ACM *Sigplan Notices* **27**,12 (Dec. 1992), 39-46.

Inlining Semantics for Subroutines which are Recursive

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

Abstract

Many modern programming languages offer a facility for "inlining" designated procedure and function calls, but this process is not well defined in most language standards documents. We offer a model for inlining which has the property that the resulting code does not depend upon the presentation of the order of function definitions, and which also provides a finite and consistent interpretation for the inlining of mutually recursive functions. Finally, when used on "tail-recursive" functions, our model of inlining provides for the equivalent of "loop unrolling".

Introduction

Many programming languages—e.g., Fortran, Ada, C++, Common Lisp—offer a facility for "macro-expanding" procedure and function calls, so that the operations of the procedure or function are performed "in line" with the other operations of the calling procedure rather than being performed "out of line" within a "closed" subroutine [Allen80] [Harrison77].

The goal of such an integration is to improve execution efficiency. For very small subroutines, the overheads of handling arguments and constructing stack frames can exceed the cost of the operations within the subroutine itself. With the growing popularity of highly modular programming techniques—e.g., "abstract data types"—the percentage of such small procedures has grown dramatically, and many functions of this sort are often used only to access a single variable. Since "inlining" such a subroutine eliminates function-calling overhead, major performance gains can be obtained [Scheifler77] [Stroustrup86].

Subroutine inlining (also called "procedure integration") can also open up a substantial number of additional opportunities for compiler optimizations, including constant propagation, dead code elimination, etc. As a result, inlining even large subroutines can sometimes yield substantial performance improvements due to effects equivalent to *partial evaluation* [Ershov77] [Turchin86]. In particular, entire recursions can be eliminated at compile time for certain constant arguments if the depth of the recursion is not too great.

The semantics of inlining ("procedure integration"), however, are usually not well specified in programming language standards documents. Perhaps this is because inlining is not usually thought of as a semantic issue, since whether a procedure is inlined or not should not change the values that are computed by the program. The Ada language [Ada83], for example, considers "inlining" to be a "pragma", rather than a first-class language element, and therefore leaves the compiler implementor much flexibility in the actual semantics that it implements. C++ [Stroustrup86] considers "inline" to be a "hint" to the compiler, but does not further specify what this means. We believe that this ambiguity can be a source of great confusion, greatly reducing the usefulness of this important feature. During the system integration phase of a software project, for example, the system implementors need to precisely control which subroutines are inlined, and which are not, so that they may achieve a balance between running speed and code size. We therefore propose that the semantics of procedure inlining be carefully specified to provide intuitive and useful results in the most common cases.

Inlining in Common Lisp[1]

Although other programming languages have offered "inlining" services for at least a decade, Lisp systems have been reticent about providing this capability. There are several reasons for this reticence. First, Lisp has traditionally offered powerful *macros*, which provide the programmer with much of the efficiency that other languages can obtain only through inlining. Since Lisp macros are strictly more powerful than inlining,[2] an inlining capability has usually been considered only as an afterthought. Secondly, Lisp has traditionally used dynamically-scoped variables rather than the lexically-scoped variables of Algol-derivative languages, and traditional Lisp macros already provide (very nearly) the correct semantics for inlining functions whose internal variables are all dynamically scoped (!); however, more sophisticated binding environment and/or renaming schemes are required to preserve the semantics of

[1]We use Common Lisp for many of our examples, because it is the only standardized language which provides the expressiveness to represent the different situations to be discussed—e.g., it offers a syntax for anonymous subprograms, and its parentheses explicitly delimit the scope of its lexical names.

[2]The truth of this statement depends upon the ability of the macro-expansion to query the lexical environment to determine the free/bound status of names, and the special/lexical status of variable names; this ability was only recently proposed for Common Lisp [Steele90,8.5].

