INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Le système Caml Special Light: modules et compilation efficace en Caml

Xavier Leroy

N ° 2721
Novembre 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ISSN 0249-6399

Le système Caml Special Light: modules et compilation efficace en Caml

Xavier Leroy

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Cristal

Rapport de recherche n°2721 — Novembre 1995 — 21 pages

Résumé : Ce rapport présente une vue d'ensemble du système Caml Special Light, une implémentation expérimentale du langage Caml offrant deux extensions majeures: premièrement, un calcul de modules (incluant les foncteurs et les vues multiples d'un même module) dans le style de celui de Standard ML, mais s'appuyant sur les avancées récentes dans la théorie du typage des modules et préservant la compatibilité avec la compilation séparée; deuxièmement, un double compilateur, produisant à la fois du du code natif efficace, pour les applications de Caml gourmandes en temps de calcul, et du code abstrait interprété, pour la rapidité de compilation et le confort de mise au point.

Mots-clé : Caml, ML, langages fonctionnels, systèmes de modules, foncteurs, compilation séparée, production de code, bytecode, fermetures.

(Abstract: pto)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

The Caml Special Light system: modules and efficient compilation for Caml

Abstract: This report gives an overview of the Caml Special Light system, an experimental implementation of the Caml language with two major extensions: first, a module calculus (including functors and multiple views of a module) in the style of Standard ML, but relying on recent advances in the type theory of modules, and supporting separate compilation; second, a compiler producing both efficient native code, for computation-intensive Caml programs, and interpreted bytecode, for fast turnaround and ease of development.

Key-words: Caml, ML, functional languages, module systems, functors, separate compilation, code generation, bytecode, closures.

1 Introduction

Cet article présente une vue d'ensemble du système Caml Special Light [20], une implémentation expérimentale du langage Caml [34]. Caml Special Light est une réimplémentation complète du système Caml Light [21] qui offre deux extensions majeures: premièrement, un calcul de modules (incluant les foncteurs et les vues multiples d'un même module) dans le style de celui de Standard ML [23, 26], mais s'appuyant sur les avancées récentes dans la théorie du typage des modules [17, 12] et préservant la compatibilité avec la compilation séparée; deuxièmement, un double compilateur, produisant à la fois du du code natif efficace, pour les applications de Caml gourmandes en temps de calcul, et du code abstrait interprété, pour la rapidité de compilation et le confort de mise au point.

Le besoin de ces deux extensions se faisait sentir depuis plusieurs années. Le système de modules très simple de Caml Light [34, chapitre 10] est fort pratique, en raison notamment de sa parfaite compatibilité avec la compilation séparée, mais il est incapable d'exprimer certaines formes de paramétrisation utiles aussi bien à petite échelle [1] qu'à plus grande échelle [22]. Le compilateur de bytecode de Caml Light est universellement apprécié pour sa rapidité de compilation, mais la vitesse d'exécution des programmes produits n'est pas suffisante pour certaines applications. Enfin, le code du système Caml Light commençait à dater un peu, et une réécriture complète me semblait nécessaire pour permettre le développement de futures extensions, en particulier dans le domaine de la programmation par objets.

Les deux parties de cet article présentent l'une, le calcul de modules, et l'autre, la technologie de compilation de Caml Special Light. La présentation est volontairement informelle; le lecteur se reportera aux références pour plus de détails.

2 Les modules

2.1 Présentation du langage de modules

Un programme Caml classique se compose d'une suite de définitions de valeurs, de types et d'exceptions. (Les expressions e évaluées pour leurs effets peuvent être vues comme des définitions triviales de valeurs `let _` $= e$.) Un système de modules fournit des constructions pour définir, nommer et utiliser des ensembles de définitions. L'unité de base du langage de modules de Caml Special Light est la *structure*, c'est-à-dire un ensemble de définitions entre `struct`...`end`:

définitions de valeurs et de fonctions	`let` $x = \ldots$
définitions de types	`type` $t = \ldots$
définitions d'exceptions	`exception` E
définitions de (sous-)modules	`module` $X = \ldots$
définitions de types de modules	`module type` $T = \ldots$

Voici par exemple une structure fournissant le type entier et une fonction d'ordre sur ce type:

```
module IntOrder =
  struct
    type t = int
    let compare x y =
      if x = y then Equal else if x < y then Less else Greater
  end
```

Les composantes d'une structure peuvent être employées dans tout contexte de la même sorte (une composante de valeur dans un contexte de valeur, une composante de type dans un contexte de type, etc.) grâce à la "notation pointée" (*dot notation*) bien connue: `IntOrder.t` désigne le type `t` défini dans la structure `IntOrder`; `IntOrder.compare`, la fonction `compare` de cette structure. Des chemins d'accès plus long permettent l'accès dans les modules imbriqués: `M.N.t` désigne le type `t` de la sous-structure `N` de la structure `M`. Une construction `open` permet d'économiser sur la notation pointée: dans la portée de `open IntOrder`, l'identificateur `compare` est lu comme `IntOrder.compare`, et de même pour `t`.

Les *signatures* typent les structures de la même manière que les types (du langage de base) typent les valeurs. Une signature `sig`...`end` est une collection de déclarations qui spécifie les noms des composantes définies par une structure ainsi que les propriétés de typage qu'elles doivent vérifier:

spécifications de valeurs (σ est le schéma de types que doit posséder la valeur x)	`val` $x : \sigma$
spécifications de types, abstraites (spécifie seulement l'existence et l'arité du type t mais laisse son implémentation non spécifiée)	`type` t
spécifications de types, manifestes (spécifie exactement l'implémentation du type t)	`type` $t = \tau$
spécifications d'exceptions	`exception` E
spécifications de (sous-)modules (M est le type de module que doit posséder le module X)	`module` $X : M$
spécifications de types de modules, abstraites	`module type` T
spécifications de types de modules, manifestes	`module type` $T = M$

Par exemple, une signature possible pour la structure `IntOrder` ci-dessus est

```
sig
  type t = int
  val compare: t -> t -> order
end
```

Les signatures permettent de cacher certaines composantes d'une structure ou de les rendre abstraites par l'intermédiaire d'une contrainte de signature. Par exemple,

```
module M1 = (IntOrder: sig val compare: int -> int -> order end)
```

cache la composante `t` de `IntOrder`: `M1` n'a qu'une composante accessible, `compare`. De même,

```
module M2 = (IntOrder: sig type t val compare: t -> t -> order end)
```

oublie l'égalité entre le type `IntOrder.t` et le type `int`, rendant `M2.t` abstrait.

