0, %xmm1  # %xmm1 > %xmm0 ?
        jbe     .L115
        movq    $1, %rax      # %rax := 1
        jmp     .L114
.L115:
```

```
        xorq    %rax, %rax      # %rax := 0
.L114:                          # %rax := boxed_int(%rax)
        leaq    1(%rax, %rax), %rax
        ret
```

Après transformation, le code assembleur montre qu'aucune allocation n'a lieu. Le compilateur a pu détecter que les références (utilisées pour passer les arguments à la place des appels récursifs) ne s'échappaient pas du corps de la fonction, et a pu les remplacer par de simples registres flottants. Le gain en performance est alors extrêmement important.

Nous présentons maintenant quelques mesures de temps sur ces deux fonctions. La fonction est appelée N fois, sur une liste de longueur M. La seule optimisation supplémentaire effectuée par ocamlpro par rapport à ocamlopt est celle décrite dans ce papier.

Fonction	N (liste)	M (loop)	Temps (s) ocamlopt	Temps (s) ocamlpro
`sum_i`	100	10,000,000	3.25	2.02
`sum_i`	1,000	1,000,000	2.90	1.91
`sum_i`	10,000	100,000	3.48	2.87
`sum_i`	100,000	10,000	3.55	3.00
`sum_i`	1,000,000	1,000	5.54	5.52
`sum_f`	100	10,000,000	4.97	2.05
`sum_f`	1,000	1,000,000	5.15	4.07
`sum_f`	10,000	100,000	5.11	4.05
`sum_f`	100,000	10,000	6.47	5.48
`sum_f`	1,000,000	1,000	9.33	9.06

Ce tableau montre que la transformation sur `sum_i` est efficace, mais la différence diminue quand le nombre d'appels à la fonction diminue. En effet, la différence entre les deux fonctions est principalement l'allocation de la fermeture et de la référence lors du premier appel, les appels récursifs ayant des coûts équivalents. Au contraire, pour `sum_f`, la différence provient à la fois de la non-allocation de la fermeture au premier appel, mais aussi de la non-allocation du paramêtre `sum` lors des appels récursifs.

5. Conclusion

Dans ce papier, nous avons présenté un algorithme permettant de transformer une définition de fonctions mutuellement récursives en une seule fonction non récursive, en utilisant uniquement les constructions disponibles dans le langage intermédiaire du compilateur Objective-Caml.

Cet algorithme ne peut fonctionner que quand tous les appels récursifs sont terminaux, qu'aucune fonction (sauf une) ne s'échappe de la portée de la définition, et que le graphe d'appel des fonctions est réductible — ce qui, en pratique, est le cas pour les fonctions qui nous intéressent.

Cette transformation est particulièrement utile, car elle permet d'inliner certaines fonctions de la bibliothèque standard sans modifier la passe d'inlining du compilateur Objective-Caml, comme la fonction `List.iter`, très fréquemment utilisée. Elle permet aussi d'augmenter les opportunités d'autres optimisations, comme la suppression des références, la suppression des fermetures allouées localement, ou le boxing des flottants.

Références

[1] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21 :367–375, July 1974.

[2] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. In *Journal of Functional Programming*, page 2002, 1999.

[3] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01 : Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 26–37, New York, NY, USA, 2001. ACM.

[4] Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL '92 : Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188, New York, NY, USA, 1992. ACM.

[5] Xavier Leroy. *The Objective Caml System : Documentation and User's Manual*, 1996. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

[6] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[7] Manuel Serrano. Inline expansion : When and how ? In *PLILP '97 : Proceedings of the9th International Symposium on Programming Languages : Implementations, Logics, and Programs*, pages 143–157, London, UK, 1997. Springer-Verlag.