Bigloo: a portable and optimizing compiler for strict functional languages

Manuel Serrano[1,2] and Pierre Weis[1]

[1] INRIA, B.P. 105, Rocquencourt, 78153 Le Chesnay Cedex, FRANCE
[2] Université de Montréal C.P. 6128, succ. centre-ville Montréal Canada H3C 3J7

Abstract. We present Bigloo, a highly portable and optimizing compiler. Bigloo is the first compiler for strict functional languages that can efficiently compile *several languages*: Bigloo is the first compiler for full Scheme *and* full ML, and for these two languages, Bigloo is one of the most efficient compiler now available (Bigloo is available by anonymous ftp on `ftp.inria.fr` [192.93.2.54]).
This high level of performance is achieved by numerous high-level optimizations. Some of those are classical optimizations adapted to higher-order functional languages (e.g. inlining), other optimization schemes are specific to Bigloo (e.g. a new refined closure analysis, an original optimization of imperative variables, and intensive use of higher-order control flow analysis). All these optimizations share the same design guideline: the reduction of heap allocation.

1 Introduction

Strict functional programming languages have many different variations, but they all belong to the same family, the so-called "λ-languages" family. The Bigloo compiler is devoted to the compilation of this class of languages. It was not designed to be the compiler of a specific programming language. It is carefully crafted to be a good compiler for the untyped λ-calculus with n-ary functions, and features many analyzes and optimizations to efficiently deal with functions.

To turn this optimizing functional core language compiler into a compiler for a full language, Bigloo provides room for the addition of preprocessors before the beginning of the compilation process. This is mandatory since Bigloo has almost no hard-wired hypotheses about its high-level source language: for instance Bigloo does not assume the source language to be type checked, neither statically or dynamically. If necessary, and before the compilation by Bigloo, a preprocessing pass must explicitly indicate these runtime type-tests in the source text. This preprocessing mechanism is powerful enough to change the syntax of the source language: it suffices to write a preprocessor that generates λ-terms from whatever the source level syntax could be.

Having to compile a large spectrum of languages, the compiler must also be independent of the set of primitives of the language: Bigloo's library has been designed to be easily modified when changing the source language.

This technique has already been used to turn Bigloo into a compiler for various functional languages: ML [18, 19], Scheme [16], Meroon [10]; a preprocessor for the Dylan language [2] is on the way. The compilers obtained by adding these preprocessors to Bigloo are efficient, and in each case, they compare favorably to the best compilers specialized to the source language.

Bigloo is also highly portable: it virtually exists on every Unix platform. This quality is due to its target language: Bigloo generates C code.

In this paper we present an overview of analyses and optimization used in Bigloo (more detailed descriptions of the algorithms can be found in [12]). In section 2 we explain why we use C as target language, and in section 3 why Bigloo generates "natural" C code. In section 4 we describe Λ^n, the Bigloo source language. In section 5 we present general optimizations. Section 6 is devoted to the presentation of the Scheme and ML front-ends. Before concluding, we present benchmarks in section 7.

2 Portability

The C programming language plays a crucial role in today's computers equipment. With their new computers, most vendors offer an optimizing C compiler which is able to exploit the processor's new capabilities. C is so widely used that it presumably drives the design of new computer architectures. Even for such low-level tasks as system programming, computers are now designed to be programmed in C, and no more directly in assembly code. As practical evidence of this, we can cite the lack of documentation for some assembly languages and the strange behavior of some proprietary C compilers that correct the defects of the hardware by avoiding the generation of instruction sequences that will fall into a processor bug.

Because C runs on so many machines, a C program is highly portable: having portability in mind, one can reduces machine dependent parts to a minimum. Moreover, C compilers produce efficient code. For these reasons, we choose C as the target language of our functional language compiler.

3 Which kind of C code to generate ?

Using C, we get almost for free a portable compiler. Unfortunately this advantage has some drawbacks: as desired, C lets us ignore the machine specific features, but this abstraction may slow down the code produced by our compiler, or complicate the generation of C code. In effect, the C compiler's way to implement some constructs may not fit the semantics of our input languages, or may result in an inefficient implementation of the corresponding input language construct. The solutions to overcome this problems are highly dependent of the kind of C code generated by the compiler. They are many variations, but two main directions emerge:

1. the generation of C code that mimics a virtual machine, in this case the C compiler is considered as a virtual assembler.
2. the generation of C code that resembles handwritten C code.

Each method has its own advantages and its own drawbacks, but we claim that an optimizing compiler must generate "natural" C code (direction 2).

