OCamlCC - Traduire OCaml en C en passant par le bytecode

Michel Mauny, Benoît Vaugon

OCamlCC – Traduire OCaml en C en passant par le bytecode

M. Mauny & B.Vaugon

Unité d'Informatique et d'Ingéniérie des Systèmes,
Équipe Sûreté et Fiabilité des Logiciels,
ENSTA-ParisTech,
828, boulevard des Maréchaux,
F-91762 Palaiseau Cedex
{prénom.nom}@ensta-paristech.fr

Résumé

Nous présentons des résultats préliminaires concernant OCamlCC, un compilateur de programmes OCaml en code natif basé sur la traduction en C des exécutables bytecode.

1. Introduction

Le système OCaml [5] est doté à la fois d'un générateur de bytecode, portable sur une grande variété d'architectures et d'un générateur de code natif ciblant les architectures les plus communes.

Il peut toutefois être frustrant de ne pas pouvoir produire du code natif OCaml pour des architectures peu communes, qu'elles soient trop anciennes, trop récentes ou trop rares, alors qu'elles sont dotées de compilateurs performants pour d'autres langages. Vient alors l'idée naturelle — et déjà ancienne [3, 7, 8, 9] — de traduire les programmes OCaml en des programmes écrits dans des langages dotés de compilateurs performants. La bibliothèque d'exécution OCaml étant elle-même écrite en C, ce langage est un candidat de choix, sachant la vaste disponibilité des compilateurs C (GCC, CLang, ICC, pour n'en citer que trois parmi les plus connus).

Bien sûr, l'idée n'est pas nouvelle et il existe ou a existé quelques compilateurs [3, 7, 8, 9] produisant du C à partir de langages fonctionnels.

Cependant, notre traduction d'OCaml vers C présente l'originalité de prendre comme point de départ le bytecode OCaml et d'utiliser la bibliothèque d'exécution standard d'OCaml, et donc en particulier son gestionnaire mémoire, ce qui fournit une complète compatibilité avec les compilateurs de la distribution OCaml.

2. Le bytecode, un langage source

Comme l'ont déjà remarqué J. Vouillon et V. Balat dans la conception de leur traducteur d'OCaml en JavaScript [11], le bytecode OCaml représente un bon point de départ pour de telles opérations : sa conception est stable et les exécutables bénéficient des opérations réalisées par le compilateur OCaml telles l'édition de liens et la collection des constantes. La conception d'outils prenant ces exécutables comme point de départ ne nécessite donc généralement pas de modifier la chaîne de compilation d'OCaml, simplifiant drastiquement leur évolution et leur maintenance.

La génération de code C à partir d'exécutables bytecode OCaml représente ainsi une séquence d'étapes de compilation qui prolonge strictement la production de bytecode OCaml. Cette génération de C et sa compilation peuvent être utilisées comme une étape terminale du processus de développement. Elle permet aussi la distribution de logiciels sous la forme de bytecode portable, laissant au « client » le soin d'en produire du code natif pour sa ou ses architectures matérielles.

3. Traduction du bytecode en C

Grossièrement, l'exécution du bytecode OCaml consiste en l'interprétation d'instructions agissant sur des registres et une pile « virtuels », c'est-à-dire émulés par des variables et un tableau C. Ces éléments stockent les arguments et les résultats intermédiaires de calculs, et servent aussi de racines du graphe de la mémoire accessible, qui doit être parcourue périodiquement par le gestionnaire mémoire.

La traduction du bytecode peut prendre plusieurs formes, allant de la macro-expansion de chacune des instructions jusqu'à une véritable dé-compilation dans le but d'obtenir un programme C structuré. Nous décrivons ci-dessous une première version d'un générateur naïf de C, et détaillons ensuite les mécanismes mis en œuvre qui nous permettent d'obtenir du C structuré et apte à être optimisé par les compilateurs C.

3.1. Macro-expansion du bytecode

La génération de C (de bas niveau) à partir du bytecode OCaml peut être quasi-triviale : il suffit de traduire (par un `#define`) chaque instruction de bytecode en le fragment de code C qui la met en œuvre dans l'interprète. C'est en fait la technique utilisée par la première version d'OCamlCC, réalisée par le second auteur, et, pour tout dire, ce fut là le point de départ du projet OCamlCC.

Une telle traduction présente un défaut essentiel : le programme C résultant est essentiellement constitué d'une seule fonction dont la taille peut pousser les compilateurs C au-delà de leurs limites. Par exemple, la fonction C émanant de la traduction du bytecode de la commande `ocamlc` dépasse les 10^6 instructions, et GCC est dans l'impossibilité de la compiler sur certaines machines [1].

Cela dit, la macro-expansion des instructions de bytecode en C présente aussi quelques caractéristiques intéressantes. Comme avec l'interprète de bytecode, les appels terminaux n'utilisent naturellement pas d'espace de pile, aucun changement de la bibliothèque d'exécution OCaml n'est nécessaire et les performances sont encourageantes : l'exécution est de 2 à 7 fois plus rapide que celle du bytecode interprété, et le code natif engendré par `ocamlopt` est lui-même de 2 à 8 fois plus rapide que le bytecode macro-expansé en C. La taille des exécutables, de 1,6 à 2 fois plus importante que celle de ceux produits par `ocamlopt`, reste modérée.

3.2. Dé-compilation

Le C obtenu par une telle macro-expansion n'est pas seulement volumineux, il est aussi marqué par un style de programmation fait de sauts calculés et d'accès à toutes les valeurs au travers d'un tableau global (la « pile virtuelle »), qui n'est pas toujours favorable à l'optimisation par le compilateur C.

Afin d'obtenir un meilleur style, OCamlCC isole les corps des fonctions (λ-abstractions) dans le bytecode et engendre une fonction C par λ-abstraction OCaml. Le prototype de chaque fonction ainsi produite est : `value f(value env, ...);`, où `value` est le type C des représentations des valeurs OCaml et où le paramètre `env` représente la partie environnement des fermetures construites à partir du code de `f`.