Copyright © 1992 by Nimble Computer Corporation

the lexically-scoped names of modern Common Lisp during inlining. Finally, for the optimization of language-required library functions such as `mapcar`, Lisp programmers have found mechanisms such as "compiler optimizers" to be more useful and flexible than simple function inlining.[3]

Due to the availability of a syntax for anonymous functions (including function closures), the process of inlining in Lisp can be cleanly broken up into two parts: the replacement of a function name by its anonymous functional value, and the process of simplifying and optimizing the resulting expression. It should be obvious then that the simple process of inlining (i.e., name lookup) produces by itself only minor savings—the real savings come from the further simplifications and optimizations which then become possible. The code below shows the effect of the name lookup portion of inlining on a simple local function definition.

```
(flet ((foo (n) <body of foo>)) ;'flet' defines local non-recursive functions.
      (declare (inline foo))    ; command 'foo' to be inlined.
  (......(foo <arg1>) ..... (foo <arg2>) ...))
  ; transforms essentially into
(progn ... ((lambda (n) <body of foo>) <arg1>) ...
           ((lambda (n) <body of foo>) <arg2>) ....)
```
[4]

Problems in the Semantics of Inlining

One of the first problems in the semantics of inlining is whether an inline declaration/pragma/hint should apply to the *definition* of the function, or to an *occurrence* of its use, or both. The implication in C++ and Ada is that if one defines a function to be "inline", then *every* occurrence is inlined, and the function definition itself need not appear in the object code as a normal function, since it is no longer referenced (unless its address is taken, e.g., by C's `&`). On the other hand, a programmer might wish to selectively control inlining on an occurrence-by-occurrence basis, perhaps by indicating that everywhere within the scope of an `inline` declaration the occurrences of the function will be inlined.

If one is to specify inlining on an occurrence-by-occurrence basis, then one may need to somehow warn the compiler to save the source code for a routine, in case that someone later wants to inline that routine. For example, Common Lisp requires the specification of `inline` at both the definition and the occurrence; the first `inline` tells the compiler to save the function definition, while the second `inline` tells the compiler to actually perform the substitution. The alternative is to cause the compiler to save the source code for *every* routine, which may be expensive in memory space and/or file space, although modern CASE development environments already save all source code. An inlining capability interferes with separate compilation because one may wish to inline a routine whose source code is separately compiled. In fact, one may wish to inline a routine whose source code is not available at all (at least in human-readable form)—e.g., if it is part of a library purchased from an independent vendor who wishes to protect this source code. These issues are identical to those involved in the compilation of Ada *generics*, since Ada generics are almost universally treated as compile-time macros; Ada systems solve this problem by keeping pre-parsed or raw source text (possibly encrypted) for all routines in a *library*.

Inlining also interferes with *overloading*, in that it is difficult to specify which version of an overloaded function is meant using only the function name by itself. Either the current Ada-83 rule must be used, which specifies that *every* overloading of the function name must be inlined when a pragma `inline` specifies the function name, or else one must specialize the function with a parameter/result *signature* to disambiguate which version of the overloaded function is intended to be inlined.

Both C++ and Ada have the capability of accessing the *address* of a function which is given by name.[5] Therefore, besides the problem of disambiguating an overloaded name, we now have the additional problem that such an address may not exist if every occurrence of the function has already been inlined.

Contrary to the standard presumption, inlining in Common Lisp is *not* a semantically transparent activity, because a user can easily detect whether a function has been inlined by modifying the "function cell" of the symbol denoting the function. A Common Lisp `inline` declaration, however, gives the compiler *permission* to assume that the function cell of the symbol will be constant throughout the execution of the program. As a result of this semantic

---

[3]Common Lisp now provides for "compiler macros" [Steele90,8.4], whose functionality is similar to that of compiler optimizers.

[4]Of course, this transformation is only a schematic; lexical variables bound between the definition and occurrence of `foo` have to be renamed to avoid 'capturing' the free lexical variables of `foo`, and dynamic variables may require additional processing. `unwind-protect` also introduces additional complications [Baker92MC].

[5]An Ada-83 program can't actually *do* anything with the address of a function/procedure, however, because Ada-83 has no function/procedure datatype and no `apply` or `funcall` mechanism to call this procedure by its address.

Copyright © 1992 by Nimble Computer Corporation

subtlety, a Common Lisp compiler is cannot simply ignore `inline`/`notinline` declarations, because these declarations have semantically significant consequences.[6]

The inlining of a recursive function is not well-defined, since it either produces an infinite amount of program text, or it is inlined to only one (or perhaps zero) levels. Mutually dependent and mutually recursive functions present additional problems, since it may not be immediately obvious whether the functions are mutually recursive, and even if they are not mutually recursive, the order of processing by the compiler may change the meaning of which occurrences are inlined.