Les modules paramétrés se présentent sous la forme de *foncteurs*, c'est-à-dire de fonctions des modules dans les modules. Le langage de modules ressemble donc à un lambda-calcul simplement typé dont les objets de base sont les structures. Le langage ne sépare pas foncteurs et structures: les deux sont des modules au même titre et se nomment par la liaison `module`. Par exemple, voici un module paramétré par un type et une fonction d'ordre total sur ce type, qui implémente un type abstrait des ensembles sous forme d'arbres binaires croissants:

```
module Set =
  functor (Elements: sig type t val compare: t -> t -> order end) ->
    struct
      type element = Elements.t
      type set = Empty | Node of set * element * set
      let empty = Empty
      let rec add elt set =
        match set with
          Empty -> Node(Empty, elt, Empty)
        | Node(l, e, r) ->
            match Elements.compare elt e with
              Less -> Node(add elt l, e, r)
            | Equal -> set
            | Greater -> Node(l, e, add elt r)
      let rec member elt set = ...
    end
```

Un type de module acceptable pour ce foncteur est:

```
Set:
  functor (Elements: sig type t val compare: t -> t -> order end) ->
    sig
      type element = Elements.t
      type set
      val empty: set
      val add: element -> set -> set
      val member: element -> set -> bool
      ...
    end
```

Ce type de module rend abstraite la composante `set` du résultat du foncteur, cachant ainsi la représentation des ensembles à l'utilisateur et assurant que l'invariant "les arbres sont croissants de gauche à droite pour l'ordre `Elements.compare`" est toujours vérifié.

L'application de foncteurs se note comme l'application de fonctions et renvoie de nouveaux modules:

```
module IntSet = Set(IntOrder)
... IntSet.add 3 IntSet.empty: IntSet.set ...
```

2.2 Les types manifestes

L'innovation majeure de ce système de modules vis-à-vis des modules de Standard ML est l'introduction de spécifications de types manifestes (`type` $t = \tau$) dans les signatures. En Standard ML, une signature spécifie uniquement le nom et l'arité d'une composante de type, mais ne peut ni révéler son implémentation, ni la cacher systématiquement: un mécanisme séparé et non directement contrôlé par l'utilisateur assure la propagation des égalités entre types.

Les types manifestes fournissent un mécanisme simple et bien compris dans le cadre de la théorie des types [17, 12] pour assurer plusieurs tâches importantes du système de modules:

Donner des signatures complètes aux structures: des signatures qui contiennent exactement toutes les informations de types nécessaires au typage des clients de la structure. Par exemple, l'hypothèse

```
IntOrder:  sig type t = int  val compare: t -> t -> order end
```

suffit à établir que l'application `IntOrder.compare 1 2` est bien typée, puisque l'égalité `IntOrder.t` = `int` se déduit de la signature. De même, l'hypothèse

```
IntSet: sig type set  ... end
```

garantit que `IntSet.set` est un type abstrait, puisqu'aucune égalité entre lui et un autre type n'est connue. Dans cette approche, il n'est pas nécessaire de tenir à jour des approximations de structures plus riches que les signatures, comme les "signatures sémantiques" de Standard ML. Non seulement le typage des modules s'en trouve simplifié et rapproché d'un système de types classique, mais de plus la compilation séparée à la manière de Modula-2 est facilitée [17].

Exprimer la propagation des égalités de types à travers les foncteurs. La combinaison des types manifestes et des types fonctoriels dépendants (le nom du paramètre peut apparaître dans le type du résultat) permet d'exprimer très précisément les relations entre les composantes de types du résultat d'un foncteur et celles de son argument. Par exemple, le type du foncteur `Set` donné page 5 traduit le fait que le type `element` du résultat est identique au type `t` de l'argument, alors que le type `set` est toujours abstrait. Au moment de

l'application `Set(IntOrder)`, la substitution du paramètre formel `Elements` par l'argument effectif `IntOrder` dans la signature du résultat donne:

```
IntSet:
  sig
    type element = IntOrder.t
    type set
    ...
  end
```

De cette signature et de celle de `IntOrder`, le typeur déduit immédiatement `IntSet.element` = `int`, comme désiré.

Ce mécanisme de propagation s'applique aussi pour les foncteurs d'ordre supérieur (les foncteurs prenant d'autres foncteurs en argument), modulo une extension de la notation pointée de la forme $F(A).t$, autorisant une référence à la composante de type t du résultat de l'application du foncteur F au module A. On se reportera à [18] pour plus de détails.

Exprimer des contraintes de partage entre les arguments d'un foncteur. Standard ML fournit une construction spéciale, `sharing type`, pour exiger l'égalité entre deux composantes de types des arguments d'un foncteur. En Caml Special Light, la contrainte s'exprime par un type manifeste en position d'argument de foncteur. Par exemple, le foncteur Standard ML

```
functor F(structure A: sig type t ... end
          structure B: sig type u ... end
          sharing type A.t = B.u) = ...
```

se traduit en Caml Special Light par

```
module F =
  functor (A: sig type t ... end) ->
  functor (B: sig type u = A.t ... end) -> ...
```

ou, avec une notation plus compacte,

```
module F(A: sig type t ... end)(B: sig type u = A.t ... end) = ...
```

Le type manifeste dans la signature de `B` a les mêmes effets que la contrainte de partage en Standard ML: supposer `A.t` = `B.u` dans le corps du foncteur, et exiger à chaque application `F`$(C)(D)$ que C.`t` = D.`u`.