1. Par exemple avec GCC 4.7.2 / Linux 64 bits, sur un Intel Core i7 doté de 12 Go de mémoire vive.

Maintenant que nous avons des fonctions C correspondant aux abstractions OCaml du programme original, nous optimisons le code des corps des fonctions en :

- changeant certains appels indirects (*via* une fermeture) en des appels directs à la fonction C correspondant au code de la fermeture, lorsqu'il est possible de déterminer statiquement la fonction appelée,
- procédant à une sorte de propagation des pointeurs de fonction, transportant les composants des fermetures depuis leur lieu de création vers leur site d'usage,
- évitant la construction de fermetures inutilisées,
- supprimant le paramètre `env` lorsqu'il est inutile,
- et en procédant à l'extraction de certaines valeurs (par exemple les valeurs non allouées) de la pile de l'interprète de bytecode OCaml.

Nous ne procédons pas à une construction du graphe de flot de contrôle (CFA) mais plutôt à une propagation de valeurs. Ces optimisations, à l'exception de la dernière, sont classiques et sont toutes réalisées à l'aide de techniques relevant de l'interprétation abstraite [4].

Dans l'état actuel de notre implantation, nous analysons et traduisons des programmes complets et nos analyses statiques n'ont donc pas à être modulaires. À l'évidence, si notre schéma de compilation ne passait pas à l'échelle, nous adopterions une autre architecture, soit à base de compilation séparée (par exemple en produisant des fichiers C à partir des fichiers `.cmo` d'OCaml), soit à l'aide d'autres techniques (par exemple en utilisant une technique de cache pour compiler les programmes C que nous produisons).

La dernière des optimisations mentionnées ci-dessus vise à transférer des valeurs OCaml depuis la pile de l'interprète de bytecode vers des variables C. Cette technique est présentée dans la section suivante.

4. De la pile OCaml aux variables C

4.1. Motivations

Que l'on utilise l'interprète de bytecode `ocamlrun` ou la macro-expansion, la pile d'exécution contenant les résultats intermédiaires de calcul est simulée par le tableau C déjà mentionné. Ce tableau est l'une des racines du gestionnaire mémoire : il est parcouru et mis à jour lors de chaque passage du ramasse-miettes. Ce tableau est pointé par un ensemble de variables globales, et c'est l'existence de ces pointeurs globaux qui empêche le compilateur C de faire la plupart des optimisations classiques car il doit en permanence maintenir l'intégrité du contenu de ce tableau.

Pour faciliter les optimisations du compilateur C, on souhaiterait stocker certaines valeurs OCaml dans des variables C plutôt que dans ce tableau. Le programme ne peut cependant pas maintenir de copie locale de n'importe quel élément de la pile OCaml. En effet, maintenir une copie d'un pointeur sur un bloc OCaml fait courir le risque de se retrouver avec un pointeur invalide si le bloc qu'il référence est déplacé par le gestionnaire mémoire.

Remarquons que nous ne pouvons considérer ni la pile du processus, ni les registres du processeur, comme des racines du gestionnaire mémoire. En effet, les racines du gestionnaire mémoire doivent vérifier un ensemble d'invariants qui sont invalidés par certaines optimisations des compilateurs C. Par exemple, aucune racine (et aucun emplacement mémoire accessible depuis une racine) ne doit contenir de pointeur au milieu d'un bloc représentant un tableau OCaml. Considérons alors un algorithme effectuant le parcours d'un tableau en repartant systématiquement du début du tableau : la plupart des compilateurs C optimiseront ce code en maintenant des pointeurs au milieu du tableau pour en optimiser les accès.

Enfin, si la pile OCaml grossit beaucoup (par exemple lors de l'exécution de nombreux appels imbriqués non terminaux), le tableau simulant cette pile peut devoir être ré-alloué. Ces ré-allocations

potentielles compliquent beaucoup les accès à la pile car ils doivent systématiquement être effectués de manière indirecte, c'est à dire en passant par une variable globale.

Il peut cependant être frustrant de stocker dans la pile OCaml des valeurs ne nécessitant pas d'être manipulées par le gestionnaire mémoire. C'est par exemple le cas pour les entiers, les constructeurs constants, ou des valeurs allouées dont on peut prouver qu'aucun gestionnaire mémoire ne sera lancé pendant qu'on les manipule. Cette optimisation consiste donc à stocker ces valeurs dans des variables locales C plutôt que dans la pile OCaml.

4.2. Algorithme

L'algorithme décrit ici est utilisé après le découpage du bytecode en fonctions, et après toutes les autres optimisations, en particulier la propagation des pointeurs de fonctions. Il fonctionne en effet beaucoup mieux lorsque les appels se font à des fonctions connues plutôt qu'en passant par des fermetures (comme c'est toujours le cas dans le bytecode).

Cet algorithme est appliqué sur chaque fonction. Son but est de déterminer le maximum d'éléments de l'espace de pile OCaml utilisé par la fonction pouvant être transformés en variables locales C, et ce, de manière sûre.

Il s'agit bien entendu d'un problème indécidable. En effet, il est en particulier impossible d'écrire un algorithme pouvant systématiquement (et statiquement) déterminer si une variable sera effectivement dé-référencée à l'exécution. Pour approcher de manière performante la solution optimale de ce problème, on se base sur l'heuristique suivante : on considère qu'un élément de la pile OCaml **peut** être « transformée » en une variable locale C s'il vérifie l'une des quatre conditions suivantes :

- Il ne peut pas être écrit comme un pointeur (plus précisément, il n'existe aucun chemin d'exécution dans lequel on y stocke l'adresse d'un bloc mémoire ou une valeur inconnue, comme par exemple le résultat d'un appel à une fonction inconnue).
- Il ne peut pas être lu comme un pointeur (dé-référencé, passé en argument d'une fonction inconnue, ...).
- Il existe une instruction qui le manipule explicitement comme un entier, c'est à dire qu'il est l'opérande d'une opération entière (+, −, ..., indice lors de l'accès à un tableau, ...)
- Aucun GC ne peut être lancé pendant sa durée de vie (à l'occasion d'une allocation, lors de l'appel à une fonction inconnue, ...).