Given a recursive function, there may be reasons for inlining the function to more than one level of depth. This process is analogous to the "loop-unrolling" that is performed in many optimizing compilers for pipelined and vector architectures [Ellis86], where it is entirely reasonable to unroll a loop 4 or 8 times. In fact, if one views "tail-recursion" as isomorphic to looping, then inlining to a depth of n is equivalent to unrolling the corresponding loop n times. While loop unrolling is reasonably well-defined, the semantics of inlining a recursive subroutine to depth n is typically not well-defined.

Semantical Problems of Inlining

Before giving a semantics for inlining, we must first clean up a few minor problems.

We view the deletion from the object module of inlined procedures which are no longer referenced as an optional optimization which is more appropriately handled by a linking loader which notices that there are no references to the procedure, and hence the procedure need not appear in the executable image.[7] We therefore assume that the compiler will compile every procedure into a normal closed subroutine—even those whose occurrences are inlined. By deferring the optimization of eliminating unreferenced procedures to link time, we eliminate the problem in C++ and Ada of trying to take the address of a nonexistent procedure. The mere existence of a reference to the procedure will cause the linker to include the procedure in the executable image, where its address can be taken without an error.[8]

We further assume that the compiler and/or development library keeps a copy of the source code for each function, so that the non-availability of source code will not be an excuse for refusing to inline a procedure.[9] While this may require that a (possibly encrypted) version of every library routine be available to the compiler, the mechanics of this process are essentially orthogonal to the semantics of inlining.

Inlining of Definitions

We first consider the possibility that "inline" refers to the procedure itself, rather than to its occurrences. This means that *all* occurrences of a procedure declared "inline" are to be inlined. We contend that the only consistent interpretation of this policy is that recursive procedures (including indirectly recursive procedures) expand into an infinite program text.

Consider the following sequence of definitions.

1. Declare inline procedure `a`,`b`.
2. Define procedure `a`.
3. Define procedure `b` which calls `a`.
4. Define procedure `c` which calls `b`.

There are several ways a compiler could process these definitions. The compiler could either simply attach definitions to `a` and `b`, without looking at the program text for these routines. Then, when defining `c`, the compiler would expand `b`, and then expand `a`, yielding a program text for `c` that did not reference either `a` or `b`. Alternatively, the compiler could process the definitions of the inlined procedures while defining them, in order to "speed up" later expansions of procedures like `c`. Given the particular sequence here, either method will work.

---

[6]Common Lisp would have more consistent semantics if it provided for a *constant* `symbol-function` binding; such a declaration is implicit in [Baker93ER].

[7]Many compiler/linker systems already support this optimization for more prosaic purposes. This optimization is not semantically transparent in Lisp, however, because `eval` could attempt to reference the function at any time, and the optimization must therefore must be under the control of the programmer.

[8]We have run into more than one compiler system whose scheme for deleting unreferenced subroutines is flawed. By placing the responsibility for this activity on the linker, the optimization need be debugged but once.

[9]Even without the overheads of complex code management systems, many Lisp systems already have the capability to do "meta-point", which retrieves the ASCII source text for a given function into an editor window.

Consider, however, the following revised sequence;

1. Declare inline procedure `a`,`b`.
2. Define procedure `b` which calls `a`.
3. Define procedure `a`.
4. Define procedure `c` which calls `b`.

In this case, the scheme which simply binds the unprocessed text for `a` and `b` would still work properly, while the scheme which attempted to expand the body of `b` during its definition would still not have access to the body of `a`, so that a call to `a` would still remain unexpanded in the body of `c` after `b` has been inlined.

The following order is pessimal:

1. Declare inline procedure `a`,`b`.
2. Define procedure `c` which calls `b`.
3. Define procedure `b` which calls `a`.
4. Define procedure `a`.

Now, neither of the inline bodies are available early enough during the processing to be actually inlined, so the body of `c` contains unexpanded calls to `b` which contains an unexpanded call to `a`.