On montre formellement [19] que toute contrainte de partage entre types peut s'exprimer ainsi en termes de types manifestes. Standard ML fournit également des contraintes de partage entre structures (exigeant l'égalité complète entre sous-structures de deux structures, par exemple), qui ne sont pas exprimables en termes de types manifestes, car elles expriment non seulement des égalités entre les composantes de types des structures, mais aussi entre

les valeurs. Le système de modules de Caml Special Light fait une croix sur les spécifications d'égalité entre valeurs, qui sont difficiles à formaliser (une grande partie de la complexité de [26] est due au partage entre structures) et d'un intérêt pratique faible.

L'expression du partage par des types manifestes pose un problème pratique de duplication de signatures. Supposons les identificateurs `S1` et `S2` liés à deux grosses expressions de signatures:

```
module type S1 = sig type t ... end
module type S2 = sig type u ... end
```

Un foncteur prenant en paramètre `A` : `S1` et `B` : `S2` partageant `A.t` et `B.u` nécessite de dupliquer la définition de `S1` ou `S2` pour y introduire une égalité de types:

```
module F(A: S1)(B: sig type u = A.t ... end) = ...
```

C'est en pratique très désagréable. Pour corriger ce problème, Caml Special Light introduit une construction `with` sur les types de modules: `S2 with type u = A.t` est la signature identique à `S2`, à l'exception du champ `type u` qui est remplacé par `type u = A.t`. Avec cette notation, le foncteur `F` s'écrit de manière très compacte:

```
module F(A: S1)(B: S2 with type u = A.t) = ...
```

L'avantage de cette notation `with` est qu'elle s'élimine facilement avant le typage proprement dit, et ne complique donc pas les règles de typage du calcul de modules.

Pour traiter les modules emboîtés, la construction `with` a été étendu à `with type` *chemin de type* = *expression de type*, par exemple `with S.t = int`. On dispose aussi de `with module` *chemin de module* = *chemin de module* pour identifier en bloc tous les types contenus dans deux sous-structures.

2.3 Modules et compilation séparée

Le calcul de modules de Caml Special Light s'intègre très naturellement dans le mécanisme de compilation séparée à la manière de Modula-2 qu'on trouvait déjà dans Caml Light. Dans ce système, une unité de compilation u se compose de deux fichiers: un fichier d'interface u`.mli` contenant des spécifications pour les composantes publiques de l'unité, et un fichier d'implémentation u`.ml` contenant des définitions pour ces composantes. D'autres unités de compilation peuvent faire référence aux composantes exportées par u via la notation pointée $u.t$ (notée u_t en syntaxe Caml Light) ou `open` u.

En termes du langage de module, ce mécanisme de compilation séparée se traduit comme suit: chaque unité u est une structure; sa signature est contenue dans le fichier u`.mli`; son implémentation, dans le fichier u`.ml`. Tout se passe comme si l'utilisateur avait défini

```
module u = (struct contenu de u.ml end : sig contenu de u.mli end)
```

De plus, le typage de *u*.`ml` et *u*.`mli` s'effectue dans l'environnement initial $u_1 : \Sigma_1; \ldots; u_n : \Sigma_n$, où $u_1 \ldots u_n$ sont les noms des unités de compilations disponibles (dans le répertoire courant, la bibliothèque standard et les répertoires ajoutés par l'utilisateur) et $\Sigma_1 \ldots \Sigma_n$ leurs signatures (données par les fichiers `.mli` correspondants). Le système de fichier joue ici le rôle d'environnement de typage persistant entre les compilations.

La force de ce mécanisme de compilation séparée est que chaque fragment de programme peut être lu, compris et vérifié du point de vue des types en ayant uniquement accès aux interfaces des unités qu'il utilise, sans jamais avoir besoin de leurs implémentations. Ceci autorise en particulier l'écriture des fragments d'un programme dans un ordre arbitraire et en détectant au plus tôt les incohérences entre fragments, contrairement à d'autres systèmes de compilation séparée qui imposent un développement ascendant (*bottom-up*), une programmation complètement fonctorialisée [29], ou retardent les vérifications de cohérence inter-fragments [32, 11].

2.4 Le typeur des modules

La première implémentation du typeur des modules, réalisée par François Pottier [28], utilisait des marques (*stamps*) uniques pour représenter les types générés, ou en d'autres termes les classes d'équivalences de types modulo les équations induites par les types manifestes. Cette technique, employée dans toutes les implémentations des modules de Standard ML [24, 9, 25], semblait a priori plus efficace que l'implémentation directe des règles de typage du calcul de modules; des travaux antérieurs sur l'équivalence entre ce calcul de modules et celui de Standard ML en assuraient la correction et la complétude [19].

Cette implémentation à base de marques s'est révélée beaucoup plus difficile que prévu. D'une part, les manipulations de types de foncteurs, nécessitant des unifications destructrices mais temporaires, sont difficiles et peu commodes. D'autre part, la compilation séparée nécessite des marques persistantes, conservant leur identité unique à travers plusieurs exécutions du compilateur; ces marques persistantes sont difficiles à réaliser dans le système Unix de manière portable [3].

L'implémentation actuelle du typeur de modules est beaucoup plus naïve et suit à la lettre les règles de typage données dans [17, 18]. Les problèmes d'efficacité du typage évoqués plus haut, liés en particulier au fait que chaque accès à une composante d'un module nécessite une substitution pour préserver les dépendances sur les composantes précédentes, sont évités par une implémentation soignée du type abstrait des environnements de typage, qui précalcule ces substitutions lors de l'ajout d'un module à un environnement, et utilise des arbres AVL pour garantir l'accès en temps logarithmique en la taille de l'environnement.

Les problèmes de persistance liés à la compilation séparée disparaissent entièrement: les noms d'unités de compilation étant uniques, et les types qu'elles définissent étant représentés par des chemins d'accès à partir de ces noms, l'unicité de ces types est assurée automatiquement.

