A Fresh Look to Inlining Decision

Manuel Serrano

Université de Montréal
C.P. 6128, succ. centre-ville, Montréal (Québec), Canada H3C 3J7
Serrano@IRO.UMontreal.CA

Invited paper to ICS 95, Mexico City

Abstract

Included in a compiler for functional or object oriented languages, inline function expansion has been reported as one of the most valuable optimizations. Unfortunately, it has an important counterpart: since it duplicates function body, it enlarges the code of the compiled programs as well as the resulting object code. The main contribution of this paper is to present a simple compile time inlining decision algorithm where the code length increasing factor is a constant that can be tuned by the compiler designer and where execution improvements are comparable with other previous sophisticated technics.

Our major concern is about functional languages. With these languages, recursive functions are widely used: the second contribution of this paper is the presentation of an original *ad hoc* inlining framework for recursive functions which is more accurate than function unfolding.

Experimental results demonstrate that our inlining technics succeed in producing small *and* efficient compiled object codes.

1 Introduction

Inline function expansion (shortly named "inlining" in this paper) replaces a function invocation with a modified copy of the function body. Many works studying compilation of functional or object oriented languages mentionned that inlining is one of the most valuable optimizations [2, 5, 11].

Inlining can reduce execution times of compiled programs because it removes calling sequences and it increases the impact of further optimizations:

- Function call sequences are expensive because they require many operations (context save and context restore (i.e. memory fetches) and jumps). In the worst case, for small functions, calling sequence could even be more expensive than body evaluation.
- Inlining can enlarge the impact of the other compiler optimizations because inlined function bodies are modified copies. The formal function parameters are replaced (or bound to) the actual parameters. Known properties about the actual parameters can then be used to optimize the duplicated function body. For instance, inlining helps the compilation of polymorphic functions because modified versions could become monomorphic.

  Inlining helps the optimization of the called function but it also helps the compilation of the body of the calling function. Informations about actual parameters are used to improve the compilation of an inlined function body; informations about an inlined function result are used to improve the calling function.

As inlining duplicates function bodies, it has an obvious drawback: it increases the size of the program to be compiled and possibly the size of the produced object file. In this case, compilation becomes more time consuming (because after inlining the compiler must compile larger abstract syntax trees) and, in the worst case, executions can become slower. On modern architectures, fitting into memory caches is mandatory. This holds for both data and code segments. The smaller the executable code is, the greater will be the chances that it will fit into the code cache.

To prevent code explosion, inlining expansion cannot be applied to all function calls. The most difficult part of the inlining optimization is the design of a realistic decision inlining algorithm. This paper focuses on this problem. It presents a linear compile time algorithm which allows a fine control of the code growing. Code length increasing factor is a constant of our algorithm and can be tuned by the compiler designer. The presented algorithm does not use profiling informations nor user annotations.

Functional languages paradigm implies a large use of functions. For instance, the most popular book about the Scheme programming language [1] uses functional abstractions in order to implement abstract data types. Functions are widely used and particularly recursive ones. All functions (i.e. recursive and non-recursive ones) can be inlined with a general framework but a refined expansion can be achieved for recursive functions. This paper presents an original and efficient framework to inline recursive functions.

This paper is organized as follows: section 2 contains a refutal of previous inlining decision algorithms. Section 3 presents our algorithm. Section 4 presents the recursive functions inlining scheme. Section 5 reports on the impact of the inlining expansion optimization when included in a compiler for functional languages.

2 Refutation

This section contains a refutal of the main inlining decision criteria and algorithms previously published. We classify the previous works into four parts. The remarks made in this section have driven the design of the algorithm presented in section 3.

2.1 User inlining indications

Some systems (e.g. gcc [13]) or language definitions (e.g. C++ [14] or Modula-3 [3]) allow programs to contain inlining annotations.

