$1 + 1 = 1$: an optimizing Caml compiler

M. Serrano P. Weis

INRIA Rocquencourt [*]

Abstract

We present a new Caml compiler, which was obtained by an original approach: a simple pipeline between two existing compilers, each one devoted to half of the compilation process. The first compiler is a Caml compiler, it is in charge of the front end, and ensures compatibility. The second compiler is an optimizing Scheme compiler, it constitutes the back end, and ensures efficiency. These are Caml Light 0.6 byte-code compiler and a Scheme compiler (Bigloo). Using this technology, we were able to write the optimizing compiler in only two man-months. The new compiler is bootstrapped, fully compatible with the Caml Light 0.6 compiler, and features interesting inter-module optimizations for curried functions. It produces efficient code, comparable with the one produced by other ML compilers (including SML/NJ). Our new compiler, Bigloo, is freely available [1].

KEYWORDS: Caml, Scheme, ML, compilation, functional languages.

Introduction

The Caml Light compiler is a byte-code compiler. This has many advantages: high level of portability, small memory requirements (the core image of the compiler is typically 400 Ko), and very fast code generation. These good properties lead to usable Caml Light versions on personal computers. On the other hand, a byte-code compiler appears to produce slow executable programs (namely from two to twenty times slower than assembly code programs). Hence the need for an optimizing compiler.

It must be clear that the optimizing compiler will compile much more slowly than the regular compiler: so in practice we do want the best of the two technologies, the fast byte-code compiler to develop and test programs, and the optimizing compiler to get efficient executables, as soon as programs are ready to work. In this view, the optimizing compiler just performs the `-O` option of the byte-code compiler. This has an important implication: this compiler must be fully compatible with the byte-code compiler.

There have been several attempts to obtain such a compiler: CeML by Emmanuel Chailloux [4], and Camlot by Régis Cridlig [6]. Both share the same compilation strategy: C code generation, and their performances are good. CeML was not designed to be fully compatible with Caml, it was developed to prove that ML could be efficiently compiled into C. Thus, CeML has some minor differences with the core of the Caml language (for instance structural equality). The Camlot compiler treats the Caml core language, but does not feature exactly the same syntactic extensions as the Caml Light system [21] (for instance the streams feature is not completely supported). As a consequence, these two compilers failed to be fully compliant with the Caml Light system, and none can be considered as an alternative to the Caml Light byte-code compiler.

We present the design decisions and the methodology that allow us to get a fully compatible optimizing compiler for Caml.

Section 1 gives an overview of the technology applied to get the new compiler. Section 2 gives the overall architecture of the compiler. Section 3 describes the back-end compiler. Sections 4, 5, and 6 detail the remainder of the compiler. We conclude by a short comparison with related works and finally give some benchmarks.

[*]Author's address: INRIA, BP 105, 78153 Le Chesnay, Manuel.Serrano@inria.fr, Pierre.Weis@inria.fr

[1]Anonymous ftp on *ftp.inria.fr* (192.93.2.54), file lang/caml-light/bcl1.6c.0.2unix.tar.Z.

1 The compiler's technology

To achieve this compatibility goal we choose a simple scheme: the redirection of compilers. Our optimizing compiler shares the whole front end of the Caml Light byte-code compiler. This trivially ensures full compatibility for the input source code. To gain efficiency, we just have to replace the byte-code execution by machine code execution. Our optimizing compiler shares all the back end of a good optimizing compiler: the Bigloo Scheme compiler [13]. This ensures very fast code execution.

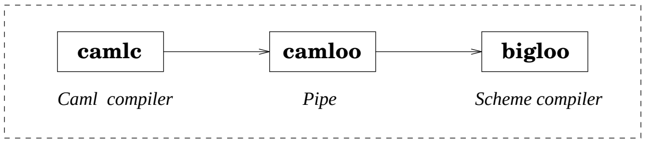

Figure 1: $1 + 1 = 1$

In short, we obtain our new compiler from two existing compilers, hence the title of this paper. We simply connect the two compilers by plugging the output of the first one to the input of the second, in the spirit of Unix pipes. We slightly modify the front end compiler to stop before code generation, but the back end compiler is left unchanged. The main job has been to implement the pipe which is at least one order of magnitude simpler than re-engeneering a back-end or a front-end.

Thus, we stop the Caml Light compiler at the end of its compilation process by omitting the byte-code generation. This phase is replaced by a text dump of a λ-like tree structure representing the program.

The pipe is just a mapping of the λ-like tree onto a Scheme program (we name this translator *Camloo*).

The implementation effort to get the new compiler has been 3000 source code lines (400 lines of Caml, 2600 lines of Scheme).

2 The global architecture

We explain how to compile Caml modules, that is how to compile a module implementation and a module interface.

2.1 Compiling implementations

When compiling the implementation part of a Caml module (a `.ml` file), `camlc` processes it as usual, until the semantics of the program is explicit: `camlc` performs syntax analysis, type checking, name resolution (qualification of identifiers), pattern matching, and production of $\lambda-$code. This $\lambda-$code is then written into a `.lam` file.

Then Camloo translates the `.lam` file into a Scheme file which is compiled by the regular Bigloo compiler.

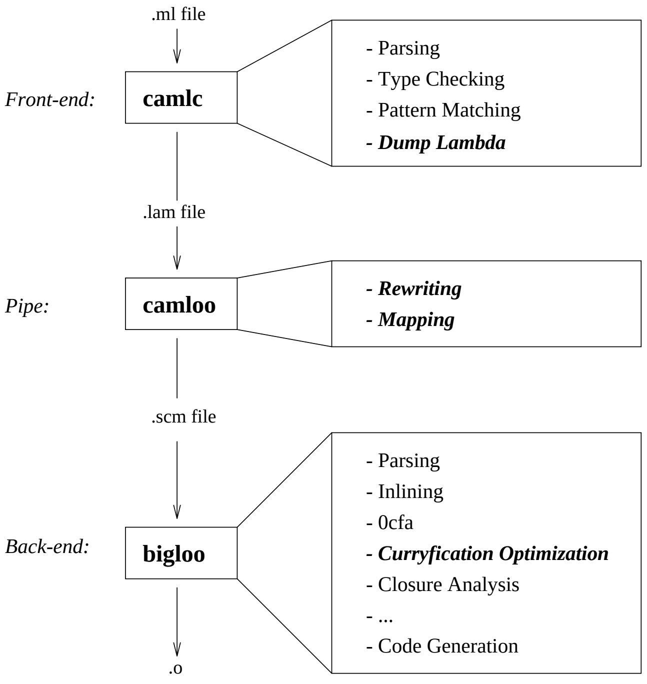

Figure 2: Bigloo: compiling a Caml implementation

In the figures, we write in **boldface** the passes we add to obtain the new compiler.