Nous avons implanté cette heuristique en faisant simplement deux passes sur le bytecode :

1. La première passe consiste à associer un identifiant à chaque résultat intermédiaire, à la manière de la mise sous forme *A-normale* d'un programme fonctionnel. Sous cette forme, chaque identifiant correspond à une valeur qui devra être stockée soit dans la pile OCaml, soit dans une variable locale C.

   Le résultat de cette analyse est donc, pour chaque instruction bytecode, l'attribution d'un identifiant pour l'accumulateur et d'un identifiant pour chaque case de la pile courante.

   Nous effectuons cette analyse par interprétation abstraite du bytecode. Chaque `PUSH` est interprété comme l'ajout d'un identifiant *frais* sur le sommet de la pile. Chaque calcul (opération arithmétique, appel non terminal de fonction, etc.) est interprété comme le dépilement éventuel d'identifiants (correspondant aux arguments) et la création d'un identifiant frais dans l'accumulateur (pour le résultat). Enfin, lorsque deux chemins d'exécution se rejoignent, on *unifie* les identifiants des accumulateurs et des piles, case par case.

2. La seconde passe consiste à déterminer, pour chaque identifiant, s'il correspond à une case de la nouvelle pile OCaml ou s'il peut être extrait et transformé en une variable C.

   Pour chaque instruction bytecode, on marque chacun des identifiants correspondant aux valeurs qu'elle manipule comme étant « lu comme un pointeur », « écrit comme un pointeur » ou

« manipulé comme un entier » en fonction de l'opération effectuée. Lorsqu'une instruction peut déclencher le lancement d'un GC (comme par exemple une allocation ou un appel à une fonction inconnue), on marque tous les identifiants correspondant à la pile courante et à l'accumulateur courant comme étant « vivants lors du lancement potentiel d'un GC ».

La synthèse de ces marquages nous permet de déterminer pour chaque identifiant s'il correspondra à une variable C ou à une case de la pile OCaml.

4.3. Formalisation

La deuxième passe de l'algorithme décrit ci-dessus est intéressante à formaliser car cela permet de se convaincre qu'elle n'engendrera pas de code faux. Plus précisément, on souhaite montrer qu'on n'extrait jamais *à tort* une case de la pile OCaml pour la transformer en variable C.

4.3.1. Définition des identifiants non extractibles

Commençons par décrire formellement la deuxième passe de l'algorithme. On souhaite construire sept ensembles :

- i : l'ensemble des identifiants dont on sait qu'ils correspondent à des entiers
- i_r : l'ensemble des identifiants que l'on lit comme des entiers
- i_w : l'ensemble des identifiants dans lesquels on écrit un entier
- p_r : l'ensemble des identifiants dont on dé-référence le contenu ou dont le contenu est donné à du code inconnu
- p_w : l'ensemble des identifiants que l'on affecte avec un pointeur ou le résultat d'un calcul que l'on ne connaît pas
- mv : l'ensemble des copies d'un identifiant vers un autre
- gc : l'ensemble des identifiants vivants lors du lancement d'un recyclage de la mémoire

On considère une fonction de transition qui, pour chaque instruction du bytecode, et pour un état d'accumulateur et de pile avant l'exécution de cette instruction (noté (a^-, s^-)) et pour un état d'accumulateur et de pile après l'exécution de l'instruction (noté (a^+, s^+)) détermine l'évolution des sept ensembles mentionnés précédemment.

Voici la définition de cette fonction pour quelques instructions représentatives du bytecode OCaml, et où les paramètres a^-, s^-, a^+ et s^+ sont laissés implicites. Chacune de ces instructions change les ensembles i, i_r, i_w, p_r, p_w, mv, gc en :

`ACC` n : i, i_r, i_w, p_r, p_w, $\{(s^-[n], a^+)\} \cup mv$, gc
Copie un élément de la pile vers l'accumulateur.
Remarque : aucun identifiant n'est ni lu ni écrit, seul un couple est stocké dans l'ensemble mv.

`PUSH` : i, i_r, i_w, p_r, p_w, $\{(a^-, s^+[0])\} \cup mv$, gc
Ajout au sommet de la pile du contenu de l'accumulateur.
Remarque : semblable à `ACC`.

`POP` n : i, i_r, i_w, p_r, p_w, mv, gc
Suppression de n cases du sommet de la pile.
Remarque : aucun ensemble n'est modifié puisqu'aucune valeur n'est accédée.

`CONST _` : $\{a^+\} \cup i$, i_r, $\{a^+\} \cup i_w$, p_r, p_w, mv, gc
Affectation de l'accumulateur avec une constante entière.

Remarque : on ajoute a^+ dans l'ensemble i car on est sûr que c'est un entier.

`ADDINT` : $\{a^-, s^-[0], a^+\} \cup i$, $\{a^-, s^-[0]\} \cup i_r$, $\{a^+\} \cup i_w$, p_r, p_w, mv, gc
Addition de l'accumulateur et du sommet de la pile avec résultat dans l'accumulateur.
Remarque : les paramètres et le résultat sont obligatoirement des entiers, d'où l'ajout à i.

`MAKEBLOCK` n`, _` : i, i_r, i_w, $\{a^-\} \cup s^-[0 \,..\, n-2] \cup p_r$, $\{a^+\} \cup p_w$, mv, $\{a^-\} \cup s^- \cup gc$
Allocation d'un bloc dans le tas.
Remarque : l'allocation pouvant provoquer un recyclage de la mémoire, les identifiants correspondant à l'accumulateur et au contenu de la pile sont stockés dans l'ensemble gc.