Rule 1 (inlining annotation) *Let d the definition of a function f; a call to f is inlined if d contains inline annotation.*

Inlining annotations are useful to implement some part of the source language (for instance, in Bigloo [10], our *Scheme & ML* compiler, inlining annotations are intensively used in the libraries for accessor and mutator primitives like `car`, `cdr`, `vector-ref` or `vector-set!`) but cannot replace automatic inlining decision. An automatic inlining decision algorithm can choose to inline a function on a call site, but also to not do it on another call site. User inlining annotations are attached to function definitions and, as a consequence, are not call site dependent. Misplaced inlining annotations can lead to code size explosion. Furthermore, inlining annotations require from the programmer a fair understanding of the compiler strategy in order to be able to judiciously place annotations.

2.2 Abstract syntax tree size counting based decision

Simpler inlining decision algorithms [13, 5] are based on the body size of called functions:

Rule 2 (body-size) *Let $\mathcal{K}$ a constant threshold, f a function, s the body size of f; a call to f is inlined if s is lesser than $\mathcal{K}$.*

The way s is computed is not crucial. A common method is to sum the number of nodes of the abstract syntax tree of f.

Rule 2 succeeds in limiting the code size growth without recursive function definitions because any function call of a program cannot be replaced by an expression bigger than $\mathcal{K}$. Hence, inlining expansion of a program containing a c call can increase the code size of $c \times \mathcal{K}$.

In order to prevent infinite loops when inlining recursive functions, rule 2 has to be extended:

Rule 3 (body-size & nested-call) *Let f a function, k a call to f; k is inlined if f satisfies rule 2 and if k is not included in the body of f.*

This rule is efficient to prevent code explosion and fairly easy to implement but it is very restrictive. It may prevent inlining of functions which could have been inlined without code growth. For example, let's study the following Scheme definition:

```
(define (plus x1 x2 x3 ... xn)
   (+ x1 x2 x3 ... xn))
```

Let n be larger than the constant threshold $\mathcal{K}$. Therefore, such `plus` function cannot be inlined because it is said to be too large (its body size is greater than $\mathcal{K}$). However, inlining calls to `plus` does not increase the compiled code size because the body size of `plus` is not greater than any call to `plus`. From this remark, Appel presents in [2] an improvement of rule 2:

Rule 4 (body-size vs call-size) *If the body of a function f is smaller than the overhead required for a function call, then f may be inlined without increasing the program size.*

2.3 Optimization estimations based decision

The inlining of a call to a function f replaces the formal parameters by the actual parameters and some of them may have special properties (for instance, some may be constants). Other compiler optimizations (e.g. *constant folding*) will be able to shrink the modified version of the body of f. Since, rule 4 neglects this, a new rule has been presented:

Rule 5 (saving estimations) *If the body of f, after inlining, will shrink by further optimizations to become smaller than the overhead required for a function call, then f may be inlined.*

For rule 5, two research works have been realized to design algorithms which estimate the savings of further compiler optimizations. The first one, due to Appel [2], estimates the savings of the other optimizations without applying them. The second, due to Dean and Chambers [6], uses an *inline-trial* scheme: rather than just estimating the savings of other optimizations, these are applied and the savings are measured by inspecting the result of the compilation. To limit the number of required compilations, each inlining decision is stored in a persistent database. Before lauching a full compilation of a call site function, the inlining decision algorithm scans its database to find a similar call (a call to the same function with the same kind of parameters). Rule 5 has, however, two important drawbacks:

- The two technics estimate the growing impact of the other optimizations on the body of a called function f in the context of a specific call site included in a function g. The increase comes from static informations known about the actual parameters. None of these technics compute the savings of applying the other compiler optimizations on the body of g. After inlining, some static informations could be known about a function result. For instance, in the following program:

```
1: (define (inc i x)
2:    (if (fixnum? i)
3:        (fixnum+ i x)
4:        (flonum+ i x)))
5:
6: (define (foo x)
7:    ...
8:       (let ((y (inc 1 x)))
9:          ...)
```

  Inlining the call to `inc` in `foo` (line *8*) allows better compilation of the body of `inc` because, since `i` is bound to the fixnum constant `1`, the test `(fixnum? i)` (line *2*) can be removed. After inlining and test reduction, it appears that `y` can only be bound to a fixnum. This information can be used to improve the compilation of `foo`.