2.2 Compiling interfaces

For the interface part of a Caml module (a `.mli` file), `camlc` processes it as usual, generating a compiled interface (a `.zi` file), and in addition a `.sci` file containing importation clauses for the Bigloo module system. Compiled interfaces serve as usual to ensure incremental recompilation of Caml modules, while `.sci` files are used by Bigloo to compile implementations.

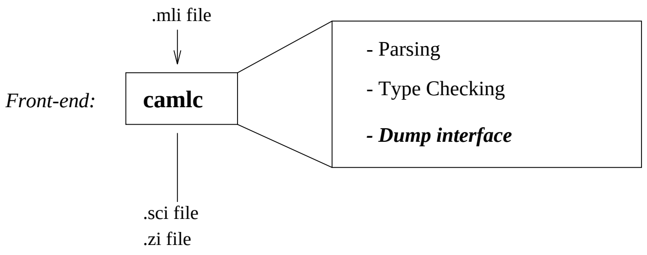

Figure 3: Bigloo: compiling a Caml module interface

2.3 Compiling Scheme or Caml

Bigloo has a builtin preprocessing mechanism which permits addition of user-defined passes to the compiler front-end. Since the invocation of these preprocessing passes can be associated to file suffixes, `camlc` and Camloo are invoked when Bigloo encounters a `.ml` or `.mli` file: seen from Bigloo, the Caml Light front end is just a new pass added before the compilation of Scheme code (in the spirit of macro expansion passes). Then we just have to invoke Bigloo to compile a Caml module. That's why the new compiler is named Bigloo as well, and since it automatically turns to Scheme compilation for a `.scm` file, the new compiler is a *Caml and Scheme* compiler.

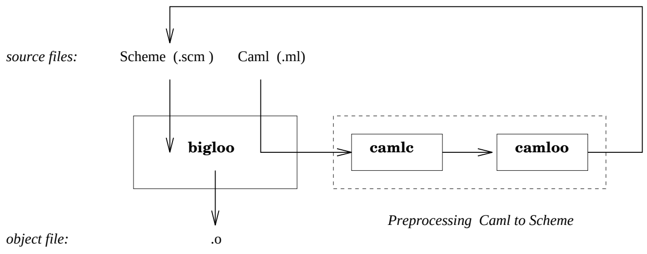

Figure 4: Bigloo: a Scheme *& Caml* compiler

3 The Bigloo Scheme compiler

Bigloo [13, 16] is one of the best Scheme compilers available. It produces very efficient code via a Scheme to C translation. It uses many powerful techniques to optimize functional programs, such as *0cfa* analysis [17, 14], open coding, common subexpressions elimination, optimized closure allocation, $\lambda-$lifting, compile-time β-reduction, constant folding, recursive calls optimization. In short, this compiler never allocates structure for control-flow (loops are compiled as C loops, direct calls to global functions are compiled as C function calls, and therefore the allocation of closures is infrequent).

Let us describe some of these optimizations and look at some examples of produced code. In effect, the Bigloo compiler produces readable C code: the code given in the following examples has been obtained by a mere alpha conversion from the actual code produced by the Bigloo compiler. In this C code, type `obj` is the C union that implements Scheme values and upper-case identifiers correspond to C macros.

3.1 Open-coding

Open-coding (or inlining) is a technique which replaces a call to a function by the body of this function. Bigloo inlines functions when some explicit `inline` directives are written in the source file or when these functions verify some pragmatic properties to prevent code explosion: the functions to inline must be defined in the module being compiled, and the size of their bodies must be small enough. Let us give some examples to illustrate the transformation.

```
(define (succ x) (+ x 1))

(define (successor y)
   (succ y))
```

the definition of `successor` is translated into:

```
(define (successor y)
   (let ((x y))
      (+ x 1)))
```

which is then translated into:

```
(define (successor y)
   (+ y 1))
```

Even recursive functions can be inlined [2]: to inline the recursive function f into function g, Bigloo defines f as a local function inside the body of g. To prevent the compiler from infinite inlining, no inlining of a recursive function can occur within its own body. For short, programs of the form:

```
(define f (lambda args body))
(define g (lambda args ... ( f ...) ...))
```

are translated into:

```
(define g (lambda args (letrec (( f args body))
                            ... ( f ...) ...)))
```

For instance, let us define `map-succ` using the `map` functional:

```
(define (map f l)
   (if (null? l)
       '()
       (cons (f (car l)) (map f (cdr l)))))

(define (succ x) (+ x 1))

(define (map-succ l)
   (map succ l))
```

When inlining `map` into `map-succ` the compiler discovers that `map` is self-recursive, so it inlines it using a local definition:

```
(define (map-succ l)
   (letrec ((map (lambda (f l)
                    (if (null? l)
                        '()
                        (cons (f (car l))
                              (map f (cdr l)))))))
      (map succ l)))
```

A further pass of the compiler demonstrates that the formal parameter `f` is a loop invariant, so `f` is substituted by its actual value (compile-time $\beta-$reduction). We get:

```
(define (map-succ l)
   (letrec ((map (lambda (l)
                    (if (null? l)
                        '()
                        (cons (succ (car l))
                              (map (cdr l)))))))
      (map l)))
```

Then inlining of function `succ` occurs, and we finally get a very efficient equivalent piece of code:

```
(define (map-succ l)
   (letrec ((map (lambda (l)
                    (if (null? l)
                        '()
                        (cons (+ 1 (car l))
                              (map (cdr l)))))))
      (map l)))
```

3.2 Closure analysis

The main design effort for the Bigloo compiler has been to compile functions as well as possible: in our mind this implies to map Scheme functions to C functions (or even to C loops) and to work hard to avoid heap-allocation of closures as much as possible.