Finalement, il est à noter que l'implémentation des environnements de typage est entièrement applicative, sans aucune modification physique. Quoiqu'un peu lourd en apparence

(presque toutes les fonctions du typeur ont l'environnement comme paramètre supplémentaire), ce style évite bien des erreurs dans l'implémentation des règles de typage et rend plus robuste la reprise sur erreur dans le système interactif.

2.5 La compilation des modules

Le langage de modules ne pose pas de problèmes majeurs de compilation [28]. Les structures sont représentées par des n-uplets contenant les composantes de valeurs, de modules et d'exceptions de la structure; les composantes de types et de types de modules disparaissent bien sûr à l'exécution. La signature d'une structure détermine les positions de ses composantes dans le n-uplet. Lorsqu'une structure est restreinte par une signature, un nouveau n-uplet est construit, contenant uniquement les composantes visibles. Les foncteurs sont compilés exactement comme des fonctions ordinaires.

La compilation du langage de base n'est pas affectée par l'introduction des modules, sauf pour ce qui est de la représentation des exceptions. En Caml Light [15], les exceptions sont représentées par des numéros uniques attribués au moment de l'édition de liens. En présence de foncteurs, une génération plus dynamique des exceptions devient nécessaire: si un foncteur renvoie une exception dans sa structure résultat, chaque application du foncteur doit créer une nouvelle exception. Pour ce faire, on identifie les exceptions par des adresses de blocs dans le tas. Évaluer une définition d'exception `exception` E revient à allouer un nouveau bloc; son adresse devient la "valeur" de l'exception E. Le bloc contient un pointeur vers le nom de l'exception, permettant ainsi l'affichage facile de l'exception.

3 Compilation

3.1 Deux compilateurs en un

La principale originalité du compilateur de Caml Special Light est de produire deux types de codes différents:

- du code (appelé *bytecode*) pour une machine abstraite, qui est ensuite interprété par un programme écrit en C;
- du code assembleur natif pour un certain nombre d'architectures (actuellement: Dec Alpha, Sparc, Mips, et Intel Pentium).

L'interprétation de bytecode est une technique éprouvée qui présente de nombreux avantages: portabilité élevée, rapidité de compilation, compacité des programmes produits, facilité de réalisation d'un toplevel interactif ou d'un débogueur, possibilité de transmettre du code à travers le réseau et de le charger dynamiquement. L'expérience acquise avec Caml Light montre que ces avantages sont très importants en pratique, en particulier pour le développement rapide de prototypes, où l'on passe plus de temps à compiler le programme qu'à l'exécuter.

Le principal inconvénient du bytecode est sa relative lenteur d'exécution: de 2 à 20 fois plus lent que le code natif, suivant le type de programme. Avec la montée en puissance des processeurs modernes, cette lenteur est tout à fait tolérable pour de nombreux types de programmes, en particulier les programmes interactifs. Cependant, il reste certains types d'applications pour lesquels la rapidité du code natif est nécessaire: calculs entiers et flottants; manipulations intensives de tableaux et de chaînes; explorations exhaustives (*brute-force searching*); et même certains types de calculs symboliques. Par exemple, l'environnement de preuve Coq [13], compilé par Caml Light, est suffisamment rapide en utilisation interactive mais un peu lent lorsqu'il compile en "batch" de grosses théories.

De plus, un certain snobisme fait que la compilation de bytecode est souvent méprisée a priori par ceux qui ne l'ont pas pratiquée. Un compilateur natif efficace semble indispensable pour les convaincre du sérieux du langage Caml.

Un compilateur natif est donc nécessaire; mais il n'est pas envisageable de tout faire avec. Les temps de compilation sont élevés, le chargement dynamique de code et l'utilisation interactive sont difficiles, l'interfaçage avec un débogueur aussi, et la transmission de code à travers le réseau très limitée.

L'approche suivie dans Caml Special Light est donc de fournir à la fois un compilateur de bytecode et un compilateur natif en assurant la compatibilité totale entre les deux. Ainsi, les programmes peuvent être développés et mis au point confortablement et rapidement avec le compilateur de bytecode, puis passés à travers le compilateur natif si leurs performances l'exigent.

L'idée n'est pas nouvelle: de nombreux systèmes Lisp, par exemple, fournissent un interprète en plus d'un compilateur natif; aussi, plusieurs compilateurs Caml vers C [10, 30, 31] ont été développés en complément de Caml Light. Dans les deux cas, le problème est de garantir que les deux compilateurs implémentent la même sémantique. En Caml Special Light, ceci est facilité par le fait que les deux compilateurs partagent à la fois les couches hautes (typage et précompilation – voir figure 1) et la bibliothèque d'exécution (*runtime system*): gestion mémoire, représentations de données, fonctions primitives. En particulier, toutes les passes susceptibles de produire des erreurs à la compilation sont partagées. Les systèmes Lisp partagent seulement l'environnement d'exécution; les compilateurs Caml vers C, seulement les couches hautes. De plus, certains traits de la sémantique de Caml (portée hyperstatique, pas de typage dynamique) rendent peut-être ce partage plus naturel qu'en Lisp.

Le résultat est une compatibilité presque totale entre les deux compilateurs de Caml Special Light. La seule différence est que certaines conditions d'erreur (accès en dehors d'un tableau, division par zéro) sont transformées en exceptions par le compilateur de bytecode, mais terminent le programme avec le compilateur natif. La récupération de ces erreurs sous forme d'exceptions est importante en utilisation interactive, mais de mauvais style pour un programme indépendant; la différence de sémantique entre les deux compilateurs ne devrait donc pas gêner en pratique.