We can achieve a consistent interpretation, however, by using the following policy. We will make two passes through the program. The first pass will be used to perform inlining, while the second pass will actually compile the code. During the first pass, we initialize every inline procedure definition with a *stub*. Then, every inlined procedure *occurrence* will be replaced by a pointer to its definition, whether the real definition is known yet or not. The definition of a procedure is a text indicating an anonymous procedure with its parameter and result profile, and its body. Thus, in the most recent example, the occurrence of `b` inside the body of `c` will be replaced by a pointer to the definition of `b`, which will still be a stub, since `b` has not yet been seen. However, after both `b` and `a` have been processed, then the stubs will have been filled in by real program text. During the second pass, the compiler will then the *apply* the explicit anonymous procedure to the given arguments (a process call "beta reduction") to complete the inlining. (There are technical issues regarding the renaming of certain local "bound variables" in order to avoid name clashes during these processes. We do not minimize these problems, but focus in this paper only on the interpretation of inlined function names.)

This policy produces a consistent interpretation for all of the inlined procedures—regardless of the order of their definitions. However, this interpretation means that inlined recursive procedures will result in an infinite amount of program text, because we will have introduced loops into the program text. These loops will occur even in the case of mutually recursive procedures, which are less obviously recursive. During the second pass, the compiler will run out of space due to the attempt to create an infinitely long program. (This error can be more easily and cheaply diagnosed by an intermediate pass which detects such loops in the program text.)

We believe that this circular interpretation is forced by tying the concept of "inlining" to the *definition* of a procedure rather than to its *occurrence*. The only consistent way to inline *every* occurrence is to eliminate every occurrence through substitution and thereby produce cycles in the program text![10]

Inlining of Occurrences

By inlining *occurrences* rather than *definitions*, the programmer is given more flexibility to control inlining, and we can consider the possibility of inlining certain occurrences of functions which are recursive. We now need to discuss the issue of the *scope* of the inline declaration itself, however, since it no longer coincides with the scope of the definition of the function. In particular, we can have a situation where a `notinline` declaration may *shadow* an `inline` declaration, and vice versa.

Suppose that we make the scope of the `inline` declaration similar to that of a new identifier. More precisely, suppose that we define a new function name whose definition happens to coincide with the outer function definition, except that the new function is guaranteed to be inlined. Then within the scope of this new identifier, all occurrences of this function will be inlined. Similarly, we can have a `notinline` declaration which will cause any occurrences within its scope to have an out-of-line calling sequence.

These semantics can be partially achieved in Common Lisp through the use of `macrolet` capability for defining lexically-scoped local macros. The `inline` version of the program `foo` is achieved through a local macro which expands into the text of the function, while the `notinline` version of the program is obtained by a different macro which expands into the calling sequence `(funcall #'foo <args>)`. We have only a partial solution, however, because any bound variables between the original definition of the function and the occurrence of its name must be "alpha-renamed" in order to avoid clashes with the free variables of the function being inlined.

---

[10]Program text with cycles is typical for machine language and microprogramming; goto's create these cycles. *Combinator reduction machines* [Turner79] typically introduce cycles into the program texts for recursive programs.

Given these semantics, the default for a new procedure definition must be `notinline`, or else we could never define a recursive procedure. What are the semantics when a recursive procedure declares itself to be `inline` within its own body? Since the interpretation which causes circular program text is not very useful, we consider what the other alternatives might be. A useful interpretation is the one which allows every identifier currently within the scope of the declaration to be replaced by *copies* of the body, but any identifiers within this copy itself are not replaced. In other words, the recursive procedure is to be inlined only *one level*.

How can this inlining be mechanized so that only the appropriate identifiers are replaced with program text? In other words, how does one avoid the "name capture" problem that one can get by bringing new occurrences of the function name itself inside the scope of the `inline` declaration?

Within the recursive definition of the normal (notinline) function `f`, the occurrences of `f` do *not* mean the text of `f` itself, but refer instead to a calling sequence for the out-of-line subroutine `f`. Therefore, when the compiler sees these occurrences of `f`, it does not go into an infinite recursion.

If one chooses the interpretation that the `inline` declaration introduces a new function name with the same spelling as the outer function name, then every lexical occurrence of that name will be replaced by a copy of the text of the function. Within the text of the function is the `inline` declaration itself, which will get processed again, however, and again cause an infinite expansion of the text. Of course, if there are no occurrences outside of its definition, then this subprogram will be dropped by the linking loader, even after consuming an infinite amount of time and space during compilation!