Compiling functions to C loops

Thus, Bigloo usually compiles Scheme functions as much as possible into C loops or C functions. When functions escape, that is when functions are used as first-class values, this efficient scheme does not apply, thus the compiler allocates heap space for function environments. Bigloo uses flat closures: the environment part is a heap block containing exactly the free variables of the functions. Despite the larger heap-allocation needed for flat closures, we do not use linked environments since they lead to memory leaks which cannot be circumvented by the Scheme user.

The closure analysis is able to map functions into C loops even if they are mutually recursive. For instance:

```
(letrec ((odd? (lambda (n)
                  (if (= n 0)
                      #f
                      (even? (- n 1)))))
         (even? (lambda (m)
                  (if (= m 0)
                      #t
                      (odd? (- m 1))))))
   (odd? 10))
```

is compiled into:

```
obj n, m;
{
   n = 10;
_odd:
   if( n == 0 )
      return FALSE;
   else
   {
      m = SUB( n, 1 );
   _even:
      if( m == 0 )
         return TRUE;
      else
      {
         n = SUB( m, 1 );
         goto _odd;
      }
   }
}
```

Compiling functions to C functions

Functions which are not the result of other functions or which are not passed as arguments (functions which don't escape) are mapped to C functions. For example `map-succ` is compiled into:

```
obj map_succ( obj l )
{
   return map( l );
}

static obj map( obj l )
{
   if( NULLP( l ) )
      return NIL;
   else
   {
      obj r;
      r = CAR( l );
      {
         obj obj1;
         obj obj2;
         obj1 = ADD( 1, r );
         obj2 = map( CDR( l ) );
         return MAKE_PAIR( obj1, obj2 );
      }
   }
}
```

Heap-allocated closures

Only the escaping functions are allocated in the heap. For instance, the functional composition of two functions returns a $\lambda-$expression as a value. Bigloo create an heap-allocated closure for the expression of line *2*:

```
1: (define (o f g)
2:    (lambda (x)
3:       (f (g x))))
```

As mentioned above, Bigloo's closures are arrays. More precisely, the first slot of a closure is a pointer to a C function which reflects the code part of the function. The second slot stores the function arity, the remainder of the array is devoted to the free variables of the function. (In ML this arity slot is indeed useless since functions have only one argument, but we need it in Scheme since there exist n-ary functions.)

Then the preceding example is compiled as:

```
obj o( obj f, obj g )
{
   obj clo;

   clo = make_closure( lambda_1, 1, 2 );
   CLOSURE_ENV_SET( clo, 1, g );
   CLOSURE_ENV_SET( clo, 0, f );
   return clo;
}

obj lambda_1( obj clo, obj x )
{
   obj f, g;
   f = CLOSURE_ENV_REF( clo, 0 );
   g = CLOSURE_ENV_REF( clo, 1 );
   {
      obj aux;
      aux = CLOSURE_ENTRY( g )( g, x );
      return CLOSURE_ENTRY( f )( f, aux );
   }
}
```

Even functions used as first class values can be compiled without heap-allocation using the following 0cfa optimization.

3.3 0cfa

The 0cfa is a static analysis which reveals static parts of control flow. The analysis implemented in Bigloo is close to the one described by O. Shivers in [17]. Since the Bigloo compiler does not use the CPS compiling style, the 0cfa analysis is not devoted to the optimization of the CPS style code generated by the compiler: it is used instead to minimize the number of heap-allocated closure [15].

Computed calls turned to direct calls

The 0cfa approximates for each variable the set of its possible values. In particular, it computes, for each function call in a program, an approximation of the set of all the functions that can be dynamically invoked at this application site. When this set is restricted to only one function, this function can be called directly. For instance, consider a functional F having as parameter a function f, which is called within the body of F. If the 0cfa analysis can prove that there is only one possible value v for f, then v is called instead of f, which is no longer a parameter for F. This saves one parameter passing and replaces a costly call to an unknown function by a direct jump.

Let us illustrate this phenomenon with the definition of a function by functional composition:

```
(define (o f g)
   (lambda (x)
      (f (g x))))

(define succ (lambda (a) (+ a 1)))

(define succ2 (o succ succ))
```

Since `succ2` can be used out of this program fragment `o` creates a closure for `succ2`. Nevertheless the 0cfa proves that `f` and `g` in the code of `o` are `succ`. Thus, the expression `(o succ succ)` is compiled as `(lambda (x) (succ (succ x)))`. After inlining of `succ`, we get for `succ2` the following C code:

```
{
   obj clo;
   clo = make_closure( lambda_1, 1, 0 );
   succ2 = clo;
}

obj lambda_1( obj clo, obj x )
{
   return ADD( ADD( x, 1), 1 );
}
```

If the definition and uses of `succ2` are local, then a call to `succ2` is an application of the very lambda expression returned by `o`, and thus there is no need to allocate a closure. For instance

```
(let ((succ2 (o succ succ)))
   (succ2 4))
```

is compiled using a dummy closure bound to UNSPECIFIED (the special Bigloo value which represents dummies):

```
{
   obj clo;
   clo = UNSPECIFIED;
   succ2 = clo;

   lambda_1( succ2, 4 )
}

static obj lambda_1( obj clo, obj x )
{
   return ADD( ADD( x, 1), 1 );
}
```

From higher-order to first-order code

In some cases the 0cfa is able to automatically replace an higher-order program by an equivalent first-order program. As an example, we study a Scheme higher-order definition of the factorial function without explicit recursion:

```
(define (fact n)
   (let ((f (lambda (g m)
               (if (= m 0)
                   1
                   (* m (g g (- m 1)))))))
      (f f n)))
```