  The two presented technics do not estimate the savings on the body of the function which contains the inlined call because it is too far long to compute it.

- The saving estimations can only be computed for local optimizations. Global optimizations (such as *inter-procedural register allocations* [4] or *control flow analysis* [12]) require compilation of the whole program and their results cannot be predicted. Because these optimizations could have high complexity (exponential for graph coloring, $\mathcal{O}(n^3)$ for the *control flow analysis*) it is, in pratice, impossible to apply them each time a function could be inlined. Generally, these optimizations have a much more important impact than local optimizations

Because saving estimations are computed on a too restrictive set of optimizations, we think rule 5 is useless. It fails in measuring the real impact of inlining in further optimizations.

2.4 Profile based decision

Some inlining decision algorithms use profile informations. Programs are ran with various sets of input data and statistics are gathered. Inlining decisions are taken in regard of these statistics. Two papers present such works [9, 7]. They are based on the same rule:

Rule 6 (profiling statistics) *When profiling statistics show that the execution time of an invocation of a function f is longer than the execution time of the evaluation of the body of f, f could be inlined.*

This rule can be merged with rule 2 to prevent too large code growth.

We do not agree with the philosophy of profiling based decision because it requires too many help from the programmer: judicious set of execution must be designed and many compilations are needed. Other optimization tools exist which require such programmer help and we think they are rarely used (for instance, how many DEC programmers are using the `cord` + `ftoc` optimizers ?).

3 The inlining decision algorithm

None of the decision algorithms presented in section 2 appears to be much convincing, hence we have designed our own.

3.1 The input language

The input language of our algorithm is very simple. It can be seen as a small Scheme [8] language. It is described in the grammar below:

Syntactic categories

v	$\in$	VarId	(Variables identifier)
f	$\in$	FunId	(Functions identifier)
Λ	$\in$	Exp	(Expressions)
k	$\in$	Cnst	(Constant values)
Π	$\in$	Prgm	(Program)
Γ	$\in$	Def	(Definition)

Concrete syntax

```
Π ::=  Γ ... Γ
Γ ::=  (define (f v ... v) Λ)
Λ ::=  k
     | v
     | (let ((v ... v) ... (v ... v)) Λ)
     | (set! v Λ)
     | (labels ((f (v ... v) Λ) ... (f (v ... v) Λ)) Λ)
     | (if Λ Λ Λ )
     | (begin Λ ... Λ)
     | (f Λ ... Λ)
```

A program is composed of several global function definitions; local functions are introduced by the `labels` special form. Other constructions are regular Scheme constructions.

3.2 Principle of the algorithm

Our algorithm uses static informations. The decision to expanse a call site depends on the size of the called function, the size of the call (i.e. the number of actual parameters) and the place where the call is located. Our decision algorithm does not require user annotation nor profiling statistics. The idea is to allow constant code growth for each call site of the program. On each call site, inlining is performed until no more calls can be expansed or until the resulting code size has reached the limit of allowed code growth.

We illustrate the algorithm behaviour on the following example:

```
 1: (define (inc-fx x)
 2:    (+fx x 1))
 3:
 4: (define (inc-fl x)
 5:    (inc-fx (inexact->exact x)))
 6:
 7: (define (inc x)
 8:    (if (integer? x)
 9:        (inc-fx x)
10:        (inc-fl x)))
11:
12: (define (foo x)
13:    (inc x))
```

Let us suppose a very inlining agressive mode where call sites can become 5 times larger after inline expansion. The line *13* call to `inc` has a size of 2 (1 for the called function plus 1 for the formal argument). The allocated inlining amount for this call site is of 10 (the factor 5 times the call site size). The body of `inc` has a size of 7 (1 for the conditional, 2 for the test and 2 for each branche of the conditional). Hence, expansion is allowed on the call of line *13*.

