Inline expansion: *when* and *how*?

Manuel Serrano
Manuel.Serrano@cui.unige.ch
http://cuiwww.unige.ch/~serrano/

Centre Universitaire d'Informatique, University of Geneva
24, rue General-Dufour, CH-1211 Geneva 4, Switzerland

Abstract. Inline function expansion is an optimization that may improve program performance by removing calling sequences and enlarging the scope of other optimizations. Unfortunately it also has the drawback of enlarging programs. This might impair executable programs performance. In order to get rid of this annoying effect, we present, an easy to implement, inlining optimization that minimizes code size growth by combining a compile-time algorithm deciding *when* expansion should occur with different expansion frameworks describing *how* they should be performed. We present the experimental measures that have driven the design of inline function expansion. We conclude with measurements showing that our optimization succeeds in producing faster codes while avoiding code size increase.
Keywords: Compilation, Optimization, Inlining, Functional languages.

1 Introduction

Inline function expansion (henceforth "inlining") replaces a function invocation with a modified copy of the function body. Studies of compilation of functional or object-oriented languages show that inlining is one of the most valuable optimizations [1, 4]. Inlining can reduce execution time by removing calling sequences and by increasing the effectiveness of further optimizations:

- Function call sequences are expensive because they require many operations (context save/restore (i.e. memory fetches) and jumps). For small functions, the call sequence can be more expensive than the function body.
- Inlining can improve the effectiveness of the other compiler optimizations because inlined function bodies are modified copies. The formal function parameters are replaced with (or bound to) the actual parameters. Known properties of 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 may become monomorphic. Finally, inlining helps the optimization of both the called and the calling function. Information about actual parameters can be used to improve the compilation of an inlined function body; information about the result of an inlined function can be 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, execution can become slower. On modern architectures, best performance is obtained when a program fits into both the instruction and data caches. Clearly, smaller executables are more likely to fit into the instruction 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 inlining decision algorithm. This paper focuses on this issue. It presents a compile-time algorithm which allows fine control of code growth and does not require profiling information or user annotations.

Functional languages are characterized by extensive use of functions and particularly recursive functions. Both types of functions can be inlined within a unique framework but a refined expansion can be achieved for recursive functions. This paper presents an efficient framework for inlining recursive functions. Experimental measurements show this framework to be one of the keys to controlling code size growth. This is done by a combination of an inlining decision algorithm (deciding *when* to perform the expansions) and *ad-hoc* expansion frameworks (describing *how* to perform the expansions).

Inlining may impair code production because of los of high level informations. For instance, Cooper *et al.* report in [5] that inlining discards aliasing informations in Fortran programs, so that compilers are unable to avoid interlock during executions. Davison and Holler show in [6] that inlining for C may increase register save and restore operations because of C compilers artifacts. On our part, we have never noticed decreases of performance when activating inlining optimization.

This paper is organized as follows: section 2 presents a study of previous algorithms and schemes. Section 3 presents our new algorithm. Section 4 presents the different inlining frameworks. Section 5 reports on the impact of the inlining expansion optimization in Bigloo, our Scheme compiler.

2 Inline expansion: *when*

2.1 Previous approaches

This section contains a presentation of the main inlining decision rules and algorithms previously published. The remarks made in this section are the basis for the design of the algorithm presented in section 3.

User inlining indications Some systems (e.g. gcc) or language definitions (e.g. C++) allow programs to contain inlining annotations.

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

Inlining annotations are useful to implement some parts of the source language (for instance, in Bigloo [12], our *Scheme & ML* compiler, inlining annotations are used intensively in the libraries for accessing and mutating primitives like `car`, `cdr`, `vector-ref` or `vector-set!`) but cannot replace automatic inlining decisions. By contrast an automatic inlining decision algorithm can choose to inline a function at a call site, but also not to do it at another call site. User inlining annotations are attached to function definitions and, as a consequence, are not call site dependent. As a result, misplaced inlining annotations can lead to code size explosion. Furthermore, inlining annotations require the programmer to have a fair understanding of the compiler strategy in order to be able to place annotations judiciously.