3.1 C as a virtual assembly language

If we use C as an assembly language, we abandon the usage of C control structures. In the C code generated by the compiler there are no C functions, the C stack is hardly ever used, and C variables just serve to maintain the registers of some abstract machine. In this case, the loss of control over the underlying hardware is minimized, since the compiler generates code for an abstract machine which is completely under its control. That way, source language features that need some knowledge about the runtime behavior of programs are easier to compile (e.g. garbage collection or the Scheme `call/cc` function). Unfortunately, these advantages are paid a high price: since the C code generated by the compiler bears no resemblance with the one written by C programmers, it is likely that C compilers will fail to optimize this code properly. The numerous optimizations performed by C compilers will probably not apply and the functional languages compiler will not benefit from the high quality of C compilation.

3.2 "Handwritten-like" C code

The alternate C code generation method is to mimic handwritten C code. In contrast with the preceding approach, the source language features that need to know the runtime behavior of programs are difficult to implement (and even more difficult to implement efficiently). The `call/cc` function is now hard to implement, and the memory management is constrained by the presence of C values in the runtime space of the program (for instance C values stored into locations in the stack): garbage collection must deal with ambiguous roots. On the other hand, we gain a good compilation of our generated C code, resulting in good and homogeneous performances not bound to some particular machine architecture. As an added benefit of this "standard C code" generation, we can run the C programming environment tools on the generated C code: symbolic debuggers, code analyzers (such as `purify`) or profilers.

In this method of C code generation, every source language construct is compiled in an equivalent C construct. More precisely, if an equivalent C construct exists, then the source language construct is compiled in that C construct. For instance, source language functions are compiled into C functions (or even in C loops when the source language functions are tail-call loops), source language variables are compiled into C variables, and so on. This mapping is an instance of what we call the "natural projection" from one language to the other. Bigloo uses this technology and projects λ-terms to handwritten-like C code. To manage C values in the stack, Bigloo uses the Boehm garbage collector [4].

4 The Λ^{n} language

The input language of Bigloo is a non-standard version of the λ-calculus: in this Λ^{n}-calculus the λ-abstractions are not restricted to abstract one variable at a time. Similarly the application is not binary, as usual in λ-calculus, but n-ary. This n-ary λ-calculus clearly contains regular λ-terms. For instance the λ-term $\lambda x.\lambda y.x$ can be encoded in Λ^{n} as a term T_1 that abstracts one variable twice: $T_1 = \lambda^1 x.\lambda^1 y.x$. Alternatively, it can be encoded as a single abstraction of two variables: $T_2 = \lambda^2 xy.x$. Terms T_1 and T_2 have different meaning and must not be confused: when applying Λ^{n} terms there is a static arity consistency check that forbids inconsistent partial application; thus T_2 cannot be applied to only one argument, while T_1 cannot be applied to two arguments: T_1 must be applied twice to only one argument at a time. For instance we can write $\texttt{@}^1(\texttt{@}^1(T_1, e_1), e_2)$ reflecting the fact that T_1 is partially applied to e_1, leading to a function result, which is then applied to e_2. On the contrary $\texttt{@}^2(T_2, e_1, e_2)$ expresses the fact that T_2 is directly applied to two arguments and does not require any partial application.

Λ^{n} calculus is call-by-value, and its semantics can be expressed using the classical weak β-reduction (with multiple substitutions in parallel to deal with n-ary abstractions and applications).

To get the complete input language of the compiler, the basic Λ^{n} calculus is extended with several other constructs: `let` bindings, conditionals, `case` constructs, and constants. Note that `let` bindings introduce variables which can be assigned to (to deal with the imperative features of the source language). Note also that definitions introduced by `let` can be recursive. For the sake of simplicity we only consider integer constants.

Λ^{n}::=

$\mid i$	*integer constants*
$\mid id$	*variable*
$\mid \lambda^m\ id_1\ \ldots\ id_m\,.\,\Lambda^{\text{n}}$	*n-ary function*
$\mid \texttt{@}^m\,(\Lambda^{\text{n}}{}_1, \ldots, \Lambda^{\text{n}}{}_m)$	*n-ary application*
$\mid$ `let` $\{id$ `=` $\Lambda^{\text{n}}\}^+$ `in` Λ^{n}	*let binding*
$\mid$ `if` Λ^{n} `then` Λ^{n} `else` Λ^{n}	*conditional*
$\mid$ `case` $\{$`[` i `:` Λ^{n}`]`$\}^+$ `[[else :` Λ^{n}`]]`	*integer switch*

5 Optimizing Λ^{n} code

The compilation scheme we use, (called "natural projection" to C), requires two kinds of high level optimizations: *(i)* optimizations to improve the natural projection (careful selection of target C constructs) *(ii)* source to source transformations on Λ^{n} code to perform optimizations that the C compiler cannot do because these optimizations require semantics knowledge about the high level source language (e.g. source language module informations).