La compatibilité entre les deux compilateurs ne va pas jusqu'à permettre le mélange dans un même programme d'unités compilées en bytecode avec d'autres compilées en code

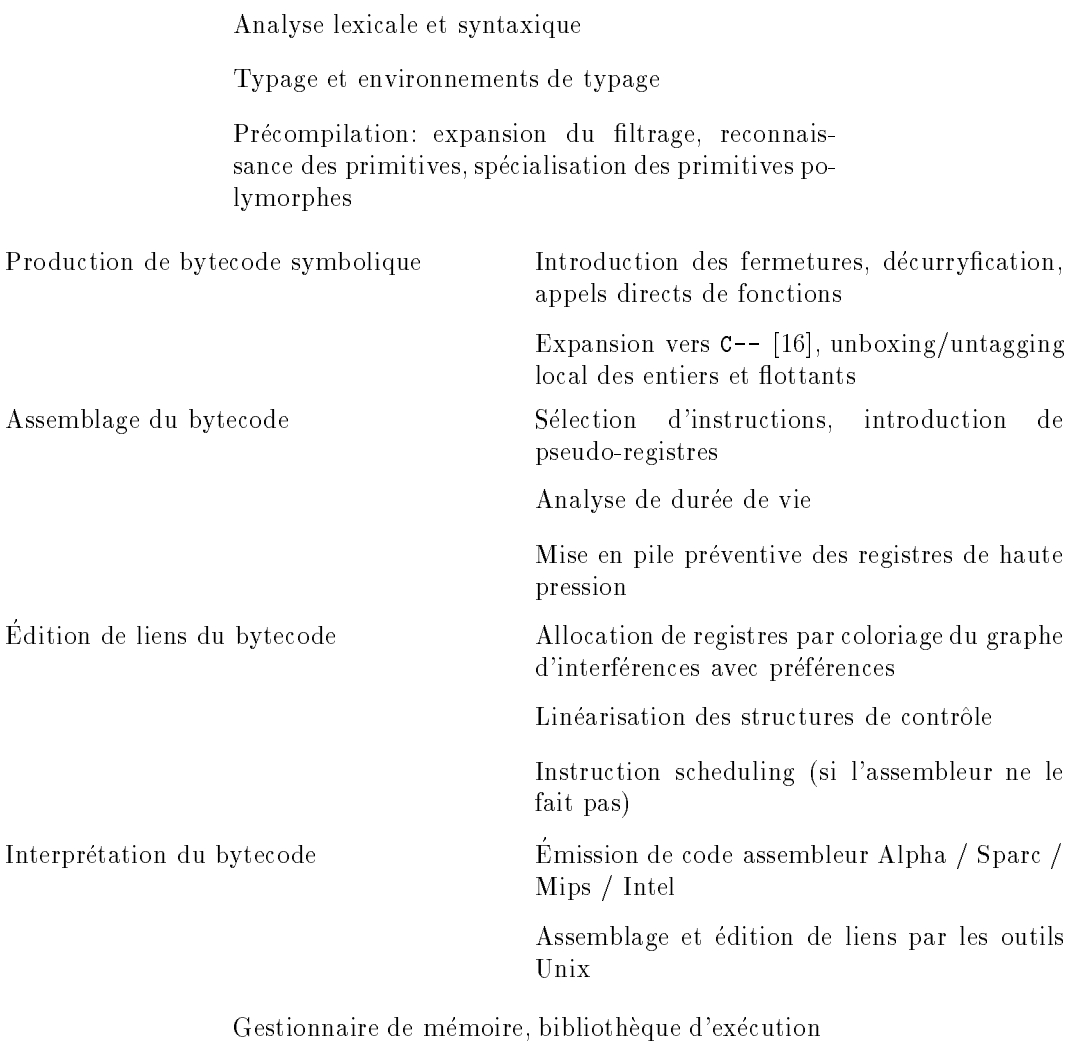

Figure 1 : Structure d'ensemble du compilateur Caml Special Light

natif. Cette interopérabilité n'est pas impossible en théorie, mais nécessite un gros travail d'implémentation. Son utilité pratique reste à démontrer.

3.2 Le compilateur de bytecode

La machine abstraite utilisée par le compilateur de bytecode de Caml Special Light est proche de la machine ZAM utilisée par Caml Light [15]. Toutes deux suivent, dans la terminologie de Peyton-Jones [27], le modèle "push-enter", par opposition au modèle "eval-apply" de la SECD ou de la CAM. C'est-à-dire que les arguments d'une fonction sont évalués et empilés de droite à gauche, puis la fonction est évaluée et appelée; son code teste alors le nombre d'arguments fournis, continue en séquence s'il y en a assez, ou renvoie une fermeture sinon. Ce modèle implémente efficacement le passage d'arguments curryfié, sans aucune allocation si la fonction est complètement appliquée.

Caml Special Light apporte quelques simplifications à la machine ZAM, afin de rendre les cas courants plus efficaces. Par exemple, les deux piles (arguments et retours) de la ZAM sont maintenant fusionnées en une seule. Aussi, le redémarrage des fonctions partiellement appliquées recopie dans la pile les arguments suspendus; ceci permet de déterminer statiquement pour chaque variable si elle est dans l'environnement alloué de la fermeture ou bien sur la pile, et il n'est plus nécessaire de tester la taille de la pile à chaque accès comme dans la ZAM [15].

Chaque instruction du code abstrait occupe maintenant un mot de 4 octets au lieu d'un octet. (Il vaudrait donc mieux parler de "wordcode" plutôt que de "bytecode".) Trois raisons à cette modification: tout d'abord, certains processeurs (Dec Alpha) lisent les mots plus efficacement que les octets. Ensuite, il n'y a plus de problèmes d'alignement des opérandes sur 2 ou 4 octets rencontrés avec la ZAM: tous les opérandes sont maintenant sur 4 octets, alignés. Finalement, au chargement du code dans l'interprète, chaque instruction peut être remplacée par l'adresse du morceau de code qui l'exécute, rendant ainsi le décodage des instructions plus efficace. Cette technique correspond à une évaluation partielle du `switch` effectuant le décodage des instructions, et est connue sous le nom de *threaded code* [5]. Elle n'est pas exprimable en C ANSI, mais s'implémente aisément avec les `goto` calculés de GNU C version 2. Cette astuce accélère d'un facteur 3 environ le décodage des instructions, qui prend plus de la moitié du temps d'interprétation du bytecode.