Size-based criteria Simpler inlining decision algorithms [14, 4] are based on the body size of called functions:

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

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 c calls can increase the code size by at most $c \times \mathcal{K}$. In order to prevent infinite loops when inlining recursive functions, rule 2 has to be extended:

Rule 3 (body-size & nested-call) *Let f be 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 helps preventing code explosion and is 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, this `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 [1] an improvement of Rule 3:

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.*

Savings estimates 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 (constants, for instance). Other optimizations such as *constant folding* may be able to shrink the modified version of the body of f. Since Rule 4 neglects this, we define:

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 the call to f may be inlined.*

Two implementations of this approach have been described. The first one, due to Appel [1], estimates the savings of the other optimizations without applying them. The second, due to Dean and Chambers [7], 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 launching 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 techniques estimate the 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. Neither computes the savings of applying the other compiler optimizations on the body of g, due to the added expense. However, after inlining, additional information may be known about a function result. For instance, in the following Scheme program:
```
1: (define (inc i x)
2:    (if (fixnum? i)
3:        (fixnum+ i x)
4:        (flonum+ i x)))
5: (define (foo x)
6:    ...
7:       (let ((y (inc 1 x)))
8:          (if (fixnum? y)
9:             ...)))
```
  Inlining the call to `inc` in `foo` (line *7*) 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` (by removing, for instance, the line *8* test).
- The saving estimations can only be computed for local optimizations. Global optimizations (such as *inter-procedural register allocations* or *control flow analysis*) require compilation of the whole program and their results are mostly unpredictable. Because these optimizations are often slow, it is, in practice, impossible to apply them each time a function could be inlined.

Because saving estimations are computed on an overly restricted set of optimizations, we think rule 5 is not highly efficient. It fails to measure the real impact of inlining in further optimizations.

Profile-based decision Some inlining decision algorithms use profile information. Programs are run with various sets of input data and statistics are gathered. Inlining decisions are taken based on these statistics. Two papers present such works [11, 8]. They are based on the same rule that can be merged with rule 2 to prevent excessive code growth:

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.*

We do not think profile-based decision algorithms are practical because they require too much help from the programmer. A judicious set of executions must be designed and many compilations are needed.

3 The inlining decision algorithm

From the remarks of Section 2.1, we have designed our own inlining decision algorithm.

3.1 The input language

The input language of our algorithm is very simple. It can be seen as a small Scheme [9] language with no higher order functions. It is described in the grammar below:

<u>Syntactic categories</u>

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)

<u>Concrete syntax</u>

$$
\begin{array}{ll}
\Pi ::= & \Gamma \ldots \Gamma\ \Lambda \\
\Gamma ::= & (\texttt{define}\ (f\ v\ \ldots\ v)\ \Lambda)
\end{array}
$$

$$
\begin{array}{rl}
\Lambda ::= & k \\
& \mid v \\
& \mid (\texttt{let}\ ((v\ \Lambda)\ \ldots)\ \Lambda) \\
& \mid (\texttt{set!}\ v\ \Lambda) \\
& \mid (\texttt{labels}\ ((f\ (v\ \ldots\ v)\ \Lambda)\ \ldots)\ \Lambda) \\
& \mid (\texttt{if}\ \Lambda\ \Lambda\ \Lambda) \\
& \mid (\texttt{begin}\ \Lambda\ \ldots\ \Lambda) \\
& \mid (f\ \Lambda\ \ldots\ \Lambda) \\
& \mid (\texttt{+}\ \Lambda\ \Lambda)
\end{array}
$$

A program is composed of several global function definitions and of one expression used to initiate computations. Local recursive functions are introduced by the `labels` special form. Other constructions are regular Scheme constructions.

3.2 Principle of the algorithm

Our algorithm uses static information. The decision to expand 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 or profiling statistics. Inspired by [1, page 92] the idea of the algorithm is to allow code growth by a certain factor for each call site of the program. When a call is inlined, the algorithm is recursively invoked on the body result of the substitution. The deeper the recursion becomes, the smaller the factor is.

