# Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages *

Koen De Bosschere,† Saumya Debray,‡ David Gudeman,‡ Sampath Kannan‡

† Department of Electronics
Universiteit Gent
B-9000 Gent, Belgium

‡ Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

## Abstract

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is straightforward: find an ordering for the "entry actions" of a procedure, and generate multiple entry points for the procedure, so as to maximize the savings realized from different call sites bypassing different sets of entry actions. We show that the problem of computing optimal solutions to arbitrary call forwarding problems is NP-complete, and describe an efficient greedy algorithm for the problem. Experimental results indicate that (*i*) this algorithm is effective, in that the solutions produced are generally close to optimal; and (*ii*) the resulting optimization leads to significant performance improvements for a number of benchmarks tested.

## 1 Introduction

The code generated for a function or procedure in a dynamically typed language typically has to carry out various type and range checks on its arguments before it can operate on them. These runtime tests can incur a significant performance overhead. As a very simple example, consider the following function to compute the average of a list of numbers:

```
ave(L, Sum, Count) =
   if null(L) then Sum/Count
   else ave(tail(L),Sum+head(L),Count+1)
```

In a straightforward implementation of this function, the code generated checks the type of each of its arguments each time around the loop: the first argument must be a (empty or non-empty) list, while the second and third arguments must be numbers.[1] Notice, however, that some of this type checking is unnecessary: the expression `Sum+head(L)` evaluates correctly only if `Sum` is a number, in which case its value is also a number; similarly, `Count+1` evaluates correctly only if `Count` is a number, and in that case it also evaluates to a number. Thus, once the types of `Sum` and `Count` have been checked at the entry to the loop, further type checks on the second and third arguments are not necessary.

The function in this example is tail recursive, making it easy to recognize the iterative nature of its computation and use some form of invariant code motion to move the type check out of the loop. In general, however, such redundant actions may be encountered where the definitions are not tail recursive and where the loop structure is not as easy to recognize. An alternative approach, which works in general, is to generate multiple entry points for the function `ave`, so that a particular call site can enter at the "appropriate" entry point, bypassing any code it does not need to execute. In the example above, this would give exactly the desired result: tail call optimization would compile the recursive call to `ave` into a jump instruction, and noticing that the recursive call does not need to test the types of its second and third arguments, the target of this jump

*K. De Bosschere was supported by the National Fund for Scientific Research of Belgium and by the Belgian National Incentive Program for fundamental research in Artificial Intelligence. S. Debray and D. Gudeman were supported in part by the National Science Foundation under grant number CCR-9123520. S. Kannan was supported in part by the National Science Foundation under grant number CCR-9108969.

Copyright 1994 ACM. Appeared in the Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994, pp. 409–420.

[1] In reality, the generated code would distinguish between the numeric types `int` and `float`, e.g., using "message splitting" techniques as in [5, 6]—the distinction is not important here, and we assume a single numeric type for simplicity of exposition.

would be chosen to bypass these tests.

However, notice that in the example above, even if we generate multiple entry points for `ave`, the optimization works *only if the tests are generated in the right order*: since it is necessary to test the type of the first argument each time around the loop, the tests on the second and third arguments cannot be bypassed if the type test on the first argument precedes those on the other two arguments. As this example illustrates, the order in which the tests are generated influences the amount of unnecessary code that can be bypassed at runtime, and therefore the performance of the program.

In general, functions and procedures in dynamically typed languages contain a set of (idempotent) "entry actions," such as type tests, initialization actions (especially for variadic procedures), etc., that are executed at entry to the procedure. Moreover, these actions can typically be carried out in any of a number of different "legal" orders (in general, not all orderings of entry actions may be legal, since some actions may depend on the outcomes of others—for example, the type of an expression `head(x)` cannot be checked until `x` has been verified to be of type list). The code generated for a procedure therefore consists of a set of entry actions in some order, followed by code for its body. There are a number of different call sites for each procedure, and at each call site we have some information about the actual parameters at that call site, allowing that call to skip some of these entry actions. Moreover, each call site has a different execution frequency (estimated, for example, from profile information or from the structure of the call graph). In general, different call sites have different information available about their actual parameters, so that an order for the entry actions of a procedure that is good for one call site, in terms of the number of unnecessary entry actions that can be skipped, may not be as good for another call site. A good compiler should therefore attempt to find an ordering on the entry actions that maximizes the benefits, over all call sites, due to bypassing unnecessary code. We refer to determining such an order for the entry actions and then "forwarding" the branch instructions at different call sites so as to bypass unnecessary code as "call forwarding."

While many systems compile functions with multiple entry points, we do not know of any that attempt to order the entry actions carefully in order to exploit this to the fullest. In this paper, we address the problem of determining a "good" order for the set of tests a function or procedure has to carry out. We show that generating an optimal order is NP-complete in general, and give an efficient algorithm for selecting an ordering using a greedy heuristic. The result generalizes a number of optimizations for traditional compilers, such as jump chain collapsing and invariant code motion out of loops. Experimental results indicate that (*i*) the heuristic is good, in that the orderings it generates are usually not far from the optimal; and (*ii*) the resulting optimization is effective, in the sense that it typically leads to significant speed improvements.