`GETFIELD _` : i, i_r, i_w, $\{a^-\} \cup p_r$, $\{a^+\} \cup p_w$, mv, gc
Lecture d'un champ du bloc stocké dans l'accumulateur avec résultat dans l'accumulateur.
Remarque : le bloc est dé-référencé et une valeur pouvant être un pointeur est écrite dans le nouvel accumulateur.

`CCALL` n`, _` : i, i_r, i_w, $\{a^-\} \cup s^-[0 \,..\, n-2] \cup p_r$, $\{a^+\} \cup p_w$, mv, $s^+ \cup gc$
Appel d'une fonction C inconnue.
Remarque : par convention, les arguments de la fonction C doivent être sauvegardés par la fonction elle-même. Il n'y a donc pas besoin de marquer les identifiants correspondants comme « vivants lors du lancement potentiel d'un GC ». Seul s^+ est alors ajouté à gc. Notons que s^+ est exactement s^- sans les $n-1$ derniers arguments de la fonction C et que cette dernière n'accède, dans la pile, qu'à ses arguments.

`APPLY` n : i, i_r, i_w, $\{a^-\} \cup s^-[0 \,..\, n-1] \cup p_r$, $\{a^+\} \cup p_w$, mv, $s^+ \cup gc$
Appel d'une fonction OCaml inconnue via sa fermeture.
Remarque : semblable à l'appel d'une fonction C inconnue.

`BRANCHIF _` : i, $\{a^-\} \cup i_r$, i_w, p_r, p_w, mv, gc
Saut conditionnel dépendant de si l'accumulateur vaut `0`.
Remarque : l'accumulateur est lu superficiellement mais pas dé-référencé. Attention, ce n'est pas obligatoirement un entier pour autant, le `BRANCHIF` peut très bien provenir de la compilation d'un test d'égalité avec `None`.

Il faut alors corriger les ensembles i, i_r, i_w, p_r et p_w vis-à-vis des copies entre identifiants représentées par l'ensemble mv. Intuitivement, on souhaite par exemple déduire que, si le contenu d'une variable v_1 est copié dans une variable v_2, puis si v_2 est utilisée dans une opération arithmétique sur des entiers, alors v_1 est un entier. De même, si le contenu de v_1 est copié dans v_2, puis v_2 est lue comme un pointeur, alors v_1 doit être considérée comme (indirectement) lue comme un pointeur.

Pour ce faire, on définit F comme étant la fermeture d'un ensemble vis-à-vis d'un ensemble de couples :

$$F(A, X) \triangleq \{y \mid \exists x \in X, (x, y) \in A\}$$

Notons F^* la fermeture réflexive transitive de F vis-à-vis de son second paramètre, et mv^{-1} l'ensemble des couples de mv dont on a permuté les composants. Nous pouvons alors définir les cinq ensembles suivants :

$$
\begin{array}{lcl}
I & \triangleq & F^*(mv \cup mv^{-1}, i) \\
I_R & \triangleq & F^*(mv^{-1}, i_r) \\
I_W & \triangleq & F^*(mv, i_w) \\
P_R & \triangleq & F^*(mv^{-1}, p_r) \\
P_W & \triangleq & F^*(mv, p_w)
\end{array}
$$

Nous pouvons maintenant définir l'ensemble V des identifiants vivants (c'est à dire accédés en lecture par au moins une instruction) et P des identifiants correspondant à des cases de la pile OCaml ne pouvant pas être transformées en variables C. Les définitions de V et P sont :

$$
\begin{array}{lcl}
V & \triangleq & I_R \cup P_R \\
P & \triangleq & (P_R \cap gc \cap P_W) \setminus I
\end{array}
$$

4.3.2. Validité de l'extraction

L'extraction d'une case de la pile OCaml serait « invalide » s'il existait un chemin d'exécution dans lequel on commencerait par écrire l'adresse d'un bloc à l'intérieur de la case, puis une allocation déclencherait le lancement d'un recyclage de la mémoire, puis le contenu du bloc serait accédé en lecture ou en écriture.

De par la définition de l'ensemble P des identifiants maintenus dans la pile OCaml, un tel scénario est clairement impossible.

4.3.3. Faiblesse de l'algorithme

On remarque cependant que la construction ci-dessus de l'ensemble P n'est pas optimal car elle ne tient pas compte de l'ordre d'évaluation. Par exemple, en OCaml, il arrive qu'une variable ait une portée lexicale plus grande que sa durée de vie réelle comme dans l'exemple suivant :

```
let c = (42, 43) in
let n = fst c in
String.create n
```

Comme la portée lexicale de `c` s'étend jusqu'à l'allocation mémoire (l'appel à `String.create`), la valeur de la variable `c` sera dans la pile OCaml au moment d'un lancement éventuel du ramasse-miettes alors qu'elle ne sera jamais dé-référencée après le recyclage de la mémoire. On pourrait l'extraire de la pile OCaml de manière sûre, mais l'algorithme précédent ne le fera pas.

4.4. Gains

En pratique, cet algorithme permet l'extraction d'une proportion assez importante de la pile OCaml. Par exemple, si l'on concatène l'intégralité des programmes utilisés pour l'étude de performance présentée en section 6, la compilation de ce code avec `ocamlcc -stat` indique qu'il contient 2404 appels de fonctions. On connaît statiquement la fonction appelée pour plus de 70% d'entre eux. Sur l'ensemble des `PUSH` présents dans le bytecode, 44% sont supprimés car on connaît statiquement la valeur mise sur la pile. Sur les `PUSH` restant, 44% mettent sur la pile une valeur que l'on sait ne pas être allouée, et 74% sont transformés en affectations de variables C. Au final, seulement 15% des `PUSH` sont réellement compilés en `PUSH`.