We illustrate the algorithm's behavior on the following example:

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

Suppose that at recursion depth zero we permit call sites to become 4 times larger and we make each recursive call to the inlining algorithm divide this multiplicative factor by 2 (later on, we will study the impact of the choice of the regression function). The line *8* call to `inc` has a size of 2 (1 for the called function plus 1 for the formal argument). The body size of `inc` is 7 (1 for the conditional, 2 for the test and 2 for each branch of the conditional). Hence, the call is expanded. Before expanding the body of `inc` in line *8*, the inlining process is launched on the body of the function with a new multiplicative factor of 2 (half the initial factor of 4). The inlining process reaches line *6*, the call to `inc-fx`. The size of the body of this function is 3 (1 for the `+` operator call and 1 for each actual argument), the call size is 2 hence this call is expanded. No further inlining can be performed on the body of `inc-fx` because `+` is a primitive operator. The inlining process then reaches the call of line *7*. The call to `inc-fl` is inlined, the multiplicative factor is set to 1 and the inner call to `inc-fx` (line *3*) is reached. This call cannot be expanded because the amount of code growth is less than the body of `inc-fx`. After the inlining process completes, the resulting code is:

```
1: (define (inc-fx x) (+ x 1))
2: (define (foo x)
3:    (if (fixnum? x)
4:      (+ x 1)
5:      (inc-fx (inexact->exact x))))
6: (foo 4)
```

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 situations, our inline expansion reduces the resulting code size.

3.3 The algorithm

The main part of the algorithm is a graph traversal of the abstract syntax tree. All function definitions are scanned in a random order (the scanning order has no impact on the result of the optimization). The inlining process, $\mathcal{I}_{ast}$ (algorithm 3.1), takes three arguments: a multiplicative factor (k), a set of functions ($\mathcal{S}$) and an abstract syntax tree ($\mathcal{A}$). It returns new abstract syntax trees.

Function calls satisfying the $\mathcal{I}_{app?}$ predicate are inlined using the $\mathcal{I}_{app}$ function (algorithm 3.2). As one can notice, the inlining decision is context dependent. A given function can be inlined on one call site and left unexpanded on another site. A function call is inlined if its code size growth factor is strictly

```
𝒦:   an external user parameter

ℐ( Π )=
   ∀f∈Π↓definitions
     f↓body ← ℐast( 𝒦, ∅, f↓body )

ℐast( k, 𝒮, Λ )=
   case Λ
     ⟦ k ⟧:
       Λ
     ⟦ v ⟧:
       Λ
     ⟦ (let ((v0 Λ0) ...) Λ) ⟧:
       let Λ'0=ℐast( k, 𝒮, Λ0 ), ...
         ⟦ (let ((v0 Λ'0) ...) ℐast( k, 𝒮, Λ )) ⟧
     ⟦ (set! v Λ) ⟧:
       ⟦ (set! v ℐast( k, 𝒮, Λ )) ⟧
       ⋮
     ⟦ (f a0 ...) ⟧:
       let a'0=ℐast( k, 𝒮, a0 ), ...
         ℐapp( k, 𝒮, f, a'0, ... )
   end
```

Algorithm 3.1: The abstract syntax tree walk

smaller than the value of k. This criteria is strong enough to avoid infinite recursions. This current version of $\mathcal{I}_{app?}$ does not make use of the $\mathcal{S}$ argument. Later versions (Section 4.2) will. The expansion of a call to a function f is computed by $\mathcal{I}_{let}$ (algorithm 3.2). 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 factor is computed. Section 5.1 will study the impact of the $\mathcal{D}ec$ function on the inlining process.

```
ℐapp( k, 𝒮, f, Λ0, ..., Λn )=
   if ℐapp?( k, 𝒮, f, n + 1 )
     then ℐlet( k, 𝒮, f, Λ0, ..., Λn )
     else ⟦ (f Λ0 ... Λn) ⟧

ℐapp?( k, 𝒮, f, csize )=
   function-size( f↓body ) < k * csize

ℐlet( k, 𝒮, f, Λ0, ... )=
   let x0=f↓formals0, ...
     ⟦ (let ((x0 Λ0) ...) ℐast( 𝒟ec( k ), {f}⋃𝒮, f↓body )) ⟧
```