The issues and optimizations discussed in this paper are primarily at the intermediate code level: for this reason, we do not make many assumptions about the source language, except that a call to a procedure typically involves executing a set of idempotent "entry actions." This covers a wide variety of dynamically typed languages, e.g., functional programming languages such as Lisp and Scheme (e.g., see [15]), logic programming languages such as Prolog [4], GHC [17] and Janus [11, 13], imperative languages such as SETL [14], and object-oriented languages such as Smalltalk [10] and SELF [6]. The optimization we discuss is likely to be most beneficial for languages and programs where procedure calls are common, and which are therefore liable to benefit significantly from reducing the cost of procedure calls. However—the title of the paper notwithstanding—the optimization is not limited, *a priori*, to dynamically typed languages: it is also applicable, in principle, to idempotent entry actions, such as initialization and array bound checks, in statically typed languages, and some optimizations used in statically typed languages, such as inverse eta-reduction/uncurrying/argument flattening in Standard ML of New Jersey [1], can also be thought of as instances of call forwarding (see Section 6).

## 2 The Call Forwarding Problem

As discussed in the previous section, the code generated for a procedure consists of a set of entry actions, which can be carried out in a number of different legal orders, followed by the code for its body. Each procedure has a number of call sites, and at each call site there is some information about the actual parameters for calls issued from that site, specifying which entry actions must be executed and which may be skipped.[2] This is modelled by associating, with each call site, a set of entry actions that must be executed by that call site. Moreover, each call site has associated with it an estimate of its execution frequency: such estimates can be obtained from profile information, or from the structure of the call graph of the program (see, for example, [3, 19]). Finally, different entry actions may require a different number of machine instructions to execute, and therefore have different "sizes."

Our objective is to order the entry actions of the procedures in a program, and redirect calls so as to by-

[2] The precise mechanism by which this information is obtained, e.g., dataflow analysis, user declarations, etc., is orthogonal to the issues discussed in this paper, and so is not addressed here.

pass unnecessary actions where possible, in such a way that the total number of instructions that are skipped, over the entire execution of the program, is as large as possible. However, it is not difficult to see that for any procedure $p$ in a program, the code to set up and execute procedure calls in the body of $p$ is separate from the entry actions of $p$. Because of this, the order of $p$'s entry actions—and therefore, the number of instructions that are skipped by calls to $p$ in an execution of the program—neither influence nor are influenced by the order of the entry actions for any other procedure in the program. The problem of maximizing the total number of instructions skipped by call forwarding for the entire program, then, reduces to the problem of maximizing, for each procedure, the number of instructions skipped by calls to that procedure. For our purposes, therefore, the *call forwarding problem* is the problem of determining a "good" order for the entry actions of a procedure so that the savings accruing from bypassing unnecessary entry actions over all call sites for that procedure, weighted by execution frequency, is as large as possible.

The problem can be generalized by allowing code to be copied from a procedure to the call sites for that procedure. As an example, suppose we have a procedure with entry actions $a$ and $b$, and two call sites: $A$, which can skip $a$ but must execute $b$; and $B$, which can skip $b$ but must execute $a$. Suppose the entry actions are generated in the order $\langle a, b\rangle$, then call site $A$ can skip $a$, but $B$ cannot skip $b$ and therefore executes unnecessary code (a symmetric problem arises if the other possible order is chosen). A solution is to copy the entry action $a$ at the call site $B$, i.e., execute the entry action at $B$ before jumping to the callee. If we allow arbitrarily many entry actions to be copied to call sites in this manner, it is trivial to generate an optimal solution to any call forwarding problem: simply copy to each call site the entry actions that call site must execute, then branch into the callee bypassing all entry actions at the callee. This obviously produces an optimal solution, since each call site executes exactly those entry actions that it must execute, and can be done efficiently in polynomial time. However, it has the problem that such unrestricted copying can lead to significant code bloat, since there may be many call sites for a procedure, each of them getting a copy of most of the entry actions for that procedure (we have observed this phenomenon in a number of application programs).

The best solution to this problem is to impose a global bound on the total number of entry actions that may be copied, across all the call sites occurring in a program, but this turns out to be complicated to implement because when performing call forwarding on any particular procedure, we have to keep track of the number of entry actions copied for all the procedures in the program, including those that have not yet been processed by the optimizer! A simple and effective approximation to this approach is to assign, for each procedure, a bound on the number of entry actions that can be copied to each call site for that procedure. If we start with a global bound on the total number of entry actions that can be copied, such per-procedure bounds can be obtained by "dividing up" the global bound among the procedures (possibly taking into account, for each procedure, the number of call sites for it and their execution frequencies, so that procedures with deeply nested call sites can copy more entry actions and thereby effect greater optimization). A discussion of heuristics for establishing such per-procedure bounds is beyond the scope of this abstract: we simply assume, in the discussion that follows, that for each procedure there is a bound on the number of its entry actions that can be copied to any call site.

The call forwarding problem can therefore be formulated in the abstract as follows:

**Definition 2.1** A call forwarding problem is a 5-tuple $\langle E, C, w, s, k\rangle$, where:

- $E$ is a finite set (representing the entry actions of the procedure concerned);
- $C$ is a multiset of subsets of $E$ (representing the entry actions that each call site must execute);
- $w : C \longrightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers, is a function that maps each call site to its "weight", i.e., execution frequency;
- $s : E \longrightarrow \mathcal{N}$ represents the "size" of each element of $E$ (representing the number of machine instructions needed to realize the corresponding entry action); and
- $k \geq 0$ represents a bound on the number of entry actions that can be copied to call sites.