Cette optimisation apporte en efficacité sur différents plans. Elle favorise les optimisations du compilateur C car les données manipulées sont dans des variables locales plutôt que dans un tableau global. De plus, l'ensemble de racines manipulé par le GC est moins « pollué » par des constantes ou par des pointeurs qui ne seront en fait plus utilisés. Elle apporte un gain notable en efficacité.

5. Problèmes

Nous listons ici quelques-uns des problèmes que nous avons rencontrés. Il s'agit de problèmes très classiques auxquels ont eu à faire face la majorité des compilateurs produisant du C à partir de langages fonctionnels.

Gestion des exceptions

OCamlCC utilise par défaut les primitives C `setjmp/longjmp` pour encoder le traitement des exceptions OCaml de façon compatible avec la bibliothèque d'exécution OCaml. Cette dernière utilise elle aussi ce mécanisme.

OCamlCC permet aussi de générer du code C++ pouvant être compilé par exemple avec le compilateur `g++`. L'utilisateur peut alors demander l'utilisation la structure de contrôle `try-catch` de C++ à la place de `setjmp/longjmp`.

Comme on peut s'y attendre, le choix du mécanisme d'exceptions influence profondément les performances du programme compilé. Ce point est discuté dans la section 6.

Optimisation des appels terminaux

Il est bien connu que les compilateurs C peuvent implémenter incorrectement les appels terminaux [1], de sorte qu'un appel terminal utilise tout de même de l'espace de pile. Ce non respect des appels terminaux a plusieurs origines. D'une part, en C, le fait qu'un appel soit terminal n'est pas une notion syntaxique, contrairement à ce qui se passe en OCaml, par exemple. D'autre part, la sémantique du langage C autorise à passer plus d'arguments que nécessaire à une fonction ce qui rend la convention d'appel utilisée par le langage difficilement compatible avec le respect des appels terminaux. Ce non-respect des appels terminaux a été problématique pour tous les compilateurs de langages fonctionnels cherchant à produire du C efficace.

Durant la conception et le développement d'OCamlCC, trois techniques ont été implantées pour honorer les appels terminaux :

1. L'insertion de code assembleur.
   Ce code n'est compatible qu'avec `gcc` et n'a été réalisé que pour les architectures `x86` et `x86-64` avec l'unique but de pouvoir tester rapidement OCamlCC sur de vrais programmes. En dehors de sa très faible portabilité, le principal inconvénient de cette technique est qu'elle oblige à désactiver l'« inlining » des fonctions utilisant le code assembleur en question, ce qui est pénalisant pour les performances.
2. La génération de C portable dans lequel, d'une part, les arguments (à l'exception du premier et de l'environnement de la fonction appelée) sont systématiquement passés dans des variables globales, et d'autre part, les appels à la fonction `setjmp` sont délégués à des fonctions auxiliaires. Lorsque la fonction auxiliaire souhaite effectuer un appel terminal, elle stocke les arguments dans des variables globales et c'est son appelant qui effectue l'appel terminal à sa place. Il s'agit d'une variante d'une technique standard nommée « trampoline ». Le principal inconvénient de cette technique est une perte de performance. En effet, l'utilisation systématique de variables globales pour les arguments autres que le premier et les appels indirects (c'est à dire passant par des pointeurs de fonction C) perturbent les analyses de flot du compilateur C.
3. La génération de C portable basée également sur une sorte de trampoline, mais où les fonctions reçoivent leurs arguments de façon habituelle.
   Le principe est simple : déléguer chaque appel terminal que le compilateur C ne sait pas gérer à l'appel non terminal actif le plus récent. Plus précisément, un appel terminal non géré par le compilateur C est compilé en :

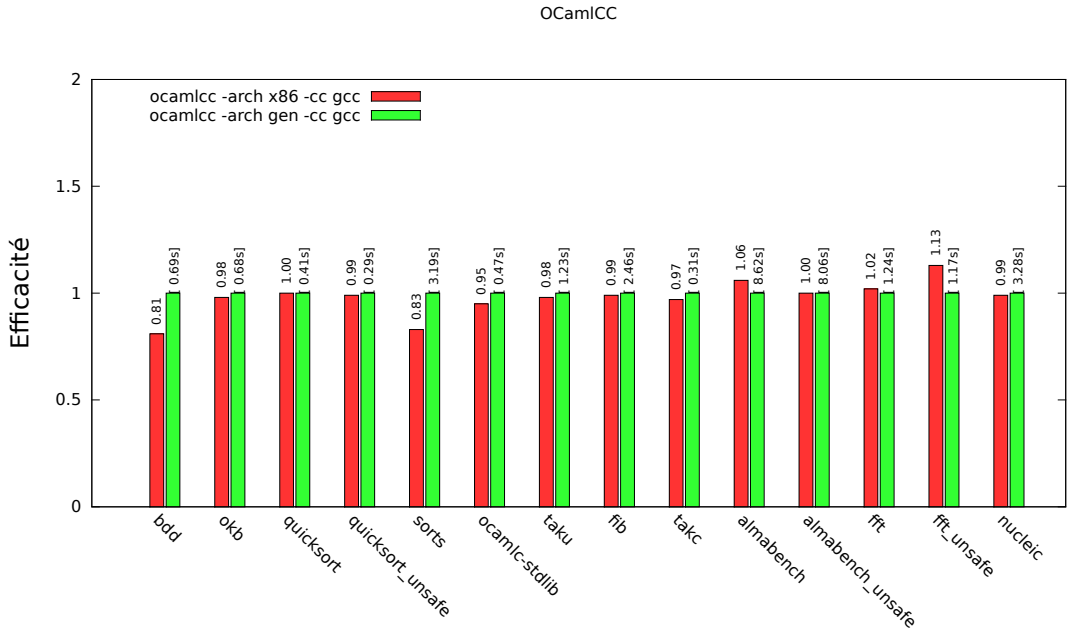

FIGURE 1 – Efficacité comparée des codes produits avec OCamlCC utilisant de l'assembleur (`ocamlcc -arch x86`) et la production de C portable (`ocamlcc -arch gen`, normalisée à 1). Au sommet de la colonne normalisée apparaît le temps d'exécution du programme, et sur les autres colonnes le rapport d'efficacité. Plus l'ordonnée est élevée, plus le code est efficace.