Algorithm 3.2: The let-inline expansion

3.4 Inlining in presence of higher-order functions

Inlining a function call requires knowing which function is called in order to access its body. In the presence of higher-order functions, the compiler sometimes does not know which function is invoked. The algorithm presented above can be extended to accept higher-order functions by adding the straightforward rule that calls to unknown functions are not candidates to expansion. In order to enlarge the set of the possibly inlined function calls of higher order languages, Jagannathan and Wright have proposed in [10] to apply a control flow analysis before computing the inline expansion. For each call site of a program, the control flow analysis is used to determine the set of possibly invoked functions. When this set is reduced to one element the called function can be inlined. This work is complementary to the work presented here. Jagannathan and Wright enlarge the set of inlining candidates, while we propose an algorithm to select which calls to inline from this set.

4 Inline expansion: *how*

Section 2 described the algorithm to decide *when* a function call should be inlined. In this section we show *how* a functional call should be inlined. Functions are divided into two classes: *non-recursive* and *recursive* ones.

4.1 Inlining of non-recursive functions (let-inlining)

The inlining of a call to a non-recursive function has been shown in algorithm 3.2. Non-recursive function inlining is a simple β-reduction: it binds formal parameters to actual parameters and copies the body of the called function.

4.2 Inlining of recursive functions (labels-inlining)

Self-recursive functions can be inlined using the transformation $\mathcal{I}_{let}$ but a more valuable transformation can be applied: rather than unfolding recursive calls to a certain depth, local recursive definitions are created for the inlined function (following the scheme presented in [13]). When inlining a recursive function f, $\mathcal{I}_{labels}$ (algorithm 4.1) creates a local definition and replaces the original target of the call with a call to the newly created one. It is more valuable to introduce local functions than unrolling some function calls because the constant propagation and other local optimizations are no longer limited to the depth of the unrolling; they are applied to the whole body of the inlined function. The previous definitions of functions $\mathcal{I}_{app}$ and $\mathcal{I}_{app?}$ have to be modified. Recursive calls should not be further unfolded. This is avoided by making use of the $\mathcal{S}$ argument in the $\mathcal{I}_{app?}$ function.

We show the benefit of the $\mathcal{I}_{labels}$ 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))
```

$\mathcal{I}_{app}$(k , $\mathcal{S}$, f , Λ_0 , ..., Λ_n)=
  if $\mathcal{I}_{app?}$(k , $\mathcal{S}$, f , $n+1$)
    then if f is a self recursive function ?
      then $\mathcal{I}_{labels}$(k , $\mathcal{S}$, f , Λ_0 , ..., Λ_n)
      else $\mathcal{I}_{let}$(k , $\mathcal{S}$, f , Λ_0 , ..., Λ_n)
    else $[\![$ (f Λ_0 ... Λ_n) $]\!]$

$\mathcal{I}_{app?}$(k , $\mathcal{S}$, f , $csize$)=
  if $f \in \mathcal{S}$
    then false
    else $function\text{-}size$($f{\downarrow}_{body}$) $<$ $k * csize$

$\mathcal{I}_{labels}$(k , $\mathcal{S}$, f , Λ_0 , ..., Λ_n)=
  let $\lambda' = \mathcal{I}_{ast}$($\mathcal{D}ec$(k) , $\{f\} \bigcup \mathcal{S}$, $f{\downarrow}_{body}$),
  let $x_0 = f{\downarrow}_{formals_0}$, ...
    $[\![$ `(labels ((`f `(`x_0 ...`)` λ'`))` `(`f Λ_0 ... Λ_n`))` $]\!]$

Algorithm 4.1: The labels-inline expansion

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 replaced with its actual value `succ`. Then, `succ` is open-coded and we finally get the equivalent definition:

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

Thanks to the labels-inline expansion and to constant propagation, closure allocations 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 closures is no longer needed.

4.3 Inlining as loop unrolling (unroll-inlining)

We have experimented an *ad-hoc* inlining scheme for loops. Here we consider a loop to be any recursive function with one single inner recursive call. When a loop is to be inlined, a local recursive function is created (according to the $\mathcal{I}_{labels}$ transformation) followed by a traditional unrolling. This new expansion scheme requires the slight modifications to $\mathcal{I}_{app}$ and $\mathcal{I}_{app?}$ given in algorithm 4.2. The unrolling is actually a simple mix between the previous inlining frameworks. The transformation $\mathcal{I}_{labels}$ is applied once, followed by as many $\mathcal{I}_{let}$ transformations as the multiplicative factor k allows. We illustrate the unroll-inline transformation on the preceding `map-succ` example:

```
(define (map-succ l)
 (labels ((map (l1)
   (if (null? l1) '()
    (cons (+ 1 (car l1))
     (let ((l2 (cdr l1)))
      (if (null? l2) '()
       (cons (+ 1 (car l2))
        (map (cdr l2))
       )))))))
 (map l)))
```

$\mathcal{I}_{app}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)=
  if $\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, $n+1$)
    then if f is a self recursive function ? **and** $f \notin \mathcal{S}$
      then $\mathcal{I}_{labels}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
      else $\mathcal{I}_{let}$(k, $\mathcal{S}$, f, Λ_0, ..., Λ_n)
    else $[\![$ (f Λ_0 ... Λ_n) $]\!]$

$\mathcal{I}_{app?}$(k, $\mathcal{S}$, f, $csize$)=
  $function\text{-}size(f{\downarrow}_{body}) < k * csize$

Algorithm 4.2: The unroll-inline expansion

4.4 Related work

Little attention has been formerly given to *how* the expansion should be performed. Previous works have considered it as a straightforward β-reduction. Inlining of recursive functions has been mainly addressed into three previous papers:

- In the paper [3] H. Baker focuses on giving a semantics to the inlining of recursive functions. Inlining of recursive functions is thought as a loop unrolling by unfolding calls until a user determined depth level. The paper neither studies the impact of this inlining framework on run time performance nor attempts to present optimized transformations. We even think that the proposed transformations would probably slow down executions (because they introduce higher order functions which are difficult to implement efficiently).
- We have made a previous presentation of the labels-inline transformation in [13]. It does not focus on the inlining optimization and it merely presents the transformation without studying its impact.
- The labels-inline transformation has been used by A. Appel in [2]. His approach differs a little bit from our because Appel introduces header around every loop independently of the inlining optimization. We think this has two drawbacks. First, un-inlined loops have to be cleaned up (that is, headers have to be removed) otherwise there is an extra call overhead. More importantly, introducing header make functions abstract syntax tree bigger. Since

the inlining algorithm uses functions size to decide to inline a call, loops with header introduced are less likely to be inlined.

5 Experimental results

For experimental purposes, we used ten different Scheme programs, written by different authors, using various programming styles. Experiments have been conducted on a DEC ALPHA 3000/400, running DEC OSF/1 V4.0 with 64 Megabytes of memory.

programs	*nb lines*	*author*	*description*
Bague	104	P. Weis	Baguenodier game.
Queens	132	L. Augustsson	Queens resolution.
Confo	596	M. Feeley	Lattice management.
Boyer	640	R. Gabriel	Boyer completion.
Peval	643	M. Feeley	Partial evaluator.
Earley	672	M. Feeley	Earley parser.
Matrix	753	–	Matrix computation.
Pp	757	M. Serrano	Lisp pretty-printer.
Maze	879	O. Shivers	Maze game escape.
Nucleic	3547	M. Feeley	Molecular placement.

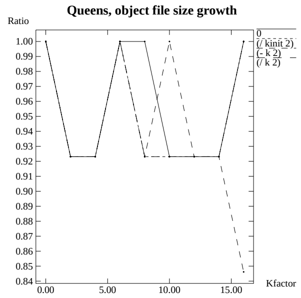
In order to be as architecture-independent as possible, we have measured duration in both user plus system cpu time and number of cpu cycles (using the `pixie` tool). For all our measures, even when the multiplicative factor is set to 0, primitive operators (such as `+` or `car`) are still inlined.

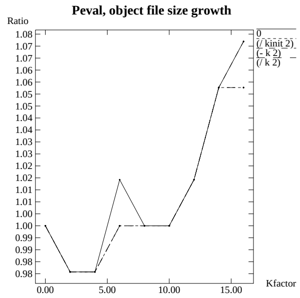
5.1 Selecting the $\mathcal{D}ec$ regression function

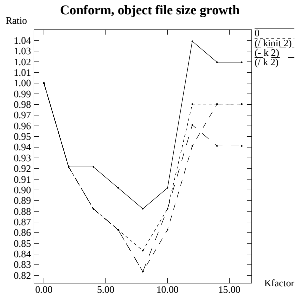