∎

A *solution* to a call forwarding problem $\langle E, C, w, s, k\rangle$ is a permutation $\pi$ of $E$, i.e., a 1-1 function $\pi : E \longrightarrow \{1, \ldots, |E|\}$. The cost of a solution $\pi$ is, intuitively, the total number of machine instructions executed, over all call sites, given that the entry actions are generated in the order $\pi$. Given a call forwarding problem $\langle E, C, w, s, k\rangle$, the cost of a solution $\pi$ for it is defined as follows. First, let $copied(c, \pi, i)$ denote (the indices of) those entry actions in $\pi$ that have to be copied to a call site $c$ if the entry point for $c$ is to bypass the first $i$ elements of $\pi$:

$$copied(c, \pi, i) = \{j \mid j \leq i \ \wedge \ \pi^{-1}(j) \in c\}.$$

Here, $\pi^{-1}(j)$ denotes the element of $E$ that is the $j^{th}$ element of the permutation $\pi$. For any call site $c \in C$, given the bound $k$ on the number of actions that can be copied to $c$, the maximum number of entry actions that can be skipped by $c$—either because $c$ does not have to execute that action, or because it has been copied from the callee to the call site—is given by

$$Skip(c, \pi) = \max\{i \ : \ |copied(c, \pi, i)| \ \leq \ k\}.$$

The cost of a solution $\pi$ can then be expressed as the weighted sum, over all call sites, of (the sizes of) the instructions that cannot be skipped by the call sites:

$$cost(\pi) = \sum_{c \in C}\{w(c) \cdot s(I) \ | \ I \in E \ \wedge \ \pi(I) > Skip(c, \pi)\}.$$

## 3 Algorithmic Issues

We first consider the complexity of determining optimal solutions to call forwarding problems. The following result shows that the existence of efficient algorithms for this is unlikely:

**Theorem 3.1** *The determination of an optimal solution to a call forwarding problem is NP-complete. It remains NP-complete even if all entry actions have equal size.*

**Proof** By reduction from the Optimal Linear Arrangement problem, which is known to be NP-complete [8, 9]. See the Appendix for details. ∎

This result might very well be of only academic interest if the number of entry actions encountered in typical programs could be guaranteed to be small. However, our experience has been that this is not the case in many actual applications. The reason for this is that, even if the number of arguments to procedures is small for most programs encountered in practice, it is not unusual to have a number of entry actions associated with a single argument (e.g., see Section 4), involving type and range checks, pattern matching and indexing code, pointer chain dereferencing (a common operation in logic programming languages), and so on. Because of this, the total number of entry actions in a procedure can be quite large, making exhaustive search for an optimal solution impractical. We therefore seek efficient polynomial time heuristics for call forwarding that produce good solutions for common cases.

### 3.1 A Greedy Algorithm

While the problem of computing optimal solutions for arbitrary call forwarding problems is NP-complete in general, a greedy algorithm appears to work quite well in practice (see Table 1). Given a call forwarding problem for a procedure with a bound of $k$ on the number of actions that can be copied from the callee to the call sites, the general idea is to pick actions one at a time, at each step choosing an action that minimizes the cost to be paid at that step. The algorithm maintains a list of call sites that do not need to execute more than $k$ of the actions chosen upto that point, and therefore can still have some actions copied to them—such call sites are said to be *active*. Each active call site $c$ has associated with it a counter, denoted by $count[c]$ in Figure 1, that keeps track of how many more actions can be copied to that call site. The weight of an action, at any point in the algorithm, is computed as the sum of the weights of the active call sites that need to execute that action, divided by the "size" of that action (recall that the size of an action represents the number of machine instructions needed to implement it)—thus, everything else being equal, an action that is more expensive in terms of the number of machine instructions it requires will have a smaller weight than one with smaller size, and hence be picked earlier, thereby allowing more call sites to bypass it. Since in general there may be dependencies between instructions that restrict the set of legal orderings (e.g., see the example in Section 4), the algorithm first constructs a dependency graph whose nodes are the entry actions under consideration, and where there is an edge from a node $e_1$ to a node $e_2$ if $e_1$ must precede $e_2$ in any legal execution; the set of predecessors of a node $x$ in this graph is denoted by $preds(x)$. The algorithm is simple: it repeatedly picks an "available" action (i.e., an action whose predecessors in the dependency graph $G$ have already been picked) of least weight, then updates the counters of the appropriate call sites as well as the list of active call sites, deleting from this list any call site that has reached its limit of the number of actions that can be copied from the callee. This process continues until all actions have been enumerated. The algorithm is described in Figure 1.

## 4 An Example

In this section we consider in more detail the `ave` function from Section 1 to see the effect of call forwarding on the code generated. To illustrate the fact that this optimization is not limited to code for type checking, we consider here a realization of this function in Prolog. As in other logic programming languages, unification between variables in Prolog can set up chains of pointers, and loading the value of a variable requires dereferencing such chains. A number of authors have shown that significant performance improvements are possible if the lengths of these pointer chains can be predicted via compile-time analysis, so that unnecessary dereferencing code can be deleted [7, 12, 16]; however, the analyses involved are fairly complex. Here we show how, in many cases, unnecessary dereference operations can be eliminated using call forwarding. The procedure is defined as follows:

```
ave([], Sum, Count, Avg) :-
   Avg is Sum/Count.
ave([H|L], Sum, Count, Avg) :-
   Sum1 is Sum+H, Count1 is Count+1,
   ave(L, Sum1, Count1, Avg).
```