Even in this puzzling program, the 0cfa analysis is able to turn the local higher-order definition of f into a first-order expression. The 0cfa proves that the formal parameter g is always the actual value f, and obtains:

```
(define (fact n)
   (letrec ((f (lambda (m)
                  (if (= m 0)
                      1
                      (* m (f (- m 1)))))))
      (f n)))
```

Optimizing currying

0cfa analysis can also remove useless curryings. For instance, this analysis is able to avoid the creation of intermediate closures when calling a curried function with several arguments.

```
(define plus
   (lambda (x)              ;; lambda-1
     (lambda (y) (+ x y))   ;; lambda-2
   ))

(define add (lambda (x y) ((plus x) y)))
```

The 0cfa tells the compiler that the only result of an invocation to plus is the function *lambda-2*. So (plus x) could be replaced by a call to *lambda-2* (which has been $\lambda-$lifted [2]). We eventually get the following equivalent program which does not require any closure allocation.

```
(define plus
   (lambda (x)
      x))

(define lambda-2
   (lambda (env y)
      (+ env y)))

(define add (lambda (x y) (lambda-2 (plus x) y)))
```

[2] Local functions are $\lambda-$lifted when they cannot be compiled as C loops

0cfa restrictions

The 0cfa analysis is local to a module: when a value can be used from outside the current module, then the analysis cannot keep track of this value usage. Thus, the analysis must be pessimistic and for instance must allocate a closure for a $\lambda-$expression, simply because it is impossible to know where this $\lambda-$expression is applied.

3.4 CSE

The common subexpression elimination (CSE) attempts to share computation of pure expressions (expressions without side effects). In Bigloo this optimization is realized in three passes:

- Purity analysis: it is a simple annotation on expressions giving a rough approximation of the side effects that occur during the evaluation of the given expression. These annotations are exact for primitives and properly propagated to functions. This analysis is not higher-order and is local to modules (imported functions are considered impure).
- Let bindings: this rewriting of the abstract syntax tree makes explicit the control flow (in the spirit of cps). It binds subexpressions to variables using let.
- Abstract syntax tree pruning: this pass is a recursive descent in the abstract syntax tree, which removes pure expressions already computed, replacing the expressions by the variable they are bound to. This is achieved using a stack of previously encountered pure expressions.

Consider the following definitions:

```
(define (add x y)
   (+ x y))

(define (double-add x y)
   (+ (add x y) (add x y)))
```

The function add is proved to be pure since + is a pure primitive. After let binding, this program is:

```
(define (add x y)
   (let ((+-res (+ x y)))
      +-res))

(define (double-add x y)
   (let ((z1 (add x y)))
      (let ((z2 (add x y)))
         (let ((+-res (+ z1 z2)))
            +-res))))
```

After pruning, we get:

```
(define (add x y)
   (let ((+-res (+ x y)))
      +-res))

(define (double-add x y)
   (let ((z1 (add x y)))
      (let ((+-res (+ z1 z1)))
         +-res))))
```

This simple first-order CSE suffices to remove many redundant dynamic type checks. Even more tests can be eliminated by a special treatment of conditionals involving type predicates. (When encountering `(if test? then-exp else-exp)` the tree walker recursively enters `then-exp` remembering that `test?` is statically known to be true, and conversely for the `else-part` expression.) This dynamic type checks elimination is useless when compiling ML code, but CSE remains a valuable optimization since it removes duplicated computations such as multiple accesses to the same field in a data structure.

4 Camloo as pipeline

As seen above, the two compilers are connected using text files. The last pass of our modified Caml Light compiler dumps into a file a λ-like expression representing the Caml source program. As it is, this file cannot be compiled by Bigloo since it is not a Scheme source text. Thus a small Scheme program (naturally called Camloo) translates this dump into a compilable Scheme source file: a Bigloo module. This translation is just a mapping of ML constructions and values to appropriate Scheme constructions and values. When designing this translation, correctness is relatively easy, since Scheme and ML are not so far from each other. In fact the main difficulty is to get an efficient translation. However, in this case, efficiency comes from simplicity: we have to map ML constructs to their *natural counterparts* in Scheme, without any extra encoding. Otherwise, the efficiency could be jeopardized, even if the semantics is preserved, since the back-end compiler cannot optimize this encoded program. For instance, it is mandatory to map ML functions to Scheme $n-$ary functions, and *not* systematically to unary ones, in order to benefit from the optimizations performed by the back-end to call functions with multiple arguments.

4.1 Mapping ML to Scheme: an example

Let us start by a simple example: consider the following `len.ml` file defining the `list_length` function.

```
let rec list_length = function
    [] -> 0
  | x :: l -> 1 + list_length l;;
```

The $\lambda-$like representation ($\lambda-$code), as obtained by the command `camlc -dump len.ml`, is the following `len.lam` file:

```
(lprim
 (pset_global
  (qualifiedident "len" "list_length"))
 ((lfunction
   (lswitch 2 (lvar 0)
    (((constrconstant
        (qualifiedident "builtin" "[]") 0 2)
        (lshared (lconst (scatom 0)) -1))
     ((constrregular
        (qualifiedident "builtin" "::") 1 2)
        (lshared
         (lprim paddint
           ((lconst (scatom 1))
            (lapply
             (lprim
              (pget_global
                 (qualifiedident "len"
                                 "list_length"))
              ())
             ((lprim (pfield 1) ((lvar 0)))))))
       -1)))))))
```

This expression is a bit hard to read: let's just say that it defines the global variable `list_length` from the module `len`, that the pattern matching has been expanded (`lswitch`), constructor names have been explicitized (as in `"builtin" "::"`), and primitives are qualified (`paddint` instead of `+`). As the careful reader may have already noticed, the Caml Light compiler uses the De Brujin's notation, so that local variable names have disappeared (replaced by indexes to their $\lambda-$binder).
From this $\lambda-$code the Camloo mapping produces the following Scheme file (`len.scm`), which is a plausible hand-crafted program.

```
(define list_length@len
  (lambda (x1)
    (if (null? x1)
      0
      (+fx 1 (list_length@len (cdr x1))))))
```

This example illustrates the direct mapping of functions, lists and integers to their Scheme correspondents. Moreover, notice that the Scheme code does not use the generic addition but the faster integer addition (straightforward implementation of the `paddint` primitive call produced by the Caml Light compiler).