We have studied the impact of the $\mathcal{D}ec$ function, first used in algorithm 3.2. We have experimented with three kind of regression functions: decrementations: $\lambda_{-\mathcal{N}}$=`(`λ` (k) (- k `$\mathcal{N}$`))`, divisions: $\lambda_{/\mathcal{N}}$=`(`λ` (k) (/ k `$\mathcal{N}$`))` and two *step* functions: $\lambda_{=0}$=`(`λ` (k) 0)` and $\lambda_{=k_{init}/\mathcal{N}}$=`(`$\lambda$` (k) (if (= k k`$_{init}$`) (/ k `$\mathcal{N}$`) 0))`.

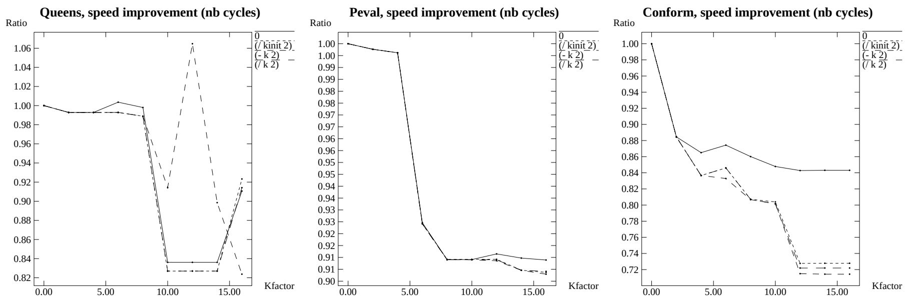
For each of these functions, we have measured the code size growth and the speed improvement. Since measurement showed that varying $\mathcal{N}$ has a small impact, we just present measurements where $\mathcal{N}$ has been set to two.

We present results for only three programs, **Queens**, **Peval** and **Conform** because they are representative of the whole 10 programs. The X axis represents the initial value of the k multiplicative factor. The Y axis of the upper graphics represents the normalized object file size. The Y axis of the lower graphics represents the normalized durations (cpu cycles).









Two remarks can be made on these measurements:

- The regression function $\lambda_{=0}$ produces less efficient executables (see for instance **Conform**) than others regression functions. This proves that a recursive inlining algorithm (an algorithm trying to inline the result of an inlined call) gives better results that a non recursive one.
- $\lambda_{=k_{init}/\mathcal{N}}$, $\lambda_{-\mathcal{N}}$ and $\lambda_{/\mathcal{N}}$ lead to about the same results. This demonstrates that only the very first recursive steps of the inlining are important. Functions like $\lambda_{-\mathcal{N}}$ or $\lambda_{/\mathcal{N}}$ have the drawback to authorize large code size expansion (about $2^{\frac{1}{\mathcal{N}} * \log_2 k^2}$ in the worst case for $\lambda_{/\mathcal{N}}$). *Step* functions like $\lambda_{=k_{init}/\mathcal{N}}$ are much more restrictive ($k^2/\mathcal{N}$ in the worst case). Choosing *step* functions leads to small and efficient compiled programs.

5.2 The general measurements

The second step of our experiment has been to study the impact of the let-inline, labels-inline and unroll-inline expansions. The results summarized in Figure 1 show the speedups and the code size increases for the 10 programs in each of the 3 frameworks. For each of them we have computed a speedup, measured as the ratio of the speed of the inlined divided by the speed of the same program with inlining disabled, and a size, measured as the ratio of the size of the inlined program divided by the size of the original program. The experiment was performed for all values of $\mathcal{N}$ (see Section 5.1 for the definition of $\mathcal{N}$) between 1 and 15. The Figure shows the results of the best speedup for each program and framework. Furthermore, we computed speedups and size increases with respect to number of cycles and actual time elapsed. Thus, in Figure 1, *cycle*/speed refer to the ratio of cycles and *cycle*/size refers to the ratio of sizes, while *sys+cpu*/speed refers to the ratio of times and *sys+cpu*/size refers to the ratio of size. Note that for a given program and framework, the value of $\mathcal{N}$ giving the best speedup is sometimes different if we measure it in cycles or elapsed time, thus the size increases may also differ.