Assume that, as in many modern Lisp and Prolog implementations, parameters are passed in (virtual machine) registers, so that the first parameter is in register `Arg1`, the second parameter in register `Arg2`, and so on. Figure 2(a) gives the intermediate code that might be generated in a straightforward way. (In reality, the generated code would distinguish between the numeric types `int` and `float`, e.g., using "message splitting" techniques as in [5, 6]—the distinction is not important here, and we assume a single numeric type for simplicity of exposition.) The first six instructions of `ave` are entry actions that can be executed in any order where the dereferencing of a register precedes its use. Moreover, at the (recursive) call site for `ave`, we know from the semantics of the `add` instruction that `Arg1` and `Arg2` are both numbers, and that there is no need for either dereferencing or type checking of these registers. The entry actions corresponding to dereferencing and type checking of these registers can therefore be bypassed by the recursive call site. Assume that apart from the recursive call, there is another call site (the "initial" call) for the procedure `ave`. For notational brevity in the discussion that follows, denote the instructions above as follows:

| | | |
|---|---|---|
| `Arg1 := deref(Arg1)` | $\mapsto$ | $a$ |
| `Arg2 := deref(Arg2)` | $\mapsto$ | $b$ |
| `Arg3 := deref(Arg3)` | $\mapsto$ | $c$ |
| `if ¬List(Arg1) goto Err` | $\mapsto$ | $d$ |
| `if ¬Number(Arg2) goto Err` | $\mapsto$ | $e$ |
| `if ¬Number(Arg3) goto Err` | $\mapsto$ | $f$ |

Finally, assume that no copying of code to call sites is allowed. Then, we can formulate this as a call forwarding problem $\langle E, C, w, s, k\rangle$ as follows:

- $E = \{a, b, c, d, e, f\}$;
- $C = \{c_1, c_2\}$, where $c_1 = \{a, b, c, d, e, f\}$ is the initial call site, and $c_2 = \{a, d\}$ is the recursive call site;
- $w = \{c_1 \mapsto 1, c_2 \mapsto 10\}$, i.e., we assume that loops iterate about 10 times on the average;
- the "size function" $s$ maps each entry action in $E$ to 1 (for simplicity); and
- $k = 0$, i.e., no copying of code to call sites is allowed.

Initially, the set of available actions is $\{a, b, c\}$, and both call sites are active, so the weights computed for these actions are: $a$ : 11; $b$ : 1; $c$ : 1. There are two actions, $b$ and $c$, that have lowest weight, and one of them—say, $b$—is picked by the algorithm. As a result, the call site $c_1$ becomes inactive. The set of available actions at this point is $\{a, c, e\}$, with weights 10, 0, 0 respectively. There are two actions, $c$ and $e$, with lowest weight, and one of them—say, $c$—is picked. The algorithm proceeds in this manner, eventually producing the sequence $\langle b, c, e, f, a, d\rangle$ as a solution to this call forwarding problem. In other words, call forwarding orders the entry actions so that the dereferencing and type tests on `Arg2` and `Arg3` come first, and can be skipped by the recursive call to `ave`. The resulting code is shown in Figure 2(b). Notice that the code for dereferencing and type checking the second and third arguments have effectively been "hoisted" out of the loop. Moreover, this has been accomplished, not by recognizing and dealing with loops in some special way, but simply by using the information available at call sites. It is applicable, therefore, even to computations that are not iterative (i.e., tail recursive), including procedures that involve arbitrary linear, nonlinear, and mutual recursion.

## 5 Experimental Results

We ran experiments on a number of small benchmarks to gauge (*i*) the efficacy of greedy algorithm, i.e., the quality of its solutions compared to the optimal; and (*ii*) the efficacy of the optimization, i.e., the performance improvements resulting from it. The numbers presented reflect the performance of `jc` [11], an implementation of a logic programming language called Janus [13] on a Sparcstation-1.[3] This system is currently available by anonymous FTP from `cs.arizona.edu`.

Table 1 gives, for each benchmark, the number of machine instructions that would be executed over all call sites *for the entry actions in the procedures only*, using (*i*) no call forwarding; (*ii*) call forwarding using the greedy algorithm; and (*iii*) optimal call forwarding. The weights for the call sites were estimated using the structure of the call graph: we assumed that on the average, each loop iterates about 10 times, and the branches of a conditional are taken with equal frequency. While the optimizations were carried out at the intermediate code level, we used counts of the number of Sparc assembly instructions for each intermediate code instruction, together with the execution frequencies estimated from the call graph structure, to estimate the runtime cost

[3] Our implementation uses a variant of call forwarding where entry actions are copied from the callee to the call sites as long as this will allow a later action to be skipped.

of the different solutions. The results indicate that the greedy heuristic has uniformly good performance: on the benchmarks, it attains the optimal solution in each case.

Table 2 gives the improvements in speed resulting from our optimizations, and serves to evaluate the efficacy of call forwarding. The time reported for each benchmark, in milliseconds, is the time taken to execute the program once. This time was obtained by iterating the program long enough to eliminate most effects due to multiprogramming and clock granularity, then dividing the total time taken by the number of iterations. The experiments were repeated 20 times for each benchmark, and the average time taken in each case. Call forwarding accounts for improvements ranging from about 12% to over 45%. Most of this improvement comes from code motion out of inner loops: the vast majority of type tests etc. in a procedure appear as entry actions that are bypassed in recursive calls due to call forwarding, effectively "hoisting" such tests out of inner loops. As a result, much of the runtime overhead from dynamic type checking is optimized away.

Table 3 puts these numbers in perspective by comparing the performance of `jc` to Quintus and Sicstus Prologs, two widely used commercial Prolog systems. On comparing the performance numbers from Table 2 for `jc` before and after optimization, it can be seen that the performance of `jc` is competitive with these systems even before the application of the optimizations discussed in this paper. It is easy to take a poorly engineered system with a lot of inefficiencies and get huge performance improvements by eliminating some of these inefficiencies. The point of this table is that when evaluating the efficacy of our optimizations, we were careful to begin with a system with good performance, so as to avoid drawing overly optimistic conclusions.