5.1 Optimizing the "natural projection"

The main design effort for Bigloo has been to compile functions as well as possible: in our mind this implies the mapping of Λ^{n} functions to C functions (or even to C loops) and to hard work to avoid heap-allocation of closures.

Closure analysis We name *closure analysis* the methodology and algorithms used in Bigloo to compile functions efficiently. It is described in the paper [13]. It involves an abstract interpretation which is based on Shiver's 0cfa analysis [15], and uses an algorithm described by Séniak in [11]. We do not present the closure analysis again, but give its results for some examples to give an idea of the way Bigloo compiles functions.

For simplicity, examples are not given using the compiler source language (Λ^{n} code) but in high level source languages (Scheme or ML).

Compiling functions to C loops Closure analysis aims at compiling Λ^{n} functions into C loops. This scheme applies when Λ^{n} functions do not escape (that is, when functions are not used as first-class values) and when these functions are always tail-recursively invoked. Here is an example of two mutually recursive functions that map to C loops:

```
(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 as:

```
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 When the previous scheme does not apply, non escaping n-ary functions map to C n-ary functions.

Consider the ML `map_succ` definition:

```
let map_succ l =
  let rec map = function [] -> []
                       | x :: l -> 1 + x :: map l in
  map l;;
```

It is compiled as:

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

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

Heap-allocated closures When functions escape, these efficient mappings do not apply: the compiler must allocate heap space for function environments. Bigloo uses flat closures: the environment part is a heap block containing exactly the free variables of the function. 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 user.

Thus, only escaping functions are allocated in the heap. This is the case for the λ-expression returned as value by the functional composition of two functions:

```
1:  let o f g =
2:       function x -> f (g x)
```

Hence Bigloo creates a heap-allocated closure for the expression of line *2*:

```
obj o( obj f, obj g )
{
   obj clo;
   clo = make_closure( lambda_1, f, g );
   return clo;
}

obj lambda_1( obj clo, obj x )
{
   obj f, g;
   f = PROCEDURE_REF( clo, 0 );
   g = PROCEDURE_REF( clo, 1 );
   return PROCEDURE_ENTRY( f )( f, PROCEDURE_ENTRY( g )( g, x ) );
}
```

As mentioned above, Bigloo's closures are arrays. Each closure contains the free variables of the function, pointers to C functions, and an integer to record the function arity. This arity slot is mandatory for dynamically typed languages to ensure soundness of functional application. Amazingly enough it is not useless for statically typed languages: it is used to optimize n-ary computed calls (applications where the functions called are not known at compile-time). Thanks to the arity slot, partial application can be checked dynamically, and if the computed call is total, then the "uncurried" entry point of the function is called (see section 5.1). This prevents closure allocations for n-ary computed calls.

Even functions used as first class values can be compiled without heap-allocation using the 0cfa analysis. In the paper [13] we have shown that, in general, with our various optimizations, less than 20 % of functions require to be heap allocated.

The $\mathcal{C}$-transformation $\Lambda^{\rm n}$ terms faithfully reflect functions arity and closures creation. In particular, the formalism expresses that n-ary functions can be applied to n arguments without any intermediate closure creation. The $\mathcal{C}$-transfor mation (uncurrying) attempts to turn a spine of unary abstractions into a single n-ary abstraction. Conversely, spines of unary applications are replaced by single n-ary applications. In any case, this transformation speeds up function applications and reduces heap allocation. For some source languages which do not feature n-ary functions, this optimization is essential. For instance, applied to Caml, the $\mathcal{C}$-transformation speeds up compiled programs by a factor of two. In effect, in Caml, n-ary functions are encoded by the programmer as n-level nested unary functions. We use as benchmark the bootstrap of the Caml compiler (12.000 lines of code).