Before expansing the body of `inc` in line *13*, the inlining process is launched on the body of the function with an amount of 3 (10, for the initial amount, minus 7, the size of the function). The inlining process reaches line *9* where is located the call to `inc-fx`. The size of the body of this function is 3 (1 for the `+fx` operator call and 1 for each actual argument). Since the amount is 3, this call is expansed. No further inlining can be performed on the body of `inc-fx` because `+fx` is an operator. The inlining process then reaches the call of line *10*. The call to `inc-fl` is inlined, the amount is set to 1 (3 minus 2 for the current call) and the inner call to `inc-fx` (line *5*) is reached. This call cannot be expansed because the amount of code growth is lesser than the body of `inc-fx`.

After the inlining process completes, the resulting code is:

```
(define (inc-fx x)
   (+fx x 1))

(define (foo x)
   (if (integer? x)
       (+fx x 1)
       (inc-fx (inexact->exact x))))
```

Dead functions have been removed (`inc` and `inc-fl`) and, as one can notice, the total size of the program is now smaller *after* inline expansion. Experimental results (see section 5) show that this phenomenon is frequent: in many cases, our inline expansion reduces the resulting code size.

3.3 The algorithm

The main part of the algorithm is a walk across the abstract syntax tree. All function definitions are scanned in a random order. The inlining function process ($\mathcal{I}_{ast}$, presented in algorithm 3.1) takes three arguments: the abstract syntax tree (τ), an inlining amount (a) and a set of functions ($\mathcal{S}$). $\mathcal{I}_{ast}$ returns new abstract trees.
Inlining of application function nodes are managed by $\mathcal{I}_{app}$ function (defined in algorithm 3.2).
A function call is inlined if the $\mathcal{I}?_{app}$ predicate (algorithm 3.3) is satisfied. As one can notice, the inlining decision is context dependent. A same function can be inlined on one call site and left unexpansed on another site. A function call is inlined

```
I( Π )=
  ∀f∈Π↓definitions
    f↓body ← I_ast( f↓body, ⊤, ∅ )

I_ast( S, a, τ )=
  case τ
    ⟦ k ⟧:
      τ
    ⟦ v ⟧:
      τ
    ⟦ (let ((v0 τ0) ...) τ) ⟧:
      let τ'0=I_ast( S, a, τ0 ),
          ⋮
      let τ'=I_ast( S, a, τ )
        ⟦ (let ((v0 τ'0) ...) τ') ⟧
    ⟦ (set! v val) ⟧:
      ⟦ (set! v I_ast( S, a, val )) ⟧
        ⋮
    ⟦ (f a0 ...) ⟧:
      let a'0=I_ast( S, a, a0 ), ...
        I_app( S, a, f, a'0, ... )
  end
```

Algorithm 3.1: The abstract syntax tree walk

if its code size growth (which is the substraction of the call site size and the function body size) is smaller than the current inlining amount. A first test checks if the called function is not included in $\mathcal{S}$ in order to prevent infinite loops during the inlining process. Loops are avoided because a call to a function f cannot be expansed when expansing a previous call to f.

```
I_app( S, a, f, a0, ... )=
  let a'=
        if a is ⊤
          then get-amount( f )
          else a
    if I?_app( S, a', f )
      then I_expanse( S, a', f, a0, ... )
      else ⟦ (f a0 ...) ⟧
```