Labels-inlining Labels-inlining is an important issue in order to minimize object file size. In the worst case, labels-inlining behaves as let-inlining, enlarging object file size, but in general, it succeeds in limiting or even avoiding expansions. This good behavior does not impact on speedup ratios.

	Best speedup for $\mathcal{N} \in$[1..15]											
Programs	*let*				*labels*				*unroll*			
	cycle		*sys+cpu*		*cycle*		*sys+cpu*		*cycle*		*sys+cpu*	
	speed	size	speed	size	speed	size	speed	size	speed	size	speed	size
bague	0.67	1.18	0.66	1.18	0.69	1.18	0.66	1.18	0.69	1.18	0.66	1.18
queens	0.86	1.53	0.89	1.53	0.82	0.92	0.85	0.92	0.81	0.92	0.89	0.92
conform	0.72	1.21	0.83	1.17	0.72	0.96	0.76	0.96	0.72	0.98	0.76	0.98
boyer	0.89	1.00	1.00	1.00	0.89	0.90	0.89	0.90	0.89	0.90	0.89	0.90
peval	0.98	1.05	1.00	1.00	0.90	1.05	0.90	1.00	0.96	1.13	0.94	1.03
earley	0.95	1.00	0.97	1.00	0.95	1.00	0.97	1.00	0.95	1.00	0.97	1.00
matrix	0.93	1.11	0.88	1.11	0.94	0.88	0.92	0.91	0.94	0.94	0.90	0.94
pp	0.94	1.22	0.90	1.11	0.95	1.10	0.92	1.07	0.95	1.14	0.92	1.09
maze	0.71	0.84	0.73	0.84	0.70	0.80	0.73	0.84	0.71	0.80	0.73	0.80
nucleic	0.99	1.15	0.95	1.15	0.98	1.15	0.98	1.10	0.98	1.15	1.00	1.00
Average	*0.86*	*1.13*	*0.88*	*1.11*	*0.85*	*0.99*	*0.86*	*0.99*	*0.86*	*1.01*	*0.87*	*0.98*

Fig. 1. Best speedups

Except for a very few programs, let-inlining does not succeed in producing faster (less cpu cycle consuming) executables than labels-inlining but it enlarges too much the object file size. For instance, the let-inlining aborts on **Earley** and **Matrix** for very high initial values of the multiplicative factor k because the abstract syntax trees of these programs become excessively large.

Unroll-inlining Few improvements come from unroll-inlining. Only **Boyer** and **Maze** are improved when this framework is used. One explanation for this result is that unroll-inlining is barely applied because most loops have only one argument (after removal of invariant parameters) and thus they can then be inlined only if their body is very small.

This is a slightly disappointing result but even the benefits of classical loop unrolling optimizations are not well established. For instance, experience with gcc in [14] is that "[general loop unrolling] usually makes programs run more slowly". On modern architectures performing dynamic instruction scheduling, loop unrolling that increases the number of tests can severely slow down execution. This was partially shown by the measures of the labels-inlining impact that showed that it is more valuable to specialize a loop rather than to unfold outer loop calls.

5.3 Related work

The inlining impact depends on the cost of function calls and thus is highly architecture-dependent. Hence, comparison with previous measures reported on inlining speedup, made on different architectures, is difficult. Furthermore, we think that it does not make sense to compare the inlining impact for very different languages. For instance, since a typical C program does not make extensive use of small functions such as a Scheme and ML program, an inlining optimizer for C should probably not adopt a framework tuned for a functional language. We limit the present comparison to a few publications.

- C. Chambers shows that inlining reduces the object file size produced by the Self compiler [4] by a very large factor (in the best case, inlining reduces the object file size from a factor of 4) while making programs run much faster (between 4 to 55 times faster). The explanation is found in [4, section B.3.1]: "In SELF, the compiler uses inlining mostly for optimizing user-defined control structures and variable accesses, where the resulting inlined control flow graph is usually much smaller than the original un-inlined graph. These sorts of inlined constructs are already 'inlined' in the traditional language environment. Inlining of larger 'user-level' methods or procedures does usually increase compile time and compiled code space as has been observed in traditional environments...".