4.2 The general mapping

We describe here the general scheme that performs the mapping. We will see later a few exceptions to the general scheme, used to map even more directly some ML data structures (e.g. lists) and syntactic constructs (e.g. curried functions).

Mapping values

- Functional values: there are no difficulties to map Caml functions to Scheme procedures: both languages are $\lambda-$languages with higher-order functionality.
- Basic Caml values such as integers, strings, characters, arrays, floating point numbers, map to the corresponding Bigloo basic values. However, a small problem arose with characters, since Caml Light characters are directly represented as integers: some primitives have been changed to reflect the nontrivial coercions between Bigloo integers and Bigloo characters.
- Constant data constructors map to Bigloo constants (for instance, `true` and `false` map to Scheme booleans `#t` and `#f`).
- Values built with non-constant data constructors map to Bigloo vectors: the application of constructor `C` to arguments `x, y, z` is translated to the Bigloo vector `#(x y z)`. The `C` constructor is mapped to the corresponding Bigloo runtime tag attached to the vector (8 bits are devoted to this tag).

- Exceptional values: values built with "extensible" data constructors are also mapped to vectors.

Mapping of exceptional values deserves some explanations. Since exceptions are generative, each one must possess a unique tag. This tag is allocated during the link phase of the compilation in Caml Light. Fortunately Scheme offers the notion of symbols which are names associated to unique objects. Thus, ML exceptions give raise to the creation of a Lisp symbol which stands for the stamp of the exception. The symbol's name is compound using the name of the exception, the static stamp of the exception definition (counting the number of redefinition of the exception), and the name of the module that defines this exception.

For instance, the constant exception `Out_of_memory` from the library module `exc`, which is evidently not redefined, has symbolic stamp `'Out_of_memory1@exc`. For non constant exceptions, the symbol is stored is the first slot of the vector representing the exceptional value: `Failure "hd"` corresponds to the vector `#(Failure1@exc "hd")`.

4.2.1 Mapping `try` ...`raise`

There is no difficulty for the rest of the Caml language constructs (mainly loops and conditionals); the last problem is the exception handling. We could have adopted a trivial implementation of `try` ...`with` ... and `raise` with the powerful Scheme `call/cc` library function. It suffices to define a `try-with` macro (the variable `raise` is bound to the current handler), as suggested by:

```
(define raise (lambda (x)
                (error "uncaught exception" x)))

(define-macro (try-with computation match-handler)
  `((call/cc
     (lambda (return)
       (let ((previous-raise raise))
         (set! raise
               (lambda (exc)
                 (set! raise previous-raise)
                 (return
                  (lambda ()
               (,match-handler exc)))))
         (let ((result ,computation))
           (set! raise previous-raise)
           (lambda () result)))))))
```

Unfortunately, Scheme implementations using stacks are known to have a slow `call/cc` function [11]. Producing C code using the C stack, Bigloo is no exception. Since Caml code may use heavily `try` and `raise` constructs, we need a more efficient implementation of exception handling. In C, we classically use `setjmp` and `longjmp`, but once more these functions are not designed to be used as intensively as in the C code generated from ML sources. This results in poor runtime behavior for programs which heavily use `try` and `raise`. Thus, some ports of our compiler directly use assembly code to implement exception handling (e.g. 19 lines for the Sparc processor).

Pattern matching

Camloo is *not* in charge to map Caml pattern matching to Scheme expressions. Pattern matching is expanded into basic switches and conditionals by the Caml Light compiler. Camloo just maps these two constructions to their natural Scheme equivalents.

We gave up to map Caml pattern matching to the Scheme pattern matching featured by Bigloo for sake of compatibility: this way our new compiler offers all the Caml Light specific extensions such as streams.

4.3 Modules

Caml Light modules are mapped by Camloo onto Bigloo modules. Bigloo modules are self contained: interface and implementation are in the same file. When compiling a Caml Light module interface, Camloo produces two files: a regular "`.zi`" compiled interface file, and a Bigloo module with an empty implementation ("`.sci`" file) which contains the definitions of primitives found into the Caml Light interface. When compiling a Caml Light module implementation, Camloo produces one Bigloo module which contains the body of the Caml Light implementation, and a Bigloo module header containing the prototypes of all exported functions and variables.

This mapping of Caml Light modules to Bigloo modules obliges to compile only modules having an implementation. This is a restriction to the Caml Light module language, but every Caml Light *executable* program is compilable by Bigloo. However, this restriction is of no importance in our case, since our compiler is designed to produce efficient executables for already fully developed programs.

4.4 Optimizing the mapping

Natural mapping of curried functions

The general mapping for Caml functions as described above is correct, but such a trivial mapping would be inefficient, since ML curried functions would lead to multiple function invocations, with allocation of intermediate closures.

In effect, ML does not possess n-ary functions. This is usually circumvented using either tuples or curried functions. From a theoretical point of view, these approaches give the same expressive power as n-ary functions. From the compiler point of view, it is mandatory to map those encoded n-ary functions to truly n-ary functions. Since in Caml Light the usual encoding of n-ary functions is currying, Camloo has to map Caml Light curried functions to Scheme n-ary functions. It would also be worthwhile to map functions using tupled arguments to n-ary functions, but this remains to be done.

As mentioned above, the 0cfa analysis used by the Bigloo compiler already optimizes curried application, but this analysis is not powerful enough, since it only works for local (or local to a module) functions. Thus, it remains of paramount importance to design a general optimization that maps Caml Light functions with n curried arguments to Scheme procedures with n arguments, particularly for global functions.

This new optimization requires the collaboration of Camloo and Bigloo. Camloo produces for every global (exported or not) function two entry points: the first entry is devoted to partial application (applications which build closures), and the other one is called directly when the function is applied to all its curried arguments (total application).