Algorithm 3.2: The application inlining

The expansion of a call to a function f is computed by $\mathcal{I}_{expanse}$ (algorithm 3.4). It replaces the call by a new version of the body of f. This new version is a copy of the original body, α-converted and recursively inlined. To recursively enter the inlining process, a new amount (obtained by substracting the size of the body of f to the current amount) and a new set of functions (extended to f) are computed. New inlining amounts are computed by the function *get-amount* (algorithm 3.5). It uses function arity and a threshold ($\mathcal{K}$) that can be set by the compiler user. When a function has a large number of formal parameters, its calls are large nodes in the abstract syntax tree and it is valuable to inline them. Using arity, *get-amount* allows inlining of functions having many formal parameters which solves the problem mentionned in section 2.2. The formula computed by *get-amount* has been tuned studying statistics of many program behaviours.

```
I?_app( S, a, f )=
  let size=ast-size( f↓body )
    if f∈S
      then false
    else if size < f↓arity
      then true
    else if size < a
      then true
    else if f is only called once in Π
      then size < κ * a
    else false
```

Algorithm 3.3: The application inlining predicate

```
I_expanse( S, a, f, a0, ..., an )=
  let a'=a − (ast-size( f↓body ) − f↓arity),
  let β=f↓body,
  let β'=I_ast( {f}⋃S, a', β ),
  let x0=f↓formals0,
      ⋮
  let xn=f↓formalsn
    ⟦ (let ((x0 a0) ... (xn an)) β') ⟧
```

Algorithm 3.4: The inlining expansion

```
get-amount( f )=
  2 * ( ( 2 * K ) + ( 1 + f↓arity ) )
```

Algorithm 3.5: Computing inlining amount

4 Inlining expansion

Section 3 described the algorithm to decide *when* a function call should be inlined. In this section we show *how* a call should be inlined. Function are divided into two classes: *non-recursive functions* and *recursive ones*. The previous definition of function $\mathcal{I}_{expanse}$ has to be modified (algorithm 4.1).

4.1 Inlining of non-recursive functions

The inlining of a call to a non-recursive function has been shown in algorithm 3.4. The body of $\mathcal{I}_{let}$ is the body of the old definition of $\mathcal{I}_{expanse}$. Non recursive function inlining is a simple β-expansion: it

$$\mathcal{I}_{let}\left[\!\!\left[\begin{array}{l}\texttt{(define (}\mathit{fun}\ f_0\ \ldots\ f_n\texttt{)}\\ \quad\Lambda\texttt{)}\\ \\ \ddots\\ \quad\boxed{\texttt{(}\mathit{fun}\ a_0\ \ldots\ a_n\texttt{)}}\\ \qquad\ddots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{(define (}f\ x_0\ \ldots\ x_n\texttt{)}\\ \quad\Lambda\texttt{)}\\ \\ \ddots\\ \quad\boxed{\begin{array}{l}\texttt{(let ((}x_0\ a_0\texttt{)}\ \ldots\ \texttt{(}x_n\ a_n\texttt{))}\\ \quad\Lambda\texttt{)}\end{array}}\\ \qquad\ddots\end{array}\right]\!\!\right]$$

Figure 1: The inlining of non-recursive functions

$\mathcal{I}_{expanse}$($\mathcal{S}$, a, f, a_0, ..., a_n)=
 if f is a self recursive function ?
 then $\mathcal{I}_{rec}$($\mathcal{S}$, a, f, a_0, ..., a_n)
 else $\mathcal{I}_{let}$($\mathcal{S}$, a, f, a_0, ..., a_n)

Algorithm 4.1: The full inlining expansion

binds formal parameters to actual parameters and copies the body of the called function. Figure 1 illustrates this transformation.

4.2 Inlining of recursive functions

Self-recursive functions can be inlined using the transformation $\mathcal{I}_{let}$ of figure 1 but a more valuable transformation can be applied: rather than unrolling recursive calls to a certain depth, local recursive definitions are created for the inlined function. When inlining recursive function f, $\mathcal{I}_{rec}$ (figure 2) creates a local definition f' and replaces the call to f by a call to the f' function. It is more valuable to create local function than unrolling some function calls because the constant propagation and other local optimizations are not bound, anymore, to the depth of the unrolling; they are applied to the whole body of the inlined function.