Bien entendu, la taille des exécutables bytecode augmente – d'un facteur 3,5 environ. Cette augmentation est tout à fait tolérable en pratique: des 60 ko de bytecode pour le compilateur Caml Light, on passe à 260 ko de wordcode pour le compilateur Caml Special Light; cela reste fort raisonnable. La vitesse d'exécution est nettement augmentée par-rapport à Caml Light (voir la section 3.4).

3.3 Le compilateur de code natif

Le compilateur natif de Caml Special Light produit directement du code assembleur. La traduction vers C a souvent été présentée comme le moyen simple et portable de production de code natif efficace. Malheureusement, elle entraîne des temps de compilation élevés et ne

permet pas l'implémentation efficace des exceptions et du GC. De plus, les GC conservatifs que la production de C oblige à utiliser [4, 6] sont non seulement inefficaces mais possiblement incorrects en présence de certaines optimisations du compilateur C [8], rendant ce type d'implémentation essentiellement impossible à prouver. Finalement, l'aspect "héroïque" inhérent à la réalisation d'un compilateur efficace est plus net encore lorsqu'on produit de l'assembleur plutôt que du C.

La production directe d'assembleur pose évidemment un problème de portabilité. Sur les 35000 lignes de Caml composant le compilateur Caml Special Light, environ 1000 sont spécifiques au processeur cible: le module qui décrit le processeur (nombre de registres, conventions d'appel, etc.) et le module qui émet du code assembleur à partir d'une liste de pseudo-instructions. S'y ajoutent environ 200 lignes d'assembleur écrit à la main pour interfacer le code produit et la bibliothèque d'exécution (en particulier, le GC). Le reste du compilateur est indépendant du processeur, et suffisamment paramétré pour s'accommoder de la plupart des architectures RISC d'aujourd'hui et d'au moins une architecture CISC (le Pentium). J'estime à une ou deux semaines le temps de portage vers une nouvelle architecture.

Caml Special Light n'est pas à proprement parler un compilateur optimisant: il n'essaye pas de transformer le code du programmeur pour le rendre plus efficace; mais il fait des efforts considérables pour ne pas introduire d'inefficacités dans le code source et en fournir une traduction fidèle. La force de cette approche est que le comportement du code produit, en particulier son profil d'allocation, reste prévisible et facilement contrôlable par modifications du source; sa faiblesse est que le compilateur ne fait jamais de miracles et ne peut extraire un code décent d'un source essentiellement mal écrit.

Un corollaire immédiat de cette approche est que le compilateur n'alloue dans le tas aucune structure de contrôle sauf les fermetures correspondant aux fonctions de l'utilisateur. Allouer dans le tas les blocs d'activation, comme le fait SML/NJ [2], est un excellent exemple d'inefficacité introduite par le compilateur. Caml Special Light stocke très classiquement les blocs d'activation dans la pile Unix. Cette approche ne se prête pas à l'implémentation efficace de `call/cc` mais se marie bien avec des processus légers (*lightweight threads*).

Les passes du compilateur sont présentées figure 1. Nous détaillons ci-dessous les points saillants du compilateur.

Appels directs de fonctions connues. Le compilateur tient à jour pour chaque variable et pour chaque champ de structure une approximation de la forme "cette variable contient une fermeture de telle fonction avec telles variables libres". Cette approximation sert à transformer les applications de fonctions connues en appels directs au code de ces fonctions. L'appel direct évite un accès dans la fermeture et permet un bien meilleur préchargement (*prefetch*) des instructions sur les processeurs modernes. De plus, l'environnement n'est pas passé en paramètre supplémentaire si on sait que la fonction n'a pas de variables libres.

L'approximation est effectuée de manière très naïve, correspondant à peu près à la première itération de l'analyse 0CFA [33, 30]. Ainsi, dans l'exemple suivant

```
let f x = fun y -> ...
```

```
let g = f 5
let h = (fun x -> x) g
```

les appels à `f` et `g` sont directs, mais l'approximation de `h` est "je ne sais pas". Cette approximation grossière suffit cependant à transformer en appels directs plus de 80% des appels exécutés, sur nos tests; une analyse plus poussée ne s'impose pas.

L'approximation d'une unité de compilation est sauvée dans un fichier, permettant aux clients de cette unité d'appeler directement les fonctions qu'elle définit. Ceci ne compromet pas la compilation séparée: si ce fichier n'existe pas ou n'est plus à jour vis-à-vis du source, des appels indirects sont produits.

Décurryfication Le code produit pour une fonction curryfiée suppose que tous les arguments sont passés simultanément. Un appel direct n'est produit que si le bon nombre d'arguments est fourni. Pour les applications partielles et les appels indirects, la fermeture contient, outre un pointeur vers le code et l'arité de la fonction, un pointeur vers un combinateur général de curryfication, qui reçoit les arguments un à un et les accumule avant de se brancher sur le code de la fonction. Les applications de fonctions inconnues à plusieurs arguments testent l'arité de la fermeture et se branchent directement au code si le bon nombre d'arguments est fourni; sinon, on passe par un combinateur général d'application multiple, qui applique les arguments un à un. L'utilisation de combinateurs généraux partagés par toutes les fonctions du programme fait économiser beaucoup en taille de code par-rapport à la production de points d'entrée spéciaux pour chaque fonction curryfiée, sans perte notable d'efficacité.

Représentations non allouées (unboxing) Les compilateurs Caml Special Light utilisent des représentations de données très traditionnelles: les entiers sont marqués (*tagged*) par le bit de poids faible à 1, les flottants sont alloués dans le tas (*boxed*). Une analyse locale aux fonctions élimine les marquages et allocations inutiles de résultats intermédiaires. Les n-uplets et les records sont systématiquement alloués.

Le compilateur expérimental Gallium [16], l'ancêtre du compilateur natif Caml Special Light, allait beaucoup plus loin dans l'élimination des allocations de flottants et de n-uplets, en s'appuyant sur les informations de typage statique. Cette technologie n'a pas été reprise en Caml Special Light, car elle nécessite un GC et une bibliothèque d'exécution sachant traiter des blocs hétérogènes (mélangeant pointeurs/données marquées et données non allouée/non marquées). Ceci s'obtient en transmettant des informations de typage statique au GC, au prix d'une plus grande complexité et d'un certain ralentissement du GC. De plus, les techniques de non-allocation de Gallium ne sont guère compatibles avec un interprète de bytecode; il n'aurait donc plus été possible de partager le GC et la bibliothèque d'exécution entre les deux compilateurs Caml Special Light.