$$
\mathcal{C}\left[\!\!\left[\begin{array}{l}
\texttt{let} \ \ldots \\
\quad \boxed{\begin{array}{l} f \ \texttt{=} \ \lambda^1 \ x_1 . \\ \qquad\qquad \ddots \\ \qquad\qquad \lambda^1 \ x_n . \ \Lambda^{\mathrm{n}} \end{array}} \\
\ddots \\
\boxed{\texttt{@}^1 (\ldots \ (\texttt{@}^1 (f, \ a_1), \ \ldots), \ a_n)} \\
\qquad \ddots
\end{array}\right]\!\!\right]
=
\left[\!\!\left[\begin{array}{l}
\texttt{let} \ \ldots \\
\quad \boxed{\begin{array}{l} f_1 \ \texttt{=} \ \lambda^1 \ x_1 . \\ \qquad\quad \ddots \\ \qquad\quad \lambda^1 \ x_n . \ \texttt{@}^n (f_2, \ x_1, \ \ldots) \\ f_2 \ \texttt{=} \ \lambda^n \ x_1 \ \ldots \ x_n . \ \Lambda^{\mathrm{n}} \end{array}} \\
\ddots \\
\boxed{\texttt{@}^n (f_2, \ a_1, \ \ldots, \ a_n)} \\
\qquad \ddots
\end{array}\right]\!\!\right]
$$

Fig. 1. The $\mathcal{C}$-transformation

The $\mathcal{C}$-transformation (figure 1) produces for every curried function f two function definitions, f_1 and f_2. The function f_1 is used for partial applications of f whereas f_2 serves for total applications. The function f_2 is n-ary, hence total applications of f become n-ary applications.

Functions f_1 and f_2 are related to f by a naming convention so that the $\mathcal{C}$-transformation also applies globally and even across module barriers. Moreover f_1 and f_2 are dynamically linked, so that n-ary computed calls to f can benefit from the transformation.

The $\mathcal{C}$-transformation can be illustrated by the following Scheme code program transformation:

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

((add 0) 1)
```

```
(define add1
   (lambda (x)
      (lambda (y)
         (add2 x y))))
(define add2
   (lambda (x y) (+ x y)))

(add2 0 1)
```

η-transformation The η-transformation optimization belongs to the same class as the $\mathcal{C}$-transformation: it avoids intermediate closures creation and partial evaluations. The η-transformation (figure 2) encapsulates *function definitions* into extra functional abstractions.

The following source to source rewriting illustrates the transformation:

```
let do_list f =
  let rec loop = function
      [] -> ()
    | a::l -> f a; loop l
  in loop;;

do_list print_int [1; 2];;
```

```
let do_list f new =
  let rec loop = function
      [] -> ()
    | a::l -> f a; loop l
  in loop new;;