Constraints

So far, we have exhibited only problems, not solutions. Before we sketch a solution to these problems, we list the constraints we wish to place on any potential solution.

1. *The programmer knows what he/she is doing*. In other words, `inline` and `notinline` declarations are not *hints*, but *commands*. Our semantics try to provide a finite interpretation for these commands, but whenever such an interpretation is not possible, the compiler should complain, rather than quietly ignoring an `inline` command.

2. The sequence of a non-recursive function definition followed by an `inline` declaration, followed by an occurrence of the function name should cause the name to be replaced by the function definition.

3. The sequence of a recursive function definition followed by an `inline` declaration, followed by an occurrence of the function name should cause the name to be replaced by a finite program text which is intuitively a "one-level" expansion of the function.

4. The declaration `(inline-k foo)`, where `k` is a non-negative constant integer, should have the effect of unrolling a tail-recursive (iterative) function `foo` exactly `k` times.

Hint: the lambda-calculus Y operator

The *lambda calculus* [Barendregt84] provides an interesting insight into the problem of inlining functions. The lambda calculus does not explicitly provide for recursive functions, but they can be readily defined using the `Y` operator [Gabriel88]. The `Y` operator operates on the "kernel" of a function definition to produce a full-fledged function in which internal references to the function itself have been properly bound to the function itself. The lambda calculus is completely functional, however, and so it cannot use the means of achieving recursive functions found in traditional programming language implementations—imperative assignment to create a circular structure in the compiled text. Rather, the lambda calculus `Y` operator expands (inlines) the function *lazily* as it is needed, so that it is expanded only to the depth of recursion actually required, rather than infinitely.

Solution #1

The hint from the lambda-calculus tells us that we cannot hope to solve our problem unless we can break the recursive cycles at compile-time and focus on non-recursive substitutions. In Common Lisp terminology, this means that we must somehow translate the `labels` special forms which produce cyclic binding environments into `flet` or `macrolet` special forms which produce non-cyclic binding environments. The translation below achieves this by moving the cycles from compile-time to run-time:

```
(defmacro labels (fns &body forms)
  (let ((nfnames (mapcar #'(lambda (ignore) (gensym)) fns)))
    `(let ,nfnames
       (macrolet ,(mapcar #'(lambda (f nf &aux (fn (car f)))
                              `(,fn (&rest a) (list* 'funcall ',nf a)))
                          fns nfnames)
         (setq ,@(mapcan #'(lambda (f nf) `(,nf #'(lambda ,@(cdr f))))
                         fns nfnames))
         ,@forms))))
```

Copyright © 1992 by Nimble Computer Corporation

Since this definition is inscrutable, we show its effect on a recursive program for the factorial function:

```
(labels ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
  (fact 10)) =>
```

```
(let (fact-gensym)
  (macrolet ((fact (&rest a) (list* 'funcall 'fact-gensym a)))
    (setq fact-gensym #'(lambda (n) (if (zerop n) 1 (* n (fact (1- n))))))
    (fact 10)))
```

The ordering of the forms in the above emulation of `labels` by `let`, `macrolet` and `setq` is extremely important. The run-time lexical variable for holding the recursive function definition must first be defined without a value; `macrolet` is then used to define a local lexical macro which simply expands into an out-of-line call to the function stored in that variable; finally, the cycle is completed at run-time by an assignment of the function closure to the run-time lexical variable. Notice that the closure must be defined inside the `macrolet`, since the closure has occurrences of the function name (`fact`) which reference the function defined by the `macrolet`. Since `macrolet` macros do not appear as recursive functions to the compiler, the meaning of inlining for the function stub is well-defined.