$\mathcal{I}_{rec}$($\mathcal{S}$, a, f, a_0, ..., a_n)=
 let $a'=a - \mathit{ast\text{-}size}(f{\downarrow}_{body})$,
 let $\beta=f{\downarrow}_{body}$,
 let $\beta'=\mathcal{I}_{ast}(\{f\}\bigcup\mathcal{S}, a', \beta)$,
 let $x_0=f{\downarrow}_{formals_0}$,
 $\vdots$
 let $x_n=f{\downarrow}_{formals_n}$,
 let $\mathit{call}=(f\ a_0\ \ldots\ a_n)$
 $[\![$ (labels ((f (x_0 ... x_n) β')) call) $]\!]$

Algorithm 4.2: The recursive inlining expansion

The transformation performed by $\mathcal{I}_{rec}$ is presented in figure 2.

We show the benefit of the $\mathcal{I}_{rec}$ by presenting the processed transformations on the following Scheme example:

```
(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 detects that `map` is self-recursive, so it inlines it using a local definition:

```
(define (map-succ l)
   (labels ((map' (f l)
               (if (null? l)
                   '()
                   (cons (f (car l))
                         (map' f (cdr l))))))
      (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 obtain:

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

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

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

Thanks to the recursive-inlining scheme and to

$$\mathcal{I}_{rec}\left[\!\!\left[\begin{array}{l} \texttt{(define (}fun\ x_0\ \ldots\ x_n\texttt{)} \\ \quad \Lambda\texttt{)} \\ \ddots \\ \quad \boxed{\texttt{(}fun\ a_0\ \ldots\ a_n\texttt{)}} \\ \qquad \ddots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{(define (}f\ x_0\ \ldots\ x_n\texttt{)} \\ \quad \Lambda\texttt{)} \\ \ddots \\ \quad \boxed{\begin{array}{l}\texttt{(labels ((}f'\ \texttt{(}x_0\ \ldots\ x_n\texttt{)}\ \Lambda\texttt{))} \\ \quad \texttt{(}f'\ \ a_0\ \ldots\ a_n\texttt{))}\end{array}} \\ \qquad \ddots \end{array}\right]\!\!\right]$$

Figure 2: The inlining of recursive functions

constant propagation, some closure allocation are avoided and computed calls are turned into direct calls. The whole transformation *speeds up* the resulting code and it may *reduce* the resulting code size (because the code for allocating closure is not needed any more).

5 Experimental results

progs	*size*	*description*
Queen	131	Queens resolution.
Confo	596	Lattice management.
Bague	103	Baguenodier game resolution.
Peval	643	Partial evaluator.
Byte	582	Scheme byte code interperter.
Boyer	640	Boyer completion.
Parse	670	Earley parser implementation.
Maze	800	Shivers' maze constructor.

Figure 3: Experimental results: program descriptions

We have implemented the inlining decision and the inlining technics of recursive functions into our *Scheme & ML* compiler (Bigloo, which is available by anonymous ftp on ftp.inria.fr [192.93.2.54]). Hence, we have been able to measure the impact of the optimization. We used 8 Scheme programs described in figure 3. For each of these programs, we have measured object file size, compilation time, execution time. Time are user+system time. For each program, we made 3 measures. The first without inlining; the second, with inlining threshold $\mathcal{K}$ set to 1 and the last, with $\mathcal{K}$ set to 2. All figures have been measured on a Dec Alpha 3100 with 64 Megabytes of memory. Time figures are given in figures 4, 5 and 6.