do_list print_int [1; 2];;
```

The functional `do_list` is in fact binary but the original program hides this fact, since the partial evaluation of `do_list` to a function explicitly returns a closure. In the rewritten program, the definition of `loop` is abstracted into an extra λ-abstraction `function new -> let rec loop ...`. Hence `do_list` becomes binary and the $\mathcal{C}$-transformation now applies.

This transformation has not to be confused with the η-rule of λ-calculus: in the presence of side-effects the η-rule of λ-calculus is not valid in general for arbitrary expressions. Fortunately, our transformation uses the η-rule for syntactic *functions*, for which the η-rule is always valid.

$$\eta\left[\!\left[\begin{array}{l}\boxed{\begin{array}{l}\texttt{let } f\ \texttt{=}\\ \quad\lambda^1\ x_1\texttt{.}\\ \qquad\ddots\\ \qquad\quad\lambda^1\ x_n\texttt{.}\ \Lambda^{\mathrm{n}}\end{array}}\\ \quad\texttt{in}\\ \quad\boxed{\texttt{@}^1\texttt{(... (@}^1\texttt{(}f\texttt{, }a_1\texttt{), ...), }a_m\texttt{)}}\end{array}\right]\!\right] = \left[\!\left[\begin{array}{l}\lambda^1\ \texttt{new}_n\texttt{.}\\ \quad\ddots\\ \quad\lambda^1\ \texttt{new}_{m+1}\texttt{.}\\ \qquad\boxed{\begin{array}{l}\texttt{let } f\ \texttt{=}\\ \quad\lambda^1\ x_1\texttt{.}\\ \qquad\ddots\\ \qquad\quad\lambda^1\ x_n\texttt{.}\ \Lambda^{\mathrm{n}}\end{array}}\\ \quad\texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{@}^1\texttt{(...}\\ \ \texttt{(@}^1\texttt{(... (@}^1\texttt{(}f\texttt{, }a_1\texttt{), ...),}\\ \quad a_m\texttt{), ...), new}_n\texttt{)}\end{array}}\end{array}\right]\!\right]$$

Fig. 2. The η transformation ($n > m$).

The relevance of this optimization depends on the programming style. In practice we have noticed that Caml programs are sensitive to η, especially library routines (the `do_list` example is extracted from the Caml standard library). When η applies, the impact is impressive: thanks to η, the **kb** program (section 7.2) and the Coq system [7] run two times faster (the Coq system is 20 000 lines of Caml code long, and the benchmark runs during more than 30 minutes on a dec alpha station, allocating 12 gigabytes).

The $\mathcal{I}$-transformation The $\mathcal{I}$-transformation substitutes cascades of nested tests by n-way branches. This is just rewriting some nested conditionals in the Λ^{n} code into a `case` construct (figure 3).

$$\mathcal{I}_{trans}\left[\!\!\left[\begin{array}{l}\texttt{if } \Lambda^{\mathrm{n}}{}_c = i_0 \\ \quad \texttt{then } \Lambda^{\mathrm{n}}{}_0 \\ \quad \texttt{else} \\ \quad \ddots \\ \qquad \texttt{if } \Lambda^{\mathrm{n}}{}_c = i_m \\ \qquad\quad \texttt{then } \Lambda^{\mathrm{n}}{}_m \\ \qquad\quad \texttt{else } \Lambda^{\mathrm{n}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{case } \Lambda^{\mathrm{n}}{}_c \\ \quad \texttt{[}i_0 \texttt{ : } \Lambda^{\mathrm{n}}{}_0\texttt{]} \\ \quad \quad \ldots \\ \quad \texttt{[}i_m \texttt{ : } \Lambda^{\mathrm{n}}{}_m\texttt{]} \\ \quad \texttt{[else : } \Lambda^{\mathrm{n}}\texttt{]}\end{array}\right]\!\!\right]$$

Fig. 3. The $\mathcal{I}$-transformation

The scope of $\mathcal{I}$ is limited to cascades of tests that verify:

– Every test is an expression $\Lambda^{\mathrm{n}}{}_c = i$, for some integer i and some fixed expression $\Lambda^{\mathrm{n}}{}_c$.
– The expression $\Lambda^{\mathrm{n}}{}_c$ is side effect free (since after $\mathcal{I}$ rewriting, $\Lambda^{\mathrm{n}}{}_c$ is evaluated only once).

The $\mathcal{I}$-transformation simplifies pattern matching expansion: specific pattern matching compilers (those for ML or extended Scheme) have not to take care of this optimization, which is performed by Bigloo when compiling Λ^{n} expressions.

5.2 Optimizing Λ^{n} source code

Inlining Open-coding (or inlining) suppresses some function calls, replacing these calls by the body of the function. Functions are so widely used and they are so many function calls in functional programs that inlining has a large impact on efficiency. This impact is mostly due to the functional programmer habit to define many small functions to improve the readability of their code. With a efficient inlining strategy, this good style of programming does not compromise efficiency.

Inlining improves compiled code in several aspects: *(i)* the function invocation cost disappears (that includes the cost of parameter passing instructions, context switch, register moves, and the jump to the function code that breaks the control flow) *(ii)* local optimizations of the compiler are more relevant since they apply to larger code blocks *(iii)* recursive functions inlining reduces dynamic closure allocations.

One may think that Bigloo could delegate inlining to its C compiler back-end. C compilers may have some inlining strategy, but this strategy is never aggressive enough to efficiently compile functional programs. In addition, the inlining pass of Bigloo extends the scope of other optimizations (for instance the data-flow optimizations).

Which functions to inline ? Inlining is very efficient but has to be "handled with care": inconsiderate inlining may imply an unbearable growth of the compiled code size. To prevent code explosion, Bigloo inlines only functions that verifies some pragmatic properties:

– either the function is called only once in the entire program (in this case there is no code expansion);
– or the body of the function is small enough: the size of its Λ^{n} abstract syntax tree is less than a value $\mathcal{S}$; $\mathcal{S}$ depends on the numbers of parameters of the function and on the optimization level of the compiler.

In addition, inlining cannot occur in some contexts:

– No inlining of f can occur when inlining the body of f. This prevents infinite inlining even in case of mutually recursive functions.
– Nested inlining depth is limited by the compiler since unlimited nested inlining may produce code explosion (when inlining a small function that calls another small function that calls ..., the previous criteria do not apply, even if we do want to stop inlining since the code could be growing too much).

The inlining process As mentioned above, Bigloo prevents recursive inlining of the same function, but it still accepts to inline *recursive definitions.* Bigloo features an original and efficient scheme to perform inlining of recursive functions.

In case of inlining of non recursive functions the $\mathcal{L}_{let}$ transformation (figure 5.2) applies. Inlining of recursive functions is a bit more delicate. Rather than unrolling recursive calls to a certain depth, Bigloo creates a local recursive definition for the open-coded function. When inlining recursive function f, $\mathcal{L}_{rec}$ (figure 5.2) creates the local definition and then replaces calls to f by calls to this local function.

An inlining example To illustrate the optimization, let us define `map_succ` using the `map` functional:

```
let rec map f = function [] -> []
                       | x :: l -> f x :: map f l;;
let succ x = x + 1;;
let map_succ l = map succ l;;
```

$$\mathcal{L}_{let}\left[\!\!\left[\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ x_m.\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\texttt{@}^m\texttt{(}f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_m\texttt{)}}\\ \qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ x_m.\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{let}\ x_1\ \texttt{=}\ a_1\\ \qquad\vdots\\ \quad\texttt{in}\\ \quad\Lambda^{\mathrm{n}}\end{array}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

$$\mathcal{L}_{rec}\left[\!\!\left[\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ \Lambda^{\mathrm{n}}\\ \ddots\\ \quad\boxed{\texttt{@}^m\texttt{(}f\texttt{,}\ a_1\texttt{,}\ \ldots\ \texttt{)}}\\ \qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{let}\ f\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ \Lambda^{\mathrm{n}}\\ \ddots\\ \boxed{\begin{array}{l}\texttt{let}\ f'\ \texttt{=}\ \lambda^m\ x_1\ \ldots\ .\ {\Lambda^{\mathrm{n}}}_{[f/f']}\\ \quad\texttt{in}\\ \quad\texttt{@}^m\texttt{(}f'\texttt{,}\ a_1\texttt{,}\ \ldots\ \texttt{)}\end{array}}\\ \qquad\qquad\ddots\end{array}\right]\!\!\right]$$

Fig. 4. inlining

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

```
  let map_succ l =
    let rec map f = function [] -> []
                           | x :: l -> f x :: map f l in
    map succ l;
```

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

```
  let map_succ l =
    let rec map = function [] -> []
                         | x :: l -> succ x :: map l in
    map l;;
```

Then, `succ` is open-coded, and we finally get a very efficient equivalent piece of code:

```
  let map_succ l =
    let rec map = function [] -> []
                         | x :: l -> x + 1 :: map l in
    map l;;
```

Impact of Inlining In [1] A. Appel demonstrates that inlining is the most efficient optimization of the Sml/NJ compiler. In order to measure the impact of inlining, we applied it to several Scheme and ML programs (including the full bootstrap of the Bigloo compiler which is 30.000 lines of Scheme code long). The effect of inlining seems to be architecture dependent: we obtained a speedup of 20 % on Sparc and 30 % on Mips. Our inlining decision algorithm is accurate, since on both architectures code growth is limited to 5 %.

6 Specific front-ends

To obtain a compiler for a full language we must add two modules to the core Λ^{n} compiler: a runtime library and a front-end translator from the source language to Λ^{n}.

To define a new set of primitives for a new language, the Λ^{n} compiler provides a powerful interface with its target language: the "extern" interface. This tool gives the programmer full access to the C world: C functions, C macros, and C data structures are transparently available.

A specific front-end has to deal with particular features of its source language and to provide a careful natural mapping to Λ^{n}. In the following we do not detail the whole Scheme and Caml front-ends: we just present some fine points of each front-end.

6.1 The Scheme front-end

Dynamic type tests The mapping from Scheme to Λ^{n} is easy: the front-end is mainly in charge to make dynamic type tests explicit. Data-flow analysis optimization of the Λ^{n} compiler will automatically remove many of them.

call/cc The Λ^{n} compiler ignores the `call/cc` function which has the same status as any other primitive. This function has been implemented in C, in the Scheme specific library.

6.2 The Caml front-end

The Caml front-end is in charge of type reconstruction and pattern matching expansion. In addition, the front-end optimizes references. This front-end is described in details in the paper [14].

Optimizing References A naive implementation of ML references would be to use Λ^{n} arrays. A more efficient scheme is to map references to Λ^{n} variables. This is admissible when the reference is not used as a first-class value (that is passed to a function or stored into data structures). The Caml front-end includes an analysis devoted to this mapping. Let us take as example a simple while loop, using a reference to control the loop:

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

The Caml front-end maps the variable `x` into an imperative Λ^{n} variable; then, Bigloo maps the `while` construct into a C loop, and the `x` variable to a C variable:

```
{
   obj x;
   x = 10;
loop:
   if( GT( x, 0 ) )
   { print_int__io( x ); x = ADD( x, 1 ); goto loop; }
   else BNIL;
}
```

This clever mapping of references is highly optimizing for imperative ML programs: the **sort** program of section 7.2 runs 50% faster.

The `try/raise` *construct* As for the Scheme `call/cc` function, these constructs are implemented as library functions.

7 Benchmarks

This section presents the obligatory benchmark figures obtained by our compiler, compared with other ML and Lisp compilers. Time figures present the minimum of three consecutive runs; times are expressed in seconds and represent user+system times. For compilers which produce C code (Bigloo, Camlot [6], scheme-to-c [3], sml2c [17]), C files are compiled with `gcc` using the -O2 option.

Bigloo is available on many Unix platform (Sparc, HP-PA, Mips, Alpha, Intel, Next, RS6000, MC68k, ...). We measure the execution times on a Sun 4 (Sparc 2 architecture, running SunOs 4.1.2, 64 Mo of memory): we report the best user+system time of 3 consecutive runs.

For all programs, compilers were used with the maximum optimization levels and suppression of range checking when available.

7.1 Lisp benchmarking

We compare three Scheme compilers (Bigloo 1.7, S2c 15mar93, and Orbit t 3.1), the LeLisp [5] 15.24 compiler (complice), and the Cmu-cl Common lisp compiler Python 1.0 (17e)) [9].

We use the Gabriel benchmarks suite: this is not completely satisfactory for Scheme since the benchmark do not feature higher order functions. However, they allow the comparison of Scheme and Lisp compilers.

Bigloo regularly obtains the best results on these benchmarks (figure 5). **Div2-rec** on the Sparc architecture is a noticeable counterexample: this is due

	Compiler				
Benchmark	**Bigloo**	**S2c**	**Orbit**	**Complice**	**Cmucl**
Dderiv	*0.57*	0.80		1.23	1.80
Deriv	*0.44*	0.57	0.82	1.00	0.59
Destru	*0.15*	0.27	0.30	0.38	0.32
Div2-iter	*0.19*	0.27	0.30	0.50	0.48
Div2-rec	0.97	1.10	0.56	*0.48*	0.66
Puzzle	*0.54*	0.55	0.59	1.46	9.56
Tak	*0.02*	0.03	0.06	0.08	0.05
Fread	*0.02*	0.96	0.10	0.04	0.06
Boyer	3.22	4.70	5.64	*2.53*	4.40

Fig. 5. Gabriel's benchmarks on Sparc architecture

to Sparc's register windows. Compilers having C as target language use this feature of the Sparc processor and consistently obtain poor results. Register windows' usage is expensive in case of deep recursion (as for **Div2-rec**).

7.2 ML benchmarking

To test ML compilers, the situation is not so easy: there is no well established ML benchmark suite. Moreover our ML compilers do not share the same input syntax (Sml or Caml), nor the same libraries. In consequence there is a non trivial rewriting to perform when porting a program from a compiler to another. This presents the use of very large benchmarks such as compiler or theorem provers (for instance, Coq only runs with Bigloo and Camlc). We thus choose small but accurate benchmarks: our programs are specially designed to test specific features of compiler (see for instance **taku** and **takc** to test the compilation of n-ary functions).

We used various programs for a total amount of 2500 lines of code. We test loops (**sort**, **soli**), array manipulations (**sort**), references (**sort**, **soli**), lists (**queens**, **life**), strings (**life**), function calls (**fib**, **takc**, **taku**, **ffib**) and exceptions (**boyer**, **kb**). 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. The programs **takc** and **taku** are two equivalent versions of the Takeuchi function, that only defer by the encoding of n-ary functions: **takc** encodes them with currification and **taku** with tuples defined as:

```
let rec tak (x, y, z) =
  if x > y then tak(tak (x-1, y, z), tak (y-1, z, x), tak (z-1, x, y))
          else z;;
```

Benchmarks concern the following ML compilers: Bigloo 1.7, Camlot (0.64), Camlc (0.6), sml/nj (1.03f) and sml2c (based on sml/nj 0.75).

Benchmark results show (figure 6) that Bigloo is very good at compiling function calls, in particular curried functions (see **takc** and **ffib**). On the other

	Compiler				
Benchmark	**Bigloo**	**Camlc**	**Camlot**	**Sml/NJ**	**Sml2c**
sort	*9.0* s	260.0 s	15.8 s	27.9 s	47.7 s
life	3.2 s	44.4 s	4.5 s	*2.8* s	5.4 s
takc	*1.4* s	30.8 s	*1.4* s	11.6 s	33.4 s
taku	7.1 s	36.2 s	6.7 s	*4.2* s	8.6 s
boyer	*4.9* s	12.6 s	8.9 s	12.0 s	8.6 s
soli	*1.2* s	28.6 s	1.3 s	4.4 s	6.8 s
kb	27.6 s	73.6 s	80.6 s	*22.1* s	37.9 s
queens	*13.3* s	95.0 s	14.6 s	31.6 s	45.3 s
ffib	*2.2* s	39.2 s	*2.2* s	5.2 s	9.4 s
fft	*22.8* s	163.0 s	28.4 s	226.0 s	-

Fig. 6. ML benchmarking on Sparc architecture

hand, the optimization of n-ary *uncurried* functions is badly missing (compare **takc** and **taku**).

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.

8 Future work

The time figures obtained for benchmarks demonstrate that a clever compilation of functions and function calls is crucial. For instance the `taku` and `takc` lines in figures 6 reveal the internal strategy used by the compilers to optimize n-ary functions: Sml/NJ and sml2c perform better on **taku** than **takc**, which reveals the optimization of n-ary functions encoded as unary functions with tuple arguments; on the other hand Camlc, Bigloo, and Camlot optimize n-ary functions encoded as curried unary functions, hence the better results for **takc** compared to **taku**. In any case, it appears that minimization of heap allocation for control flow is a corner stone of efficient functional language compilation. More generally, we claim that an optimizing compiler must work hard to minimize *any* heap allocation, either for control or for data. Bigloo already has an optimizing scheme to turn data heap allocations to stack allocation, which is very promising, since for some programs it improves runtime figures by up to a factor of 3. This allocation optimization scheme has to be studied and generalized, in particular to use “flat” allocations in the spirit of X. Leroy’s “wrap/unwrap” [8].

Conclusion

Benchmark figures for Scheme and ML show that the sharing of the Λ^n intermediate language leads to an easy and efficient compilation scheme for different languages. The generation of "Handwritten-like" C code ensures good performances, whatever the machine architecture could be. The Bigloo compiler demonstrates that the key stone to get high performances is to avoid heap allocations, in particular for control and for functions representation. Multiple high level analyses and optimizations combined with the low level optimizations of the C compiler make Bigloo one of the best existing compilers for strict functional programming languages.

References

1. A. Appel. *Compiling with continuations.* Cambridge University Press, 1992.
2. Eastern Research Apple Computer and Technology. *Dylan, An object-oriented dynamic language.* Apple Computer, Inc., April 1992.
3. J.F. Bartlett. Scheme->C a Portable Scheme-to-C Compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.
4. H.J. Boehm. Space efficient conservative garbage collection. In *ACM Conference on Programming Language Design and Implementation, SIGPLAN Notices 28, 6*, pages 197–206, 1991.
5. J. Chailloux, M. Devin, F. Dupont, J.M. Hullot, B. Serpette, and J. Vuillemin. Le_lisp version 15.2. le manuel de référence. Technical Report L-003, INRIA-Rocquencourt, France, 1986.
6. R. Cridlig. An optimizing ML to C compiler. In *Workshop on ML and its applications*, San Francisco, California, USA, 1992. ACM SIGPLAN.
7. G. Dowek, A. Felty, H. Herbelin, and G. Huet. The COQ proof assistant user's guide. Technical Report 154, Inria-Rocquencourt, France, 1993.
8. X. Leroy. Unboxed objects and polymorphic typing. In *Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, January 1992.
9. R.A. MacLachlan. CMU Common Lisp User's Manual. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213, USA, May 1992.
10. C. Queinnec. Designing MEROON v3. In Christian Rathke, Jürgen Kopp, Hubertus Hohl, and Harry Bretthauer, editors, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop*, Sankt Augustin (Germany), September 1993.
11. N. Séniak. *Théorie et pratique de Sqil: un langage intermédiaire pour la compilation des langages fonctionnels.* PhD thesis, Université Pierre et Marie Curie (Paris VI), November 1991.
12. M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels.* PhD thesis, Université Pierre et Marie Curie (Paris VI), December 1994.
13. M. Serrano. Control Flow Analysis: a Functional Languages Compilation Paradigm. In *Symposium on Applied Computing (SAC '95)*, Nashville, Tennessee, USA, February 1995.

14. M. Serrano and P. Weis. 1 + 1 = 1: an optimizing Caml compiler. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 101–111, Orlando (Florida, USA), 1994. INRIA RR 2265.
15. O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.
16. IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
17. D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Carnegie Mellon University, Pittsburg, Pennsylvania, School of Computer Science, March 1991.
18. P. Weis. The CAML Reference manual, Version 2.6.1. Technical Report 121, INRIA-Rocquencourt, 1990.
19. P. Weis and X. Leroy. *Le langage CAML*. InterEditions, Paris, 1993.