Let us give an example:

```
let rec map f = function
```

```
    [] -> []
  | x :: l -> f x :: map f l;;
map succ [1; 2];;
```

This produces two entry points for map, map and 2-map, and 2-map is called directly without any closure allocation:

```
(define map
  (lambda (x1)
     (lambda (x2) (2-map x1 x2))))

(define 2-map
  (lambda (x1 x2)
      (if (null? x2)
        '()
        (let ((obj2 (2-map x1 (cdr x2)))
              (obj1 (x1 (car x2))))
          (cons obj1 obj2)))))

(2-map succ
        (let ((obj2 (cons 2 '())))
          (cons 1 obj2)))
```

Moreover, this mapping of curried functions to n-ary functions is not restricted to functions local to a module: our compiler fully optimizes inter-modules curried functions. When a module exports a curried function, Camloo generates export clauses for its two entry points (map and 2-map in the preceding example). Then when a module imports a curried function, the Bigloo compiler imports the two entry points, so that it can retrieve the direct entry point when necessary.

Natural mapping of lists

We paid a special attention to map ML lists to Scheme lists. The main reason is the compatibility between pure Scheme code and Scheme code generated from ML. Moreover, "conses" are handled efficiently by Bigloo's runtime: "conses" have a special tag that authorizes to implement them with only two words.

This optimization is very simple: to allocate Caml lists as Scheme lists, it is sufficient to detect applications of the constructors `::@builtin` and `[]@builtin`. For list accesses, we remark that these accesses always occur in the expression part of a clause of some pattern matching, whose pattern part reveals that we are accessing a list. We then replace the generic accesses by proper applications of `car` or `cdr`. (See the `list_length` example above.)

Natural mapping of references

Naive mapping of references as regular Caml values using Bigloo vectors is inefficient: `ref 0` becomes `#(0)`. The problem with this mapping is that each fetch leads to a vector access and each modification of the reference leads to a store into the vector. This is admissible when the reference is used as a first-class value, which is passed to a function or embedded into a data structure. In practice, a very common case is the mapping of references which are bound to a variable and only used as usual imperative variable (for instance to control a loop). In this case, the natural and more efficient way to implement variables bound to reference cells is to use Scheme variables updated via `set!`. In this case, there is no need to allocate space: in most cases the variable simply stays in a register.

Let us take as example a simple while loop, using a reference as control:

```
let x = ref 10 in
while !x > 0 do print_int !x; x := !x + 1 done;;
```

The mapping turns the `while` construct into a recursive function, and the ML variable `x` into an imperative Scheme variable:

```
(let ((x1 10))
  (letrec ((loop (lambda ()
    (if (>fx x1 0)
        (begin
          (print_int@io x1)
          (set! x1 (+fx x1 1))
          (loop))
        '())))))
    (loop)))
```

which is then translated into a C variable:

```
{
   obj x1;
   x1 = 10;

loop:
   if( GT( x1, 0 ) )
   {
      print_int__io( x1 );
      x1 = ADD( x1, 1 );
      goto loop;
   }
   else
      BNIL;
}
```

When the reference is indeed used as a value, we also use the natural Scheme counterpart: a reference cell (a bit more compact than a vector, since it needs only two words).

This optimization is done by a simple syntactic analysis of the $\lambda-$like abstract representation of the Caml program: variables bound to references are mapped to Scheme imperative variables, except if they are used as first-order values (the variable appears without explicit fetch: `x` instead of `!x`).

This program transformation is easy and correct in Camloo, because Caml references which are never used as first-class values have the same semantics as Scheme imperative variables. In particular, the Scheme compiler handles the heap allocation of those imperative variables which are shared between several closures.

5 Libraries

All Caml Light libraries written in Caml were compiled directly by the new compiler. Other libraries were written in C. Among the functions provided by these libraries we distinguish:

- functions already available in Bigloo (arithmetic and basic data type primitives): nothing has to be done here.
- functions of general interest: they have been written in Scheme and added as new components to the Bigloo library. For instance, input and output of circular values (preserving sharing), a few additional functions on basic data structures (`blit_string` and `string_for_read`).
- functions specific to Caml (e.g. input-output functions or camlyacc and camllex engines) were kept in C, with a bit of rewriting (essentially the C macros).

To conclude, the port of Caml Light libraries was easy and in some cases profitable to the Bigloo system.

6 Modifying the two compilers

6.1 Fitting Caml Light

The modifications to the Caml Light compiler have been minor. Apart from the mandatory "dump pass", which is only an abstract tree printer, the sole other modification was added for efficiency. The Caml Light compiler forgets the arity and the name of constructors after pattern matching compilation: these are useless to the rest of the compiler. We had to keep these informations to help the Bigloo compiler to optimize pattern matching and value representation (e.g. lists).

6.2 Fitting Bigloo

Dynamic type checking

From the efficiency point of view, an important problem remains: Scheme is dynamically type checked and performs many runtime type checks. This is not so important in Bigloo since 0cfa analysis and common subexpressions elimination remove many of these tests (up to 75%). Nevertheless we want to take advantage of the static type checking of ML to eliminate all of them. Fortunately, as many Scheme or Lisp compilers, Bigloo has a compilation option to prevent the generation of these tests: thus when compiling Caml code we *safely* use the `-unsafe` option of the Scheme compiler.

Runtime support

Many of the runtime requirements of ML, such as automatic memory management (*GC*) and dynamic allocations of data structures (vectors, closures, strings, ...) already exist in Scheme, we get them for free since we map ML values to Scheme values.

To extend the Bigloo's runtime to implement the Caml Light specific primitives, we intensively used another facility available in Bigloo: the "foreign interface". This facility permits direct calls to C functions (or even calls to C macros) and transparent use of C values by the Scheme code, without stub code.

To summarize, except for the currying optimization, no modifications to the existing Bigloo compiler were needed to get the new compiler. Thus our new compiler has been integrated in the new release of Bigloo: Bigloo version 1.6 compiles both Caml Light and Scheme source files.

7 Related work

Some ML compilers have already been designed by the re-utilization of some existing systems. We compare Bigloo with three currently available systems: Caml V3.1, sml2c and camlot.