	Without inlining		
progs	**size**	**comp**	**run**
Queen	19 k	5.2 s	9.2 s
Confo	81 k	27 s	14.0 s
Bague	25 k	4.9 s	13.7 s
Peval	79 k	58.3 s	7.4 s
Byte	57 k	35.1 s	10.0 s
Boyer	32 k	7.4 s	3.7 s
Parse	35 k	18.4 s	9.3 s
Maze	39 k	13.1 s	7.3 s

Figure 4: Without inlining

	Threshold = 1					
progs	**size**	δ	**comp**	δ	**run**	δ
Queen	19 k	*.94*	5.2 s	*1.0*	9.1 s	*.99*
Confo	68 k	*.83*	28.2 s	*1.04*	13.0 s	*.93*
Bague	23 k	*.92*	4.9 s	*1.0*	7.7 s	*.56*
Peval	78 k	*.99*	76.4 s	*1.31*	6.9 s	*.93*
Byte	50 k	*.88*	35.1 s	*1.0*	8.9 s	*.89*
Boyer	28 k	*.87*	7.8 s	*1.05*	3.4 s	*.92*
Parse	33 k	*.94*	19.4 s	*1.05*	9.0 s	*.97*
Maze	33 k	*.85*	15.6 s	*1.19*	6.7 s	*.92*
Average		*.90*		*1.08*		*.89*

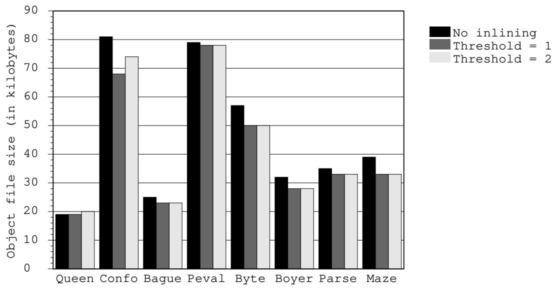
Figure 5: Inlining threshold = 1

5.1 The impact of inlining on the object file size

Our inlining optimization reduces the object file sizes. With the threshold set to 1, the reduction is about 10 % and with the threshold set to 2, the reduction is about 7 %. The figure shows that our inlining decision algorithm succeeded in eliminating the drawbacks of the previous regular inlining technics. The following chart compares the object file size obtained with the three compilation modes.

	Threshold = 2					
progs	**size**	δ	**comp**	δ	**run**	δ
Queen	20 k	*1.05*	5.4 s	*1.03*	9.1 s	*.99*
Confo	74 k	*.91*	32.6 s	*1.20*	10.8 s	*.77*
Bague	23 k	*.92*	5.0 s	*1.02*	7.7 s	*.56*
Peval	78 k	*.99*	74.7 s	*1.28*	6.9 s	*.93*
Byte	50 k	*.88*	35.6 s	*1.01*	8.9 s	*.89*
Boyer	28 k	*.87*	7.8 s	*1.05*	3.3 s	*.89*
Parse	33 k	*.94*	19.7 s	*1.07*	9.0 s	*.97*
Maze	33 k	*.85*	16.9 s	*1.29*	6.3 s	*.86*
Average		*.93*		*1.12*		*.86*

Figure 6: Inlining threshold = 2

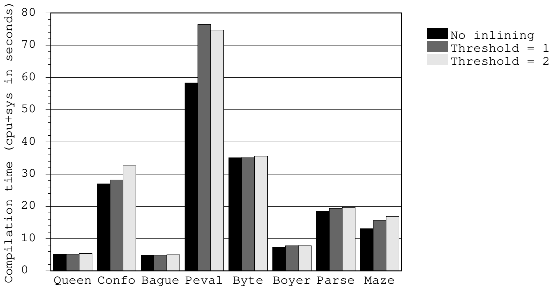


Succeeding in reducing the object code size, our algorithm performs much better than previous inlining technics: Hwu and Chang reported a code growing of 17 % in [7], Scheifler noticed an average expansion of 10 % in [9] and Davidson and Holler reported a 24 % code growing.

The exception comes from Chambers. He showed that inlining reduces the object file size produced by its Self compiler [5] by a very large factor (in the best case, inlining reduces the object file size from a factor of 4). This impressive impact of inlining comes from the nature of the source language rather than the employed technic: Self is an object oriented language and inlining is the master piece to avoid dynamic dispatches (because inlining enlarges the impact of local optimizations).