We must now modify this definition of `labels` to handle the possibility of an `inline` or `notinline` declaration within any of the recursively defined functions, or within the body of the `labels` itself. We do this by defining two versions for each `labels`-defined function—both an in-line and an out-of-line version. Ignoring for the moment the possibility of name clashes, we define for each `labels` function `foo` both a `foo-inline` and a `foo-notinline` version. We then choose which version is to be specified by the simple name `foo` by surrounding the appropriate text with a `macrolet` which defines `foo` in terms of either `foo-inline` or `foo-notinline`. In other words, we utilize a `macrolet` to rename `foo` for us as either `foo-inline` or `foo-notinline`.

```
(defun inline-name (name) (intern (format nil "~A-INLINE" name)))

(defun notinline-name (name) (intern (format nil "~A-NOTINLINE" name)))

(defmacro flet-inline (fns &body forms)
  `(flet ,fns (declare (inline ,@(mapcar #'car fns))) ,@forms))

(defmacro frename (pairs &body forms)
  ;;; Rename local lexical functions using macrolet.[11]
  `(macrolet
      ,(mapcar #'(lambda (pair &aux (nfn (car pair)) (fn (cadr pair)))
                  `(,nfn (&rest a) (cons ',fn a)))
               pairs)
      ,@forms))

(defmacro labels (fns &body forms)
  (let* ((inlines (inline-decls forms))       ;retrieve list of fns to inline.
         (notinlines (notinline-decls forms)) ;list to not to inline.
         (forms (strip-decls forms))          ;body without declarations.
         (nfnms (mapcar #'(lambda (ignore) (gensym)) fns)))
    `(let ,nfnms
       (macrolet
        ,(mapcar #'(lambda (f nf)
                     `(,(car f) (&rest a) (list* 'funcall ',nf a)))
                 fns nfnms)
        (frename
         ,(mapcar #'(lambda (f) `(,(notinline-name (car f)) ,(car f))) fns)
         (flet-inline
           ,(mapcar #'(lambda (f) `(,(inline-name (car f)) ,@(cdr f))) fns)
           (frename
             (,@(mapcar #'(lambda (f) `(,f ,(inline-name f))) inlines)
              ,@(mapcar #'(lambda (f) `(,f ,(notinline-name f))) notinlines))
             (setq ,@(mapcan #'(lambda (nf f) `(,nf #',(car f))) nfnms fns))
             ,@forms))))))
```

[11]We could have used `flet-inline` to define `frename`, but `macrolet` is more succinct.

Copyright © 1992 by Nimble Computer Corporation

```
(defmacro locally-inline (fns &body forms)
  ;;; Inline all functions in 'fns' within the body 'forms'.
  `(frename ,(mapcar #'(lambda (f) `(,f ,(inline-name f))) fns)
     ,@forms))
(defmacro locally-notinline (fns &body forms)
  ;;; Call all functions in 'fns' within the body 'forms'.
  `(frename ,(mapcar #'(lambda (f) `(,f ,(notinline f))) fns)
     ,@forms))
```

Once again, the expansion of this form is easier to understand when applied to our factorial example:

```
(labels ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
  (declare (inline fact))
  (fact 10)) =>
(let (fact-gensym)
  (macrolet ((fact (&rest a) (list* 'funcall 'fact-gensym a)))
   (frename ((fact-notinline fact))
    (flet-inline ((fact-inline (n) (if (zerop n) 1 (* n (fact (1- n))))))
     (frename ((fact fact-inline))
       (setq fact-gensym #'fact)
       (fact 10))))))
```

If we look carefully at the above expansion for `fact`, we see that both the `setq` and the body of the original `labels` occur within the same binding environment. In particular, the name '`fact`' in this environment means the same thing for occurrences within the original closure bodies, as well as within the original `labels` body; but this similar meaning is required by the Common Lisp language definition [Steele90,p.155]. We also see that occurrences of '`fact`' within the body of `fact-inline` refer to the closed version—i.e., essentially `fact-notinline`. Because our definition of `labels` breaks all circular environments, our compiler is guaranteed not to expand into infinite program text, although the text size can in the worst case blow up exponentially.