Finally, Table 4 compares the performance of our Janus system with C code for some small benchmarks.[4] Again, these were run on a Sparcstation 1, with `cc` as the C compiler. The programs were written in the style one would expect of a competent C programmer: no recursion (except in `tak` and `nrev`—an $O(n^2)$ "naive reverse" program for reversing a linked list of integers—where it is hard to avoid), destructive updates, and the use of arrays rather than linked lists (except in `nrev`, which by definition traverses a list). The source code for these benchmarks is given in Appendix B. It can be seen that the performance of `jc` is not very far from that of C, attaining approximately the same performance as unoptimized C code, and being only about a factor of 2, on the average, slower than C code optimized at level `-O4`. On some benchmarks, such as `nrev`, `jc` outperforms unoptimized C and is not much slower than optimized C, even though the C program uses destructive assignment and does not allocate new cons cells, while Janus is a single assignment language where the program allocates new cons cells at each iteration—its performance can be attributed at least in part to the benefits of call forwarding.

[4]The Janus version of `qsort` used in this table is slightly different from that of Table 3: in this case there are explicit integer type tests in the program source, to be consistent with `int` declarations in the C program and allow a fair comparison between the two programs. The presence of these tests provides additional information to the `jc` compiler and allows some additional optimizations.

## 6 Related Work

The optimizations described here can be seen as generalizing some optimizations for traditional imperative languages [2]. In the special case of a (conditional or unconditional) jump whose target is a (conditional or unconditional) jump instruction, call forwarding generalizes the flow-of-control optimization that collapses chains of jump instructions. Call forwarding is able to deal with conditional jumps to conditional jumps (this turns out to be an important source of performance improvement in practice), while traditional compilers for imperative languages such as C and Fortran typically deal only with jump chains where there is at most one conditional jump (see, for example, [2], p. 556).

When we consider call forwarding for the last call in a recursive procedure, what we get is essentially a generalization of code motion out of loops, in the sense that the code that is bypassed due to call forwarding at a particular call site need not be invariant with respect to the entire loop. The point is best illustrated by an example: consider a function

```
f(x) = if x = 0 then 1
   else if p(x) then f(g(x-1))  /* 1 */
   else f( h(x-1) )             /* 2 */
```

Assume that the entry actions for this function include a test that its argument is an integer, and suppose that we know, from dataflow analysis, that `g()` returns an integer, but do not know anything about the return type of `h()`. From the conventional definition of a "loop" in a flow graph (see, for example, [2]), there is one loop in the flow graph of this function that includes both the tail recursive call sites for `f()`. Because of our lack of knowledge about the return type of `h()`, we cannot claim that "the argument to `f()` is an integer" is an invariant for the entire loop. However, using call forwarding, the integer test in the portion of the loop arising from call site 1 can be bypassed. Effectively, this moves some code out of "part of" a loop. Moreover, our algorithm implements interprocedural optimization and can deal with both direct and mutual recursion, as well as non-tail-recursive code, without having to do anything special, while traditional code motion algorithms handle only the intra-procedural case.

The idea of compiling functions with multiple entry points is not new: many Lisp systems do this, Standard ML of New Jersey and Yale Haskell generate dual entry points for functions, and Aquarius Prolog generates multiple entry points for primitive operations [18]. However, we do not know of any system that attempts to order the entry actions carefully in order to maximize the savings from bypassing entry actions.

Some optimizations used in statically typed languages can also be thought of in terms of call forwarding. For example, Standard ML of New Jersey uses a combination of three transformations—inverse eta-reduction, uncurrying, and argument flattening—to optimize functions where all of the known call sites pass tuples of the same size as arguments, but where the function may "escape," i.e., not all of call sites are known at compile time [1]. The idea is to have the known call sites pass arguments in registers instead of constructing and deconstructing tuples on the heap, while call sites that are unknown at compile time execute additional code to correctly deconstruct the tuples they pass. This optimization can be thought of in terms of call forwarding as follows: suppose that each known call site for a function constructs and passes an $n$-tuple as the argument, which is then deconstructed with $n$ `select` operations at the callee. We can copy the $n$ `select` operations from the callee to each known call site, and forward the calls to enter the callee bypassing these operations. At each of these call sites, the construction of the argument $n$-tuple followed by $n$ `select`s on it can easily be recognized as inverse operations that can be optimized to avoid having to actually build tuples on the heap. Thus, known call sites can be executed efficiently, while call sites that are not known at compile time enter at the original entry point and execute the `select` operations in the expected way. Indeed, the whole point of inverse eta-reduction is to generate two entry points for a function so that known call sites can bypass unnecessary code: call forwarding can be seen as a way of extending this idea to get more than two entry points where necessary.

Chambers and Ungar consider compile-time optimization techniques to reduce runtime type checking in dynamically typed object-oriented languages [5, 6]. Their approach uses type analysis to generate multiple copies of program fragments, in particular loop bodies, where each copy is specialized to a particular type and therefore can omit some type tests. Some of the effects of the optimization we discuss, e.g., "hoisting" type tests out of loops (see Section 4), are similar to effects achieved by the optimization of Chambers and Ungar. In general, however, it is essentially orthogonal to the work described here, in that it is concerned primarily with type inference and code specialization rather than with code ordering. Because of this, the two optimizations are complementary: even if the body of a procedure has been optimized using the techniques of Chambers and Ungar, it may contain type tests etc. at the entry, which are candidates for the optimization we discuss; conversely, the "message splitting" optimization of Chambers and Ungar can enhance the effects of call forwarding considerably.

## 7 Conclusions

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is extremely straightforward: find an ordering for the "entry actions" of a procedure such that the savings realized from different call sites bypassing different sets of entry actions, weighted by their estimated execution frequencies, is as large as possible. It turns out, however, to be quite effective for improving program performance. We show that the problem of computing optimal solutions to arbitrary call forwarding problems is NP-complete, and describe an efficient heuristic for the problems. Experimental results indicate that the solutions produced are generally optimal or close to optimal, and lead to significant performance improvements for a number of benchmarks tested. A variant of these ideas has been implemented in `jc`, a logic programming system that is available by anonymous FTP from `cs.arizona.edu`.