5.2 The impact of inlining on compilation times

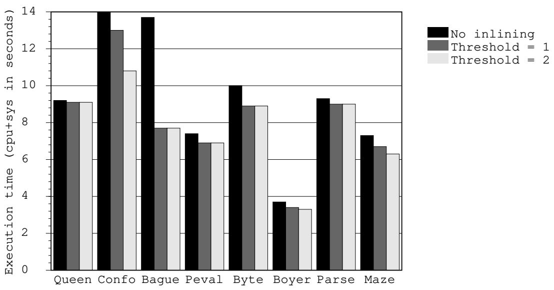
Inlining slows down the compilation process because duplicating function bodies enlarges the abstract syntax tree. The following chart illustrates the impact of inlining on compilation times.



When threshold is set to 1, compilation time is increased of 8 % and with a threshold of 2, compilation time is increased of 12 %. In both cases, we think this extra compilation time is a minor drawback.

5.3 The impact of inlining on execution times

Measured programs show that inlining always speeds up execution time. The improvement could be tiny (0.99 factor for the program Queen) or very large (.56 factor for the program Bague). An important remark is that for none of the programs tested, inlining slows down execution time; inlining is always a valuable optimization. The following chart illustrates the comparison of execution time, with enabled or disabled inline expansion.



The inlining impact depends on the performances of the used computer as it depends on the cost of function calls. Hence, comparison with previous works on inlining speed up is not really significant. However, with a threshold of 1, programs run 10 % faster than those compiled without any inlining and with a threshold of 2, they run 15 % faster than those compiled without any inlining.

Chambers reports that enabling functional inlining, programs can run between 4 and 55 times faster. As for compilation time, this improvement comes from the source language. Inlining helps the static compilation of generic functions and allows the removal of dynamic generic dispatches.

Scheifler noticed that average runtime gain for CLU programs is about 14 % while Davidson noticed that average runtime gain for C programs is of 12 %. Our inlining technics lead to comparable gains.

6 Conclusion

In this paper, we have presented two new inlining technics. The first one is a new decision algorithm based on static compile-time informations and is easy to implement. The second technic is an original framework for inlining recursive functions. Both technics have been implemented in Bigloo, our *Scheme & ML* compiler. Experimental results show that inlining has a minor impact on compilation time (12 % longer in average), reduces object file size (of about 10 %) and execution times (of about 15 % with the maximum inlining level). Our inlining technics have the same advantages as previous inlining technics, they speed up executions; however, at the same time, and in contrast to previous inlining technics, they reduce object files size.

References

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

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

[3] L. Cardelli and al. Modula-3 Report (revised). Technical Report 52, DEC Systems Research Center, Palo-Alto, CA, USA, November 1989.

[4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Language*, 6:47–57, 1981.

[5] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Technical report stan-cs-92-1240, Stanford University, Departement of Computer Science, March 1992.

[6] J. Dean and C. Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Conference on Lisp and Functional Programming*, pages 273–282, Orlando, Florida, USA, June 1994.

[7] W. Hwu and P. Chang. Inline Function Expansion for Compiling C Programs. In *Conference on Programming Language Design and Implementation*, Portland, Oregon, USA, June 1989. ACM.

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

[9] R.W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. *CACM*, 20(9):647–654, September 1977.

[10] M. Serrano. Bigloo user's manual. RT 0169, INRIA-Rocquencourt, France, December 1994.

[11] M. Serrano. *Vers une compilation portable* et *performante des langages fonctionnels*. Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris VI), Paris, France, December 1994.

[12] M. Serrano. Control Flow Analysis: a Functional Languages Compilation Paradigm. In *10th Symposium on Applied Computing*, Nashville, Tennessee, USA, February 1995.

[13] R. Stallman. Using and Porting GNU CC. for version 2.7.2 ISBN 1-882114-66-3, Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, November 1995.

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.