- We share with Jagannathan and Wright [10] three test programs: **Maze**, **Boyer** and **Matrix**. For all of them, we have measured the same speed improvement but our techniques do not increase the object file size while that of Jagannathan and Wright enlarges it by 20%.
- In [2] Appel reports on an average speedup of 5% for the labels-inlining. Our measures do not allow us to conclude in a same way. We have found that the main effect of the labels-inlining is to reduce the size of the inlined programs. As suggested by Appel his improvement may come from the reduction of the closure allocations. Less closures are allocated because in a CPS style, labels-inlining and its hoisting of loop invariant arguments may avoid the construction of some continuations.

Conclusion

We have shown in this paper that the combination of a decision algorithm with different expansion frameworks makes the inline expansion optimization more valuable. It improves run time performances (about 15% on average) while avoiding its traditional drawback, the object code size growth. The decision-making algorithm we have presented is based on static compile time informations and does not require user annotations or profiling data. The expansion framework allows inlining of recursive functions. Both are easy to implement. In Bigloo, our Scheme compiler, the decision-making algorithm and the different expansion strategies constitute less than 4% of the whole source code.

Acknowledgments

Many thanks to Christian Queinnec, Xavier Leroy, Jeremy Dion, Marc Feeley, Jan Vitek and Laurent Dami for their helpful feedbacks on this work.

References

1. A. Appel. **Compiling with continuations**. Cambridge University Press, 1992.
2. A. Appel. **Loop Headers in λ-calculus or CPS**. *Lisp and Symbolic Computation*, 7:337–343, December 1994.
3. H. Baker. **Inlining Semantics for Subroutines which are recursive**. *ACM Sigplan Notices*, 27(12):39–46, December 1992.
4. 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.
5. K. Cooper, M. Hall, and L. Torczon. **Unexpected Side Effects of Inline Substitution: A Case Study**. *ACM Letters on Programming Languages and Systems*, 1(1):22–31, 1992.
6. J. Davidson and A. Holler. **Subprogram Inlining: A Study of its Effects on Program Execution Time**. *IEEE Transactions on Software Engineering*, 18(2):89–101, February 1992.
7. 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.
8. 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.
9. IEEE Std 1178-1990. **IEEE Standard for the Scheme Programming Language**. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
10. S. Jagannathan and A. Wright. **Flow-directed Inlining**. In *Conference on Programming Language Design and Implementation*, Philadelphia, Penn, USA, May 1996.
11. R.W. Scheifler. **An Analysis of Inline Substitution for a Structured Programming Language**. *CACM*, 20(9):647–654, September 1977.
12. M. Serrano. **Bigloo user's manual**. RT 0169, INRIA-Rocquencourt, France, December 1994.
13. 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), June 1994. ACM SIGPLAN, INRIA RR 2265.
14. 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.