- le calcul des arguments et leur stockage dans des variables globales ;
- le calcul de la fermeture ou de l'adresse du code (si la fonction appelée est connue statiquement et si la fermeture n'est pas nécessaire) et son stockage dans une variable globale ;
- le retour d'une valeur unique, qui ne peut jamais être renvoyée par une fonction OCaml standard.

D'autre part, après tous les appels non terminaux à une fonction que l'on ne connaît pas ou à une fonction effectuant un appel terminal non géré par le compilateur C, on ajoute une boucle effectuant les appels terminaux qui n'ont pas pu être effectués par l'appelé. Les paramètres et les fonctions à appeler de la sorte sont pris dans les variables globales où elles ont été stockées. La boucle continue tant que la valeur renvoyée par l'appelé est cette valeur unique mentionnée ci-dessus.

Cette technique, comme la version assembleur, présente l'avantage de passer normalement les arguments aux fonctions C, c'est à dire dans la pile C ou dans les registres. Toutes les fonctions sont aussi « *inlinable* » qu'habituellement, et aucune hypothèse n'est faite sur le compilateur, ni sur l'architecture cible.

Pour chacune de ces techniques, lorsque l'on sait que le compilateur C saura gérer correctement un appel terminal, on lui laisse ce soin car le code qu'il produit est généralement plus performant. En pratique, sur les tests que nous avons effectués, les performances obtenues avec la troisième technique sont presque toujours meilleures (jusqu'à près de 20%) qu'avec les deux autres et au pire 1,13 fois moins bonnes, comme le montre la figure 1. C'est donc l'algorithme utilisé par défaut par OCamlCC.

Rattrapage des signaux

Lors du traitement d'un signal, le contrôle peut être transféré à du code nécessitant un recyclage de la mémoire par le ramasse-miettes. Les signaux ne peuvent alors être testés et traités que lorsque la mémoire est dans un état cohérent. La réception d'un signal lève un drapeau global qui doit donc être testé périodiquement pour que le signal soit pris en compte. La fréquence des tests de ce drapeau a un impact sur l'efficacité, puisque chacun de ces tests peut rompre le flot de contrôle. Se pose donc la question de savoir quand procéder à ces tests : à chaque allocation, avant chaque appel de fonction, dans le corps des boucles et à la sortie d'un « `try _ with _` » (`POPTRAP`) comme le fait le générateur de bytecode `ocamlc`, ou bien seulement à chaque allocation comme le fait le générateur de code natif `ocamlopt`. OCamlCC adopte par défaut le comportement d'`ocamlopt`, et, au travers d'une option, donne à l'utilisateur le choix entre réactivité et efficacité.

Taille du code C, ressources nécessaires à la compilation

Pour permettre l'*inlining* des fonctions de bibliothèques par le compilateur C, nous incluons textuellement (par `#include`) la bibliothèque d'exécution OCaml dans le fichier C produit.

OCamlCC produit ainsi un seul fichier C, qui peut être assez volumineux. Jusqu'à présent, sa compilation par des compilateurs tels que GCC n'a jamais été problématique. Par exemple, la compilation par `gcc -O3` du fichier C produit à partir de la version bytecode du compilateur `ocamlc` prend moins de 2 minutes et utilise moins d'1 Go de RAM. Avec `gcc -O0`, le temps de compilation descend à 34 secondes et utilise 610 Mo de RAM.

Si une telle compilation globale devenait un problème, il serait facile d'envisager une organisation du fichier C lui permettant d'être compilé par fragments, au prix éventuel de la perte d'une partie des opérations d'*inlining* du compilateur C, dans le cas où celui-ci n'effectue pas d'optimisation inter-modules lors de l'édition de liens.

6. Performances

La figure 2 montre les efficacités relatives des codes engendrés par `ocamlc`, le compilateur de bytecode OCaml, le compilateur JIT pour OCaml de B. Meurer [6], le code produit par OCamlCC compilé par `gcc -O3`, et enfin le code produit par `ocamlopt`, le générateur de code natif d'OCaml. Les colonnes représentent le rapport t_{norm}/t, où t_{norm} est le temps obtenu par le code engendré par `ocamlopt` et t le temps obtenu par le compilateur en question. Plus le temps d'exécution est faible, plus la colonne est haute. Le temps d'exécution en secondes est affiché sur la colonne normalisée à 1 (celle d'`ocamlopt`), et les autres colonnes indiquent le rapport de performance du code correspondant.

Les programmes utilisés proviennent de la suite de tests de la distribution OCaml. Les programmes les plus à gauche (de `bdd` à `takc`) effectuent du calcul symbolique ou bien des calculs sur des entiers. Les plus à droite (de `almabench` à `nucleic`) effectuent plutôt du calcul numérique flottant. On note que `ocamlcc` est toujours plus rapide que `ocamlc` d'un facteur allant de 1,7 à 6,6, voire même de 13 à plus de 100 sur `fib` et `takc`. Il faut dire que sur ces deux exemples, l'extraction des valeurs (entières, dans ce cas) de la pile, et la détection des appels statiques produisent un code C compilable efficacement. Pour cette même raison, sur ces deux exemples, `ocamlcc` est aussi plus efficace qu'`ocamlopt`. L'exemple `taku` implémente la même fonction que `takc`, mais utilise des fonctions recevant des triplets au lieu d'être curryfiées comme dans `takc`, et on voit qu'`ocamlopt`, en évitant l'allocation de ces triplets, est 2,7 fois plus rapide qu'OCamlCC sur cet exemple.