Caml V3.1 This Caml compiler [20] has been developed using the Le-Lisp [5] runtime. To obtain the compiler, implementors had to write an entirely new front-end and a code generator. Thanks to the LLM3 virtual machine and Le-Lisp runtime, portability is for free. If the Caml compiled had used the Complice Le-Lisp compiler instead of direct production of LLM3 code, this approach would have been very close to ours (except of course that Bigloo produces C code instead of virtual assembly code).

sml2c This compiler is based on the rewriting of the SML/NJ [1] compiler code generator. The CPS representation of program is compiled into a target machine whose instructions are implemented in C. On the other hand, this compiler shares the same runtime and uses the same high-level optimizations as SML/NJ.

Due to the methodology of sml2c (generation of C code to implement the instructions of a virtual machine) the C code produced is very different from hand-crafted C implementing the same source program. This C code is thus very difficult to optimize by the C compiler. On the contrary, Bigloo works hard to produce readable C code, as similar as possible to hand written C code. The idea is similar to our "natural translation" from ML to Scheme, and has the same good properties: "natural translation" from Scheme to C produces code that the C compiler can optimize very well. [3].

The main interest of the sml2c approach is to minimize the amount of assembly required (on the average, the runtime system uses about 500 lines of assembly for each port). However, we get the same advantage of portability and assembly minimalization with Bigloo (19 lines of assembly code on Sparc architecture, zero lines on others).

camlot Cridlig's approach is close to ours. He modified the Caml Light compiler and wrote a new back-end and runtime system. The C code produced by Camlot is readable: as Bigloo, Camlot performs a natural translation from ML to C.

On the other hand, the modifications to the front-end are large: this includes a new pattern matching algorithm and a new data type representation scheme. This is a hard job, and this explains why Camlot is not yet fully compatible with Caml Light (notably for streams).

These three compilers share the same idea of reusability: some reuse an existing back-end, and others reuse an existing front-end. Thus, the order of magnitude of the implementation effort to obtain the new compiler is about one half the one for a complete compiler.

By contrast, our approach is to reuse a front-end *and* a back-end: we just had to write the "pipe" between those two.

8 Benchmarks

This section presents the obligatory benchmark figures obtained by our compiler, compared with other ML compilers. Time figures present the minimum of three consecutive runs; times are expressed in seconds and concern user+system times as reported by the Unix command `/bin/time`. Thus we measured stand-alone applications (for Sml/NJ, applications are produced with the `exportFn` facility).

The C files produced by Bigloo, Camlot and Sml2c are compiled by `gcc` [18] with `-O2` option.

[3] Let alone the possibility of reasonable usage of C symbolic debuggers

compiler	**version**	**flags**
camlc	0.6	`-O fast`
camlot	0.6	`-O fast -f`
bigloo	v1.6c-0.2	`-unsafe -O3`
sml/nj	1.03f	`-noshare` on Sparc
sml2c		`-ffunction-integration -cf -O2 -gcc -funsafe-arithmetic`

Figure 5: Compiler versions and flags

We measure the execution times on Sun 4 (Sparc 2 architecture, running SunOs 4.1.2, 64 Mo of memory) and on Dec Station (Dec 5000/200, running Ultrix V4.0, 32 Mo of memory).

Figure 5 describes the compilers configuration we used.

Figure 6 shortly describes the benchmark programs we used [4]. Benchmark programs manipulating arrays are safe: all bound tests are explicit in the source code. So, we can safely use the compiler option that omit bound tests when this option is available.

Figure 7 gives for each compiler, the average size of executable files obtained.

compiler	**size on Sparc**	**size on Mips**
camlc	7 k	19 k
camlot	128 k	129 k
bigloo	288 k	232 k
sml/nj	335 k	3867 k
sml2c	387 k	447 k

Figure 7: Size of stand-alone programs

As one can notice, even if Bigloo implements two sets of primitives (one for Scheme, one for ML) its executables produced are reasonably small. Camlc produces extremely compact executables (but those are not stand-alone programs in the Unix sense). Sml/NJ executables are huge on Mips: the 1.03f experimental version of the compiler was not compilable with the `-noshare` option on this architecture.

	Camlc	**Camlot**	**Bigloo**	**Sml/NJ**	**Sml2c**
sort	260.0 s	15.8 s	**9.0** s	27.9 s	47.7 s
life	44.4 s	4.5 s	3.2 s	**2.8** s	5.4 s
takc	30.8 s	**1.4** s	**1.4** s	11.6 s	33.4 s
taku	36.2 s	6.7 s	18.8 s	**4.2** s	8.6 s
boyer	12.6 s	8.9 s	**5.9** s	12.0 s	8.6 s
soli	28.6 s	1.3 s	**1.2** s	4.4 s	6.8 s
kb	73.6 s	80.6 s	46.7 s	**22.1** s	37.9 s
queens	95.0 s	-	**13.3** s	31.6 s	45.3 s
ffib	39.2 s	**2.2** s	**2.2** s	5.2 s	9.4 s
fft	163.0 s	**28.4** s	38.0 s	226.0 s	-

Figure 8: Benchmarks on Sparc architecture

Benchmark results show that Bigloo is very good at compiling function calls, in particular curried functions (see **takc** and **ffib**). On the other hand, the optimization of n-ary *uncurried* functions is really necessary (**taku**). Bigloo and Camlot are slower than the other compilers on **kb**. This is probably due to their *GC* that does not feature generations (very efficient for this benchmark).

Bigloo has good performances for **sort** due to its natural mapping of references. Bigloo compiles loops efficiently, either imperative loops (**sort**, **soli**) or functional ones (**queens**, **life**). Exceptions are efficiently implemented by Bigloo (as well as Camlot) since the **boyer** benchmark feature good performances in spite of its intensive use of exceptions.

Sml/NJ performs the best on Mips architecture, which is not surprising according to A. Diwan, D. Tarditi and E. Moss [7]. By contrast, it features rather poor performances on Sparc architecture (not as good as Sml2c for **life** and **boyer**).