Caml Special Light évite cependant les allocations dans deux cas fréquents en calcul numérique: les tableaux de flottants, ainsi que les records dont tous les champs sont des flottants, sont représentés "à plat", sans allouer individuellement chaque flottant. Combinée avec l'unboxing local des flottants, cette astuce apporte de très bonnes performances sur le

code numérique. Dans le cas des tableaux, cela signifie que les accès polymorphes à un tableau (type statique `'a array`) deviennent polymorphes ad-hoc: ils testent dynamiquement le type du tableau et effectuent une étape de boxing ou d'unboxing supplémentaire si c'est un tableau de flottants.

Génération de code assembleur La génération de code assembleur est très classique et s'effectue fonction par fonction, avec des conventions d'appel fixes. Les bénéfices à attendre d'une allocation de registres inter-fonctions ne m'ont pas semblé justifier l'effort d'implémentation. Le code intermédiaire est d'abord transformé en séquence de pseudo-instructions machines opérant sur des pseudo-registres en nombre illimité. Les pseudo-registres sont ensuite projetés sur les registres du processeur et sur les emplacements de pile par un classique coloriage du graphe d'interférences avec utilisation de préférences pour réduire les copies [7]. La principale originalité est une passe préalable de mise en pile (*spilling*) des pseudo-registres vivants à travers un point de haute pression comme par exemple un appel de fonction. Après remplacement des structures de contrôle par des branchements, on réordonne les instructions pour réduire la latence par l'algorithme de *list scheduling* [14].

3.4 Mesures de performance

Un article sur un compilateur ne saurait être complet sans mesures de vitesse d'exécution. Les tests résumés figure 2 montrent que le wordcode de Caml Special Light est environ 2 fois plus rapide que le bytecode de Caml Light. Ce facteur est moins élevé (environ 1.5) sur des architectures plus conventionnelles que l'Alpha utilisée pour les mesures.

Le passage du wordcode au code natif accélère les programmes d'un facteur 6 en moyenne. Les accélérations les plus importantes sont obtenues pour les calculs numériques; elles sont moins importantes (facteur 3) pour les calculs symboliques, et encore plus faibles (facteur 2) pour le générateur d'analyseurs lexicaux. Il faut noter que ce dernier programme passe beaucoup de temps dans des primitives écrites en C (hachage, égalité structurelle, automate à pile) qui prennent le même temps d'exécution quel que soit le compilateur Caml utilisé.

La comparaison avec SML of New Jersey est à prendre avec discernement: la version 1.08.5 utilisée pour le test est une version de travail (il n'y a pas encore de distribution "officielle" de SML/NJ pour l'Alpha) et les programmes de test n'ont pas été revus par un spécialiste de SML/NJ pour tirer le meilleur parti du compilateur. Il semble cependant que le compilateur Caml Special Light natif offre de meilleures performances, tout particulièrement sur les calculs numériques, mais aussi sur des programmes comme Boyer et Knuth-Bendix, que je crois représentatifs de l'utilisation de ML en démonstration automatique. Les programmes jouets font apparaître des variations considérables dans les deux sens (sur le crible d'Eratosthène, SML/NJ est 4 fois plus rapide que Caml Special Light, mais 5 fois plus lent sur Takeushi curryfié), on ne peut donc pas en conclure grand-chose.

Le couple Bigloo / Camloo (un compilateur de Scheme vers C et son frontal Caml) offre d'excellentes performances sur les programmes qui allouent peu (les temps pour Quicksort sans vérifications des bornes sont très bons), mais brille moins sur les programmes qui

Compilateurs utilisés:

I Caml Light version 0.7
II Caml Special Light 1.06, wordcode
III Caml Special Light 1.06, code natif (option `CAMLRUNPARAM=o=100`)
IV Standard ML of New Jersey 1.08.5
V Bigloo 1.7 et Camloo 0.3 (options `-unsafeatsv -O4`)

Temps d'exécution (user + system), en secondes, sur une Dec Alpha 3000/300X, 175 Mhz, 96 Mo de mémoire, OSF/1 2.0.

Programme de test	(I)	(II)	(III)	(IV)	(V)	
Fibonacci	6.87	2.89	0.32	1.46	0.45	Programmes
Takeushi, curryfié	8.90	3.81	0.36	1.98	0.60	jouets
Takeushi, décurryfié	12.3	6.15	2.50	1.94	11.3	(tests
Crible d'Eratosthène	7.18	5.94	2.50	0.60	1.87	ponctuels
Quicksort	31.9	9.59	1.10	1.07	1.49	de traits
Quicksort (1)	16.4	7.20	0.93	—	0.69	du langage)
Jeu du solitaire	20.3	4.41	0.54	1.46	1.03	
Jeu du solitaire (1)	8.85	3.35	0.39	—	0.51	
FFT	96.9	61.9	4.01	46.4	28.2	Calcul
FFT (1)	65.7	58.2	3.74	—	26.8	numérique
Nucleic	14.2	11.5	1.26	— (2)	16.2	
Boyer	4.52	3.17	0.99	1.97	1.90	Calcul
Knuth-Bendix	31.3	15.5	4.45	8.33	29.4	symbolique
Générateur de lexeurs	2.84	1.83	0.89	— (3)	1.71	Compilation

(1) Compilés sans vérification des accès aux tableaux (option non disponible en SML/NJ)
(2) Ce programme ne tourne pas en SML/NJ 1.08.5
(3) Programme non traduit en SML

Facteurs d'accélération (moyennes géométriques):

	Caml Light / CSL bytecode	CSL bytecode / CSL natif	SML NJ / CSL natif	Bigloo / CSL natif
Tous les tests	1.96	6.23	2.01	2.47
Programmes jouets	2.44	6.35	1.52	1.45
Calcul numérique	1.30	13.0	11.6	8.65
Calcul symbolique	1.70	3.34	1.93	3.56
Compilation	1.55	2.06	—	1.92

Figure 2 : Vitesses d'exécution

allouent beaucoup (Knuth-Bendix), déclenchent beaucoup d'exceptions (Knuth-Bendix), ou font du calcul flottant (FFT, Nucleic). Il est probable qu'on atteigne ici les limites de la compilation vers C et des GC conservatifs. En particulier, pour les tests de calcul symbolique, le code produit par Bigloo n'est en moyenne pas plus rapide que le bytecode de Caml Special Light.