Solution #k

Our scheme for inlining has achieved our first goal—an interpretation with a finite program text which also provides for "one-level" in-line substitution. Unfortunately, this solution does not scale well, because if we want to inline a function to depth *k*, we must predefine a *k*-level expansion which can then be selected when the occurrence is encountered. In order to avoid pre-defining `k`-level expansions for every function and every `k`, we must then examine the text bodies to see whether any `k`-level expansions "occur free" within those bodies. In other words, we find it is too expensive to predefine every version of every function, without first determining if it will ever be called. Unfortunately, the actual code for this solution is too large to include here, so we will show only its expansion for our factorial example.

```
(labels ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
  (declare (inline-3 fact))
  (fact 10)) =>
(let (fact-gensym)
 (macrolet ((fact (&rest a) (list* 'funcall 'fact-gensym a)))
  (frename ((fact-notinline fact))
   (flet-inline ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
    (flet-inline ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
     (flet-inline ((fact (n) (if (zerop n) 1 (* n (fact (1- n))))))
      (frename ((fact-inline fact))
       (frename ((fact fact-inline))
         (setq fact-gensym #'fact)
         (fact 10)))))))))
```

In the above expansion for `fact`, it is important to notice that each occurrence of '`fact`' is bound at the next higher level, and thus we achieve our depth-3 inlining for this function.

We must point out that the above scheme depends upon the generation of different spellings of the function name to select the different implementations. Although this scheme is easy to understand, it is not *safe*, in the sense that the names generated by the functions `inline-name` and `notinline-name` could clash with programmer-generated names. A safer scheme would utilize a more sophisticated scheme for generating names which can not possibly clash with programmer-generated names.

Conclusions

We have shown how a precise semantics can be given for the process of subprogram integration, commonly called "inlining". Furthermore, these semantics can provide for a finite interpretation even in the presence of recursive subroutines. Our solution to the inlining of recursive functions is based on a finite-depth analogue to the `Y` operator of the lambda-calculus, which provides for recursion without requiring loops in the program text.

References

Ada83. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983. U.S. Gov't Printing Office, 1983.

Allen, F.E., *et al*. "The Experimental Compiling System". *IBM J. Res. & Devel. 24* (1980),695-715.

Allen, Randy, and Johnson, Steve. "Compiling C for Vectorization, Parallelization, and Inline Expansion". *ACM PLDI'88, also Sigplan Not. 23*,7 (July 1988),241-249.

Baker, H.G. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, H.G. "Metacircular Semantics for Common Lisp Special Forms". ACM *Lisp Pointers V*,4 (Oct-Dec 1992), 11-20.

Ball, J. Eugene. "Predicting the Effects of Optimization on a Procedure Body". *Proc.Sigplan '79 Symp. on Compiler Constr., also Sigplan Not. 14*,8 (Aug. 1979),214-220.

Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, New York, 1984.

Callahan, David, *et al*. "Interprocedural Constant Propagation". *Proc. Sigplan '86 Symp. on Compiler Constr., also Sigplan Not. 21*,7 (July 1986),152-161.

Ellis, John R. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1986, 320p.

Ershov. A.P. On the essence of translation. In Neuhold, E.J., *ed*. *Formal Description of Programming Concepts*. North-Holland, Amsterdam, 1977,391-418.

Gabriel, Richard P. "The Why of Y". *Lisp Pointers 2*,2 (Oct./Dec. 1988),15-25.

Harrison, William. "A new strategy for code generation—the general purpose optimizing compiler". *ACM POPL 4* (1977),29-37.

Hwu, Wen-mei W., and Chang, Pohua P. "Inline Function Expansion for Compiling C Programs". *Sigplan PLDI'89, also Sigplan Not. 24*,7 (July 1989),246-255.

Hyun, Kio C., and Doberkat, Ernst-Erich. "Inline Expansion of SETL Procedures". *Sigplan Not. 20*,12 (Dec. 1985),33-38.

MacLaren, M. Donald. "Inline Routines in VAXELN Pascal". *Proc. Sigplan '84 Symp. on Compiler Constr., also Sigplan Not. 19*,6 (June 1984),266-275.

Scheifler, Robert W. "An Analysis of Inline Substitution for a Structured Programming Language". *CACM 20*,9 (Sept. 1977),647-654.

Steele, G.L. *Common Lisp, the Language: Second Edition*. Digital Press, Bedford, MA, 1990.

Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

Turchin, Valentin F. "The Concept of a Supercompiler". ACM TOPLAS 8,3 (July 1986),292-325.

Turner, D. "A New Implementation Technique for Applicative Languages". *SW—Pract.&Exper. 9* (1979),31-49.

Copyright © 1992 by Nimble Computer Corporation