Differences between time figures on Mips and Sparc are much more similar for compilers producing C code. Camlot and Bigloo have close performances (significantly better than Sml2c).

[4]Mail at caml-list@margaux.inria.fr to obtain the source code of these programs.

Conclusion

Our goal was to develop a new fully compatible Caml Light compiler with good performances. Our simple pipelining approach leads to an economical realization (only 2 man-months). The result is a very good Caml compiler which compiles any Caml Light source program: it has already been used to compile the entire Coq proof assistant system [8]. Furthermore, our compiler is a Caml *and* Scheme compiler: it compiles indifferently Scheme and ML source files, and the object codes may live at the same time in the resulting executable program. As far as we know, this degree of integration between two functional languages has never been achieved before.

This seems to open a new perspective in the relationship between Scheme and ML: using our compiler the programmer may freely mix Scheme and ML code, and choose whichever language is more appropriate to develop the algorithm at hand.

Acknowledgement

We would like to thank Christian Queinnec for numerous encouragements to realize a *Scheme and Caml* compiler, and Xavier Leroy and Jean-Marie Geffroy for their careful reading of this paper.

References

[1] A. W. Appel and D. B. MacQueen. *A Standard ML Compiler*, in G. Kahn editor, Functional Programming Languages and Computer Architecture, volume 274, pages 301-324. Springer-Verlag, 1987.

[2] H. G. Baker, *Inlining Semantics for Subroutines which are recursive*, ACM Sigplan Notices 27,12 (Dec 1992), p 39–46.

[3] H. J. Boehm, *Hardware and Operating System Support for Conservative Garbage Collection*, Proc. of the International Workshop on Object-Orientation in Operating Systems, p 61–67, 1991.

[4] E. Chailloux, *An efficient way to compile ML to C*, ACM Sigplan Workshop on ML and its Applications, 1992.

[5] J. Chailloux and M. Devin and F. Dupont and J.-M. Hullot and B. Serpette and J. Vuillemin, *Le-Lisp version 15.24, le manuel de référence*, INRIA, Technical Report, 1991.

[6] R. Cridlig, *An optimizing ML to C compiler*, ACM Sigplan Workshop on ML and its Applications, 1992.

programs	**size (lines)**	**description**	**feature tested**
sort	79	The quicksort algorithm.	Loops, array manipulation, references.
life	148	The life game.	Lists and strings manipulation.
takc	11	The Takeuchi function.	Curried function call.
taku	12	The Takeuchi function.	Uncurried function call.
boyer	1765	Standard theorem-prover benchmark.	Symbolic computation, exception handling.
soli	110	The resolution of the solitaire game.	Loops, array manipulation.
kb	537	The Knuth-Bendix completion algorithm (naive).	Symbolic computation, exception handling.
queens	71	Ten queens problem solved with lists.	Accumulating and mapping on lists.
ffib	13	A Fibonacci-like higher-order function.	Call to closures.
fft	176	A Fast Fourier transform program.	Floating point arithmetics.

Figure 6: Benchmarks description

[7] A. Diwan, D. Tarditi and E. Moss, *Memory Subsystem Performance of Programs Using Copying Garbage Collection*, Symposium on Principles of Programming Languages, 1994.

[8] G. Dowek, A. Felty, H. Herbelin, G. Huet, *The COQ proof assistant user's guide*, INRIA, Technical Report 154, 1993.

[9] IEEE Std 1178-1990, *Ieee Standard for the Scheme Programming Language*, Institute of Electrical and Electronic Engineers, Inc. New York, NY, 1991.

[10] X. Leroy and P. Weis, *Manuel de référence du langage CAML*, InterEditions, Paris, Juillet 1993.

[11] Luis Mateu-Brule, *Stratégies avancées de gestion de blocs de contrôle.*, Thèse d'université, Université Pierre et Marie Curie (Paris 6), Paris, Février 1993.

[12] R. Milner, M. Tofte, R. Harper, *The Definition of Standard ML*, MIT-Press, 1990.

[13] M. Serrano, *Bigloo user's manual*, Technical Report, Inria, to appear.

[14] M. Serrano, *De l'utilisation des analyses de flots de contrôles dans la compilation des langages fonctionnels*, Technical Report, LIX/RR/93/05, 1993.

[15] M. Serrano, *Control flow analysis as an efficient optimization to compile functional languages*, unpublished.

[16] M. Serrano, *Vers une compilation performante des langages fonctionnels*, Thèse d'université, Université Pierre et Marie Curie (Paris 6), Paris, To appear.

[17] O. Shivers, *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*, Carnegie Mellon University, Technical Report, CMU-CS-91-145, 1991.

[18] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc., April 1989.

[19] D. Tarditi, A. Acharya and P. Lee, *No assembly required: Compiling Standard ML to C*, Carnegie Mellon University, Technical Report, CMU-CS-90-187, 1991.

[20] P. Weis et al, *The CAML reference manual*, Technical Report, Inria 121, 1991.

[21] P. Weis and X. Leroy, *Le langage CAML*, InterEditions, Paris, Juillet 1993.

	Camlc	**Camlot**	**Bigloo**	**Sml/NJ**	**Sml2c**
sort	414.8 s	18.0 s	**10.2** s	25.0 s	51.6 s
life	71.1 s	4.2 s	3.7 s	**2.3** s	4.9 s
takc	49.6 s	**2.5** s	**2.5** s	11.3 s	28.8 s
taku	56.7 s	5.9 s	15.2 s	**3.5** s	7.4 s
boyer	17.9 s	**2.4** s	2.7 s	3.6 s	7.1 s
soli	41.8 s	1.5 s	**1.4** s	3.5 s	7.1 s
kb	105.3 s	32.3 s	39.9 s	**10.6** s	30.4 s
queens	143.5 s	-	**8.3** s	13.0 s	42.1 s
ffib	63.8 s	**2.8** s	**2.8** s	4.4 s	8.3 s
fft	187.1 s s	**28.3** s	39.9 s	52.4 s	105.8 s

Figure 9: Benchmarks on Mips architecture