4 Directions futures

Le système Caml Special Light est loin d'être figé et reste susceptible d'évolutions majeures en fonction des utilisations futures. En particulier, il reste à mettre le système de modules à l'épreuve de gros développements, et à l'affiner en conséquence. Le travail sur le langage devrait continuer, en particulier en direction d'un calcul d'objets bien intégré à l'inférence de types et au système de modules. Sur le compilateur, l'extension la plus urgente est sans doute une passe d'expansion (*inlining*) de fonctions.

Références

[1] Parameterized types are not enough to implement sets as ordered binary trees. Discussion sur la liste `caml-list`, consultable sur le Web, `http://pauillac.inria.fr/caml/caml-list/0096.html`, mai 1993.

[2] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[3] Andrew W. Appel et David B. MacQueen. Separate compilation for Standard ML. In *Programming Language Design and Implementation 1994*, pp. 13–23. ACM Press, 1994.

[4] Joel F. Bartlett. Compacting garbage collector with ambiguous roots. Rapport technique, DEC Western Research Laboratory, 1988.

[5] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, juin 1973.

[6] Hans-Juergen Boehm. Space efficient conservative garbage collection. *SIGPLAN Notices*, 28(6):197–206, juin 1993.

[7] Preston Briggs, Keith D. Cooper, et Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, mai 1994.

[8] David R. Chase. Safety considerations for storage allocation optimizations. *SIGPLAN Notices*, 23(7):1–10, juillet 1988.

[9] Pierre Crégut et David B. MacQueen. An implementation of higher-order functors. In *Proc. 1994 Workshop on ML and its applications*, pp. 13–21. Rapport de recherche 2265, INRIA, 1994.

[10] Régis Cridlig. An optimizing ML to C compiler. In *Proceedings of the 1992 workshop on ML and its applications*, pp. 28–36, 1992.

[11] Robert Harper, Peter Lee, Frank Pfenning, et Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Rapport technique CMU-CS-94-116, Carnegie-Mellon University, 1994.

[12] Robert Harper et Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st symposium Principles of Programming Languages*, pp. 123–137. ACM Press, 1994.

[13] Gérard Huet et al. The Coq proof assistant. Logiciel et documentation disponibles sur le Web, `http://pauillac.inria.fr/coq/`, 1995.

[14] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, juillet 1988.

[15] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Rapport technique 117, INRIA, 1990.

[16] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th symposium Principles of Programming Languages*, pp. 177–188. ACM Press, 1992.

[17] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pp. 109–122. ACM Press, 1994. Version complétée soumise à publication, disponible sur le Web, `http://pauillac.inria.fr/~xleroy/manifest-types.dvi.gz`.

[18] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd symposium Principles of Programming Languages*, pp. 142–153. ACM Press, 1995.

[19] Xavier Leroy. A syntactic theory of type generativity and sharing. À paraître dans le *Journal of Functional Programming*. Disponible sur le Web, `http://pauillac.inria.fr/~xleroy/syntactic-generativity.dvi.gz`. Un résumé est paru dans *Proc. 1994 Workshop on ML and its applications*, rapport de recherche 2265, INRIA, pages 1–12, 1995.

[20] Xavier Leroy et Damien Doligez. The Caml Special Light system. Logiciel et documentation disponibles sur le Web, `http://pauillac.inria.fr/csl/`, 1995.

[21] Xavier Leroy, Damien Doligez, et al. The Caml Light system, release 0.7. Logiciel et documentation distribués par FTP anonyme depuis `ftp.inria.fr`, 1995.

[22] Barbara Liskov et John Guttag. *Abstraction and specification in program development*. MIT Press, 1986.

[23] David B. MacQueen. Modules for Standard ML. In Robert Harper, David B. MacQueen, et Robin Milner (rédacteurs), *Standard ML*. University of Edinburgh, technical report ECS LFCS 86-2, 1986.

[24] David B. MacQueen. The implementation of Standard ML modules. In *Lisp and Functional Programming 1988*, pp. 212–223. ACM Press, 1988.

[25] David B. MacQueen et Mads Tofte. A semantics for higher-order functors. In D. Sannella (rédacteur), *Programming languages and systems – ESOP '94*, Lecture Notes in Computer Science, volume 788, pp. 409–423. Springer-Verlag, 1994.

[26] Robin Milner, Mads Tofte, et Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[27] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[28] François Pottier. Implémentation d'un système de modules évolué en Caml Light. Rapport de recherche 2449, INRIA, 1995.

[29] Eugene J. Rollins. SourceGroup: a selective recompilation system for SML. In *Third international workshop on Standard ML*. Carnegie-Mellon University, 1991.

[30] Manuel Serrano. *Vers une compilation portable et performante des langages fonctionnels*. Thèse d'université, Université Pierre et Marie Curie, Paris 6, décembre 1994.

[31] Manuel Serrano et Pierre Weis. $1 + 1 = 1$: an optimizing Caml compiler. In *Proc. 1994 Workshop on ML and its applications*, pp. 101–111. Rapport de recherche 2265, INRIA, 1994.

[32] Zhong Shao et Andrew Appel. Smartest recompilation. In *20th symposium Principles of Programming Languages*, pp. 439–450. ACM Press, 1993.

[33] Olin Shivers. Control-flow analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, juillet 1988.

[34] Pierre Weis et Xavier Leroy. *Le langage Caml*. InterÉditions, 1993.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399