## References


[1] A. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

[2] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[3] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", *Proc. 19th. ACM Symp. on Principles of Programming Languages*, Albuquerque, NM, Jan. 1992, pp. 59–70.

[4] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

[5] C. Chambers and D. Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically Typed Object-Oriented Programs", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 150–164. *SIGPLAN Notices* vol. 25 no. 6.

[6] C. Chambers, D. Ungar and E. Lee, "An Efficient Implementation of SELF, A Dynamically Typed

Object-Oriented Language Based on Prototypes", *Proc. OOPSLA '89*, New Orleans, LA, 1989, pp. 49–70.

[7] S. K. Debray, "A Simple Code Improvement Scheme for Prolog", *J. Logic Programming*, vol. 13 no. 1, May 1992, pp. 57-88.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[9] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete Graph Problems", *Theoretical Computer Science* vol. 1, pp. 237–267, 1976.

[10] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[11] D. Gudeman, K. De Bosschere, and S. K. Debray, "`jc` : An Efficient and Portable Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992. MIT Press.

[12] A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe, "The Impact of Abstract Interpretation: An Experiment in Code Generation", *Proc. Sixth International Conference on Logic Programming*, Lisbon, June 1989, pp. 33–47. MIT Press.

[13] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

[14] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.

[15] G. L. Steele Jr., *Common Lisp: The Language*, Digital Press, 1984.

[16] A. Taylor, "Removal of Dereferencing and Trailing in Prolog Compilation", *Proc. Sixth International Conference on Logic Programming*, Lisbon, June 1989, pp. 48–60. MIT Press.

[17] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

[18] P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, PhD Dissertation, University of California, Berkeley, Nov. 1990.

[19] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles", *Proc. SIGPLAN-91 Conf. on Programming Language Design and Implementation*, June 1991, pp. 59–70.

# A Appendix: Proof of NP Completeness

The following problem is useful in discussing the complexity of optimal call forwarding:

**Definition A.1** The Optimal Linear Arrangement problem (OLA) is defined as follows: Given a graph $G = (V, E)$ and an integer $k$, find a permutation, $f$, from the vertices in $V$ to $1, \ldots, n$ such that defining the length of edge $(i, j)$ to be $|f(i) - f(j)|$, the total length of all edges is less than or equal to $k$. ∎

The following result is due to Garey, Johnson, and Stockmeyer [8, 9]:

**Theorem A.1** *The Optimal Linear Arrangement problem is NP-complete.*

The following result gives the complexity of optimal call forwarding:

**Theorem 3.1** *The determination of an optimal solution to a call forwarding problem is NP-complete. It remains NP-complete even if every entry action has equal size.*

**Proof:** We first formulate optimal call forwarding as a decision problem, as follows: "Given a call forwarding problem $I$ and an integer $K \geq 0$, is there a solution to $I$ with cost no greater than $K$?" We refer to this problem as CF. The proof is by reduction from Optimal Linear Arrangement problem, which, from Theorem A.1, is NP-complete. Let $G = (V, E), k$ be a particular instance of OLA. We make the following transformation to an instance $\langle A, C, w, s, k \rangle$ of CF, where:

- $A$ is the set of vertices $1, \ldots, n$ in $V$ along with two dummy vertices $s$ and $t$;
- The elements of $C$ are all doubleton sets:
  - corresponding to each edge $(u, v) \in E$, there is an element $\{u, v\}$ in $C$ with weight 1: for terminological simplicity in the discussion that follows, we refer to these elements as *normal sets*;
  - let $\Delta$ be the maximum degree of any vertex in $G$, then corresponding to each vertex $i \in G$ of degree $d_i$, there is an element $\{i, s\}$ in $C$ with weight $\frac{1}{2}(\Delta - d_i)$ (some of these sets

could have zero weight, in which case they can effectively be removed): we refer to these elements as *special sets*;

  - finally, there is an element $\{s,t\}$ in $C$ of weight $M$, where $M$ is large enough to ensure that $s$ and $t$ have to be the last two elements in any optimal ordering of the vertices ($M$ can be chosen to be $n^3$ or greater): we refer to this element as a *heavy set*.

- $s(I) = 1$ for every $I \in A$.

- $k = 0$.

We also have to define the number $K$ that is to bound the cost of the call forwarding problem so constructed. Let $K = \frac{1}{4}n(n+5)\Delta + 3M + k/2$. We claim that the instance of CF so defined has a solution with cost no greater than $K$ if and only if the given instance of OLA has a solution.

Consider any proposed order of elements in a solution to the instance of CF defined above. The cost of this solution can be decomposed as follows:

As we march along the list of elements, at each point we charge $\Delta/2$ to each of the elements we have seen so far but not to either of the special elements. If vertex $i \in G$ is encountered, the charge of $\Delta/2$ on vertex $i$ from then on can be thought of as paying $1/2$ towards each of the normal sets that contain $i$ and paying the entire cost of the special set that contains $i$. Now if both elements of a normal set have been encountered, the total cost of the set will from then on be picked up by these charges to the vertices. For a normal set $\{i,j\}$, after $i$ has been encountered and before $j$ has been encountered the extra charge of $1/2$ at each stage will be charged to the edge $(i,j)$. Breaking up the charges as above, one finds that for any order in which $s$ and $t$ finish last, the charge to the vertices is a constant independent of the order and is equal to $\frac{1}{4}(n(n+5)\Delta)$ and the charge for the heavy set is fixed at $3M$. The only variable is the charge to the edges and this charge will be exactly half the total length of the edges, since an edge gets charged only after one of its endpoints has been encountered and before the other endpoint has been encountered, i.e. for the "duration" of its length.