Sur les autres programmes, `ocamlopt` est entre 1,3 et plus de 4 fois plus efficace qu'`ocamlcc`. On notera tout de même la bonne performance du code produit par OCamlCC pour `ocamlc` lui-même, et utilisé pour compiler la totalité de la bibliothèque standard d'OCaml (`ocamlc-stdlib`). Ce code n'est

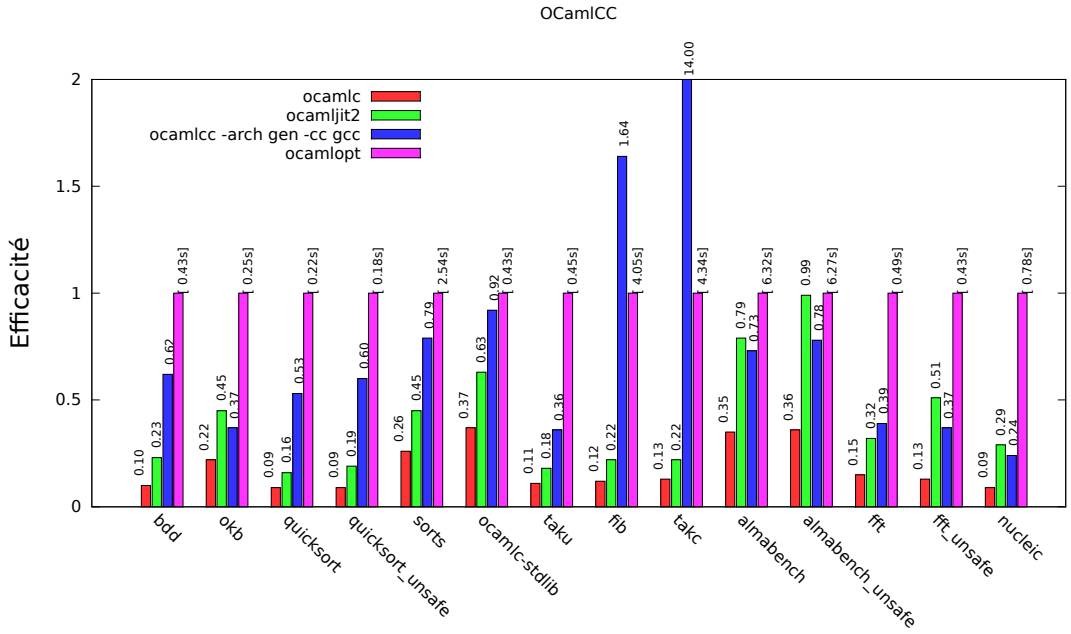

FIGURE 2 – Efficacité des codes produits par différents compilateurs OCaml, normalisée par rapport au code natif d'`ocamlopt`. Plus l'ordonnée est élevée, plus le code est efficace.

que moins de 10% plus lent que `ocamlc.opt`, résultant de la compilation d'`ocamlc` par `ocamlopt`.

Le traitement des calculs flottants effectué par OCamlJIT2 lui permet d'être jusqu'à 27% plus rapide qu'`ocamlcc` sur certains programmes de calcul numérique. OCamlJIT2 bat aussi OCamlCC d'une courte tête sur un programme comme `okb` qui utilise massivement des exceptions comme structures de contrôle. Cette différence d'efficacité est probablement due au coût du couple `setjmp/longjmp`, supérieur à celui du chaînage effectué par l'interprète de bytecode. Sur les autres programmes, OCamlJIT2 est jusqu'à 3 fois plus lent qu'OCamlCC.

Le graphique 3, quant à lui, permet de comparer l'action des différents compilateurs C (GNU `gcc`, `g++` et `clang`) sur le code produit par OCamlCC.

Même s'il est difficile de classer de façon absolue ces compilateurs C, on note que `gcc` est le plus régulier. Le « `try _ catch _` » de C++ démontre son inefficacité [2] sur le traitement des exceptions pour les programmes en faisant un usage intensif comme `okb`. Le compilateur `clang` est généralement très proche de `gcc`. Il est cependant largement distancé sur les *micro-benchmarks* que sont `fib` et `takc` où `gcc` reconnaît des schémas pour lesquels il sait produire du très bon code.

7. Développement et distribution

OCamlCC est d'ores et déjà utilisable sur des programmes réels et est doté d'une brève documentation. Si les idées qui le sous-tendent sont communes aux deux auteurs de cet article, son

2. En C++, où les exceptions sont censées être réservées à des situations exceptionnelles, la pose d'un rattrapeur ne coûte rien ; en revanche, la levée d'une exception est coûteuse à cause de l'examen de la pile qui en résulte.