Thus there is a YES answer to the instance of CF created if and only if the total length of all "normal" edges is kept to $k$ or less, or, in other words, if and only if the instance of OLA is a YES-instance. (Note that since the cost of the special sets is entirely picked up by the vertices, the lengths of the special edges do not matter.)

# B Source Code for Some Benchmarks

The source code for the benchmarks used in the comparison between `jc` and C is given below. For space reasons, only the code for the main functions is given.

`nrev` : C :

```
typedef struct s {
    int head;
    struct s *tail;
  } cons_node;

cons_node *append(l1, l2)
cons_node *l1, *l2;
{
 cons_node *l3;
 if (l1 == NULL) return l2;
 else {
   for (l3=l1; l3->tail != NULL; l3=l3->tail)
      ;
   l3->tail = l2;
   return l1;
 }
}

cons_node *nrev(l)
cons_node *l;
{
 cons_node *l1;
 if (l==NULL) return NULL;
 else {
   l1 = l->tail;
   l->tail = NULL; /* reclaim head node */
   return append(nrev(l1), l);
 }
}
```

*Janus*:

```
nrev([], ^[]).
nrev([H|L1],^R) :-
     nrev(L1,^R1), app(R1,[H],^R).

app([],L,^L).
app([H|L1],L2,^[H|L3]) :- app(L1,L2,^L3).
```

`binomial` : C :

```
/* fact() as in the factorial benchmark */

int pow(x,i)
int x,i;
{ int prod;
  for (prod=1; i>0; i--) prod *= x;
  return prod;
}
int choose(n,k)
int n, k;
{
  return fact(n) / (fact(k) * fact(n-k));
}
int binomial(x,y,n)
```

```
int x,y;
{int i, prod=0;
  for (i = 0; i <= n; i++)
    prod += choose(n,i)*pow(x,i)*pow(y,n-i);
  return prod;
}
```

*Janus*:

```
/* fact() as in the factorial benchmark */

pow(X,N,^P) :- int(X) | pow(X,N,^P,1).
pow(X,0,^P,A) :- int(X), int(A) | P = A.
pow(X,N,^P,A) :-
   int(X), int(N), int(A), N > 0 |
     pow(X,N-1,^P,X*A).

choose(N,K,^C) :- int(N), int(K) |
     fact(N,^F1), fact(K,^F2), fact(N-K,^F3),
     C = F1 // (F2 * F3).

binomial(X,Y,N,^Z) :-
   int(X),int(Y),int(N),N >= 0 |
     binomial(X,Y,N,^Z,N).
binomial(_,_,_,^0,0).
binomial(X,Y,N,^Z,K) :-
   int(X),int(Y),int(N),int(K),K > 0 |
     binomial(X,Y,N,^Z1,K-1),
     choose(N,K,^C),
     pow(X,K,^Xp),
     pow(Y,N-K,^Yp),
     Z = Z1 + C*Xp*Yp.
```

dnf : *C* :

```
dnf(In, R, W, B)
int In[], R, W, B;
{ int temp;
  while (R <= W) {
    if (In[W] == 0) {
      temp=In[W]; In[W]=In[R]; In[R]=temp;
      R += 1;
    }
    else if (In[W] == 1)
      W -= 1;
    else if (In[W] == 2) {
      temp=In[W]; In[W]=In[B]; In[B]=temp;
      B -= 1; W -= 1;
    }
  }
}
```

*Janus*:

```
dnf(In,R,W,B,^Out) :-
   int(R),int(W),R > W | Out = In.
dnf(In,R,W,B,^Out) :-
   int(R),int(W),R =< W,In.W = red |
     dnf(In[R->In.W,W->In.R],R+1,W,B,^Out).
dnf(In,R,W,B,^Out) :-
   int(R),int(W),R =< W,In.W = white |
     dnf(In,R,W-1,B,^Out).
dnf(In,R,W,B,^Out) :-
   int(R),int(W),R =< W,In.W = blue |
     dnf(In[B->In.W,W->In.B],R,W-1,B-1,^Out).
```

tak : *C* :

```
int tak(x,y,z)
int x,y,z;
{
  if (x <= y) return z;
  return tak(tak(x-1,y,z),
             tak(y-1,z,x),
             tak(z-1,x,y));
}
```

*Janus*:

```
tak(X, Y, Z, ^A) :-
   int(X), int(Y), int(Z), X > Y |
     tak(X-1, Y, Z, ^A1),
     tak(Y-1, Z, X, ^A2),
     tak(Z-1, X, Y, ^A3),
     tak(A1, A2, A3, ^A).
tak(X, Y, Z, ^A) :-
   int(X), int(Y), int(Z), X =< Y |
     A = Z.
```

factorial : *C* :

```
int fact(n)
int n;
{int prod;
  for (prod = 1; n > 0; n--)
    prod *= n;
  return prod;
}
```

*Janus*:

```
fact(N,^X) :-
   int(N), N >= 0 | fact(N,^X,1).

fact(N,^F,A) :-
   int(A), int(N), N > 0 |
     fact(N-1,^F,A*N).
fact(0,^F,A) :- int(A) | F = A.
```

**Input:** A call forwarding problem $I = \langle E, C, w, s, k \rangle$.

**Output:** A solution to $I$, i.e., a permutation $\pi$ of $E$.

**Method:**

**begin**<br>
    $Active\_Sites := C$;<br>
    construct the dependency graph $G$ for legal execution orders;<br>
    $Avail\_Instrs :=$ the root nodes of $G$;<br>
    $Processed := \emptyset$;<br>
    $\pi := \langle\ \rangle$;<br>
    **for** each $c \in C$ **do** $count[c] := k$ **od**