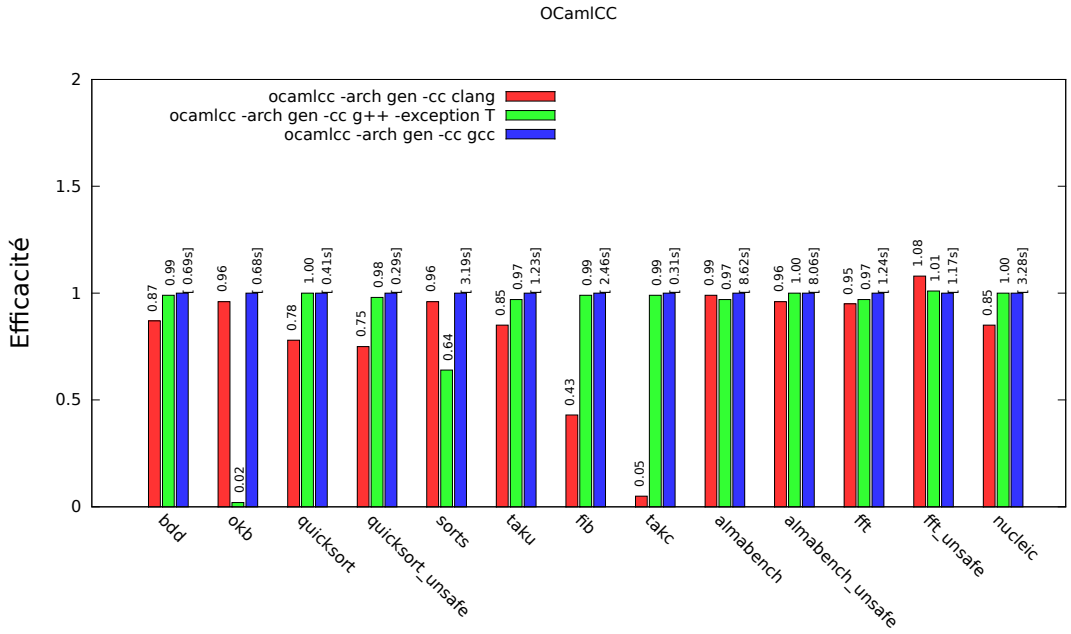

FIGURE 3 – Efficacité des différents compilateurs C sur le code produit par OCamlCC.

développement a été essentiellement réalisé par le second auteur et est publiquement disponible depuis la page :

`https://github.com/ocaml-bytes/ocamlcc`

8. Travaux futurs

Il reste de nombreuses possibilités d'amélioration d'OCamlCC. Nous optimisons actuellement les appels de fonction, mais n'avons prêté que relativement peu d'attention à l'exécution des corps de fonction, qui restent essentiellement des séquences de macros C correspondant aux instructions du bytecode. Certaines sous-séquences (opérations sur des nombres flottants, par exemple) « emballent » et « déballent » [3] inutilement des résultats de calculs intermédiaires. OCamlJIT2 élimine certaines de ces opérations et en tire des améliorations notables de performances sur les calculs flottants [6].

De façon plus générale, il conviendrait de mieux profiter du fait qu'OCamlCC traite un programme complet pour effectuer des optimisations plus agressives visant notamment à réduire plus encore les opérations d'emballage et de déballage des valeurs.

Afin de permettre la compilation d'applications volumineuses sur des petites machines, il sera probablement nécessaire de diviser le code produit en entités de taille raisonnable pouvant être compilées séparément par les compilateurs C. Ce n'est qu'une question de division du code et ne pose pas de problème particulier.

À terme, il serait intéressant qu'OCamlCC puisse produire des applications capables d'effectuer du chargement dynamique de code. Cela pourrait consister en la production de bibliothèques dynamiques (`.so`) à partir de fichiers de bytecode et utiliser le chargement dynamique du système d'exploitation.

3. Par *boxing/unboxing*.

Les compilateurs C comme GCC ont des capacités de compilation croisée qu'OCamlCC devrait être en mesure d'apporter à OCaml. Un travail de conception, de conditionnement et de documentation est nécessaire pour fournir les bonnes options et indiquer comment accéder aux bibliothèques des architectures qui pourraient être ciblées par ces capacités de compilation croisée.

À plus long terme, des langages intermédiaires autres que C peuvent être envisagés (LLVM, par exemple). Pour un langage donné, il faut d'une part traduire le bytecode OCaml vers ce langage comme nous le faisons vers C, et d'autre part compiler ou réécrire la bibliothèque d'exécution d'OCaml en ce même langage ou en un langage qui peut lui être lié.

9. Conclusion

Avec OCamlCC, nous avons montré que l'on pouvait compiler OCaml en du code natif efficace au moyen d'une traduction du bytecode en C. L'ambition d'OCamlCC est de fournir une compilation alternative au compilateur `ocamlc` sur des architectures pour lesquelles `ocamlopt` n'est pas disponible.

Le bytecode d'OCaml est très stable, et sa bibliothèque d'exécution ne varie qu'assez peu au gré des versions d'OCaml : la maintenance d'OCamlCC est donc relativement aisée puisqu'aucune synchronisation avec le système OCaml n'est nécessaire à son bon fonctionnement.

L'idée de traduire le bytecode vers un langage qui peut être lié à la bibliothèque d'exécution d'OCaml est aussi intéressante en soi et mérite d'être explorée plus avant afin de cibler d'autres langages intermédiaires.

Remerciements Merci aux rapporteurs dont les questions et remarques nous ont incités à améliorer et éclaircir quelques points de cet article.

Références

[1] Andreas Bauer. Compilation of functional programming languages using GCC—Tail calls. Master's thesis, Institut für Informatik, Technische Universität München, 2003.

[2] Colin Benner. An LLVM Backend for OCaml. To be presented at the OCaml Meeting 2012.

[3] Emmanuel Chailloux. *Compilation des langages fonctionnels : CeML un traducteur ML vers C*. PhD thesis, Université Paris 7, 1991.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 3.12, Documentation and user's manual, 2011.

[6] Benedikt Meurer. OCamlJIT 2.0 - Faster Objective Caml. *CoRR*, abs/1011.1783, 2010.

[7] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The Glasgow Haskell compiler : a technical overview, 1992.

[8] Manuel Serrano. Bigloo User's Manual. Technical Report RT-0169, INRIA, December 1994. Projet ICSLA.

[9] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required : compiling Standard ML to C. *ACM Lett. Program. Lang. Syst.*, 1(2) :161–177, June 1992.

[10] Benoît Vaugon. OCamlClean. Presented at the OCaml Meeting 2011.

[11] Jérôme Vouillon and Vincent Balat. From bytecode to Javascript : the Js_of_ocaml compiler. Presented at the OCaml Meeting 2011.