    **while** $Avail\_Instrs \neq \emptyset$ **do**<br>
        **for** each $I \in Avail\_Instrs$ **do**<br>
            compute the weight of $I$ as $(\sum\{w(c) \mid c \in Active\_Sites \text{ and } I \in c\})/s(I)$;<br>
        **od**;<br>
        $I :=$ an element of $Avail\_Instrs$ with the least weight so computed;<br>
        $\pi :=$ append $I$ to the end of $\pi$;         /* *extend solution* */

        $Processed := Processed \cup \{I\}$;         /* *update list of available instructions* */<br>
        $Avail\_Instrs := (Avail\_Instrs \setminus \{I\}) \cup \{J \in E \mid preds(J) \subseteq Processed\}$;

        **for** each $c \in Active\_Sites$ s.t. $I \in c$ **do**         /* *update list of active sites* */<br>
            **if** $count[c] = 0$ **then**<br>
                delete $c$ from $Active\_Sites$;<br>
            **else**<br>
                $count[c] := count[c] - 1$;<br>
            **fi**<br>
        **od**<br>
    **od**;<br>
    **return** $\pi$;<br>
**end**

Figure 1: A Greedy Algorithm for Call Forwarding

```
ave:   Arg1 := deref(Arg1)
       Arg2 := deref(Arg2)
       Arg3 := deref(Arg3)
       if ¬List(Arg1) goto Err
       if ¬Number(Arg2) goto Err
       if ¬Number(Arg3) goto Err
       if Arg1 == NIL goto L1
       t1 := head(Arg1)
       Arg1 := tail(Arg1)
       t1 := deref(t1)
       if ¬Number(t1) goto Err
       Arg2 := add(Arg2, t1)
       Arg3 := add(Arg3, 1)
       goto ave
L1 :   t1 := div(Arg2, Arg3)
       Arg4 := deref(Arg4)
       assign(Arg4, t1)
```

(a) Before Call Forwarding

```
ave:   Arg2 := deref(Arg2)
       Arg3 := deref(Arg3)
       if ¬Number(Arg2) goto Err
       if ¬Number(Arg3) goto Err
L0 :   Arg1 := deref(Arg1)
       if ¬List(Arg1) goto Err
       if Arg1 == NIL goto L1
       t1 := head(Arg1)
       Arg1 := tail(Arg1)
       t1 := deref(t1)
       if ¬Number(t1) goto Err
       Arg2 := add(Arg2, t1)
       Arg3 := add(Arg3, 1)
       goto L0
L1 :   t1 := div(Arg2, Arg3)
       Arg4 := deref(Arg4)
       assign(Arg4, t1)
```

(b) After Call Forwarding

Figure 2: The Effect of Call Forwarding on Intermediate Code for the `ave` procedure

| Program | no optimization | greedy | optimal |
|---|---|---|---|
| `hanoi` | 492 | 225 | 225 |
| `tak` | 574 | 172 | 172 |
| `nrev` | 726 | 360 | 360 |
| `qsort` | 1776 | 450 | 450 |
| `factorial` | 129 | 24 | 24 |
| `merge` | 720 | 330 | 330 |
| `dnf` | 124 | 25 | 25 |
| `pi` | 306 | 30 | 30 |
| `binomial` | 5963 | 1304 | 1304 |

Table 1: Efficacy of the greedy Call Forwarding heuristic (in Sparc assembly instructions)

| Program | w/o forwarding (ms) | with forwarding (ms) | % improvement |
|---|---|---|---|
| `binomial` | 5.95 | 5.14 | 13.6 |
| `hanoi` | 186 | 163 | 12.4 |
| `tak` | 299 | 207 | 30.8 |
| `nrev` | 1.17 | 0.716 | 38.8 |
| `qsort` | 2.31 | 1.87 | 19.0 |
| `merge` | 0.745 | 0.613 | 17.7 |
| `dnf` | 0.356 | 0.191 | 46.3 |

Table 2: Performance Improvement due to Call Forwarding

| Program | `jc` (J) (ms) | Sicstus (S) (ms) | Quintus (Q) (ms) | S/J | Q/J |
|---|---|---|---|---|---|
| `hanoi` | 163 | 300 | 690 | 1.84 | 4.23 |
| `tak` | 207 | 730 | 2200 | 3.53 | 10.63 |
| `nrev` | 0.716 | 1.8 | 7.9 | 2.51 | 11.03 |
| `qsort` | 1.87 | 5.1 | 9.4 | 2.73 | 5.03 |
| `factorial` | 0.049 | 0.44 | 0.27 | 8.98 | 5.51 |
| Geometric Mean : | | | | 3.31 | 6.72 |

Table 3: The Performance of `jc`, compared with Sicstus and Quintus Prolog

| Program | `jc` (J) (ms) | C (unopt) (ms) | C (opt: `-O4`) | J/C-unopt | J/C-opt |
|---|---|---|---|---|---|
| `nrev` | 0.716 | 0.89 | 0.52 | 0.80 | 1.38 |
| `binomial` | 5.14 | 4.76 | 3.17 | 1.08 | 1.62 |
| `dnf` | 0.191 | 0.191 | 0.061 | 1.00 | 3.13 |
| `qsort` | 1.33 | 1.25 | 0.34 | 1.06 | 3.91 |
| `tak` | 207 | 208 | 72 | 1.00 | 2.88 |
| `factorial` | 0.049 | 0.049 | 0.036 | 1.00 | 1.36 |
| Geometric Mean : | | | | 0.98 | 2.18 |

Table 4: The performance of `jc` compared to C