A GENERAL APPROACH TO THE OPTIMIZATION OF FUNCTION CALLS

Kay-Ulrich Felgentreu, Wolfram-Manfred Lippe

Institut für Numerische und instrumentelle Mathematik
der Universität Münster
Einsteinstraße 62, D - 4400 Münster

Abstract

This paper presents a general approach to the optimization of function calls. We define the class of "low cost function calls" and introduce a technique of detecting and executing these calls in a modified shallow binding system known as "standardized shallow binding". We show that by this technique the overhead expenses of changing environments for low cost calls are nearly cut down to zero. The new method can be applied to any imperative or applicative language. In this paper statically scoped LISP is taken as an example; it is shown how the technique has been applied in the implementation of a LISP interpreter. We also prove that our method exceeds a number of optimizations that have been proposed recently.

1. Introduction

Most higher programming languages, such as PASCAL, LISP etc., contain the notion of applying functions to arguments. In applicative languages (e.g. LISP, SASL) this is the main concept of computation. However, function calls are rather expensive operations on von Neumann architectures. When an applicative program is executed, a considerable part of the execution time is spent on changing environments, i.e. saving and restoring variable bindings. So it is not surprising that a lot of studies have been done to reduce these overhead expenses ([Gr78], [Pe78], [Fe84], [Sa-Ja84], [Fe/Li86]). They all combine a special method of binding variables ("shallow binding") with a technique of handling certain recursive functions ("tail recursive functions", "covered tail recursive functions", etc.). Some interesting improvements have been achieved in this area, but the restriction to special recursive functions has not yet been overcome.

Here we will take a more general approach to the optimization of function calls. Based on the technique of shallow binding, we will define a class of function calls for which the overhead expenses can almost be cut down to zero. We refer to these calls as "low cost calls". This notion includes all the above-mentioned concepts and also covers quite a number of non-recursive calls that have not been considered before.

In this paper a technique of detecting and executing low cost calls is developed. It can be used for any imperative ("procedural") or applicative programming language, no matter whether the programs are compiled or interpreted. In this paper we will demonstrate the power of the new technique in an implementation of static scope LISP.

As is usually done, we implement LISP as an interpreter-based inter-actice system. The interpretation consists of three phases

(1) the read phase: the program and the data are read from the input and stored in the memory in a form that is well suited for phase (2).
(2) the execution phase ("actual interpretation"): the program, which is a LISP function, is applied to the data.
(3) the output phase: the result of the application is printed on the screen.

The optimization described in this paper follows the basic strategy of putting a little extra effort into the read phase in order to gain a lot of storage and execution time during the actual interpretation: When (or, depending on the implementation, after) the program is read from the input, each function call which can be interpreted without saving and restoring variable bindings is marked with the atom T, all the other calls are marked F. So, during the actual interpretation, the interpreter can execute all the calls marked T with a minimum of overhead expenses. In this paper we will develop the decision algorithm needed in the read phase.

We proceed as follows: After a short review of the basic technique of shallow binding, we will first show how shallow binding can be improved by standardizing identifiers (section 2). This technique of "standardized shallow binding" will reduce the overhead expenses of any function call by approximately 50 per cent. Then we will introduce the notion of low cost calls (section 3). We will see that in general low cost calls are not decidable, i.e. there is no algorithm that decides whether a given function call in any given program is a low cost call or not. However, in the <u>standardized</u> shallow binding system, the decision problem is a lot simpler so that low cost calls can actually be detected, and, furthermore, they can be detected very efficiently. Finally we show that previous approaches, such as optimizing tail recursive, covered tail recursive and (to a certain extend) even crossed tail recursive function calls are simple cases of low cost calls.

2. Shallow Binding and Standardization

The technique of shallow binding for statically scoped LISP (as for instance described in [Fe/Li/Si86]) can be summarized as follows:

Given a program p, each non-standard identifier id occurring in p is uniquely associated with one location ("address") loc(id) in the heap, the so-called value cell of id, which is supposed to contain the current value of id throughout the whole interpretation (*). Note that if two functions f and g have the same variable x, both occurrences of x are identified with the same address. Further, each non-standard function f occurring in p is enclosed in a list (CLOSURE f freevals), the so-called closure of f, where freevals is (a pointer to) the list of the values of the free non-standard identifiers occurring in f. Note that a function f may have different closures within the same program p (see [Fe/Li/Si86] for an example). This closure technique is known to guarantee static scoping [Fe84].

Now, let us consider a user-defined (i.e. non-standard) function f = (LABEL id_f (v_1 ... v_n) b), where id_f is the function identifier of f, v_i are the variables of f, and b is the body of f. In order to guarantee the condition (*), the following steps have to be executed when interpreting a call (f a_1 ... a_n) or (id_f a_1 ... a_n) of f :

1) Before the evaluation of b, the old contents of the value cells of $id_f, v_1, \dots, v_n$ are pushed onto the stack, along with their addresses (saving the environment), and the new values (i.e. the closure of f, which is considered to be the "value of f", and the values of the arguments a_i) are loaded into these value cells. Then the evaluation of b is started by a subroutine jump to EVAL. This gives the following state of the memory:

heap

************	p_f	a_1'	....	a_n'	************
	loc(id_f)	loc(v_1)	....	loc(v_n)	

where a_i' is the evaluated argument a_i, i = 1,...,n,
and p_f is a pointer to the closure of f.
The value cells are not necessarily contiguous or ordered !

stack

***	val(id_f)	loc(id_f)	val(v_1)	loc(v_1)	...	val(v_n)	loc(v_n)	n+1	a_{ret}
									TOS

where val(...) is the value of ... before the interpretation of the call,
and a_{ret} is the interpreter's return address when calling EVAL.

- figure 1 -

Here, periods denote repetition, blanks stand for free storage, and stars mean data which are not relevant in this context.

2) Now, when evaluating b, the value of id_f resp. $v_1,\ldots,v_n$ can be retrieved simply by loading the contents of its value cell. The value of any non-standard identifier occurring free in b is taken from the closure of f.

3) After the evaluation of b, the n+1 old values are removed from the stack and loaded back into the proper value cells (restoring the environment).

Note that in step 1 it is not sufficient to save the values of id_f, $v_1,\ldots,v_n$. Their addresses have to be saved as well, otherwise there will be no way of knowing into which cells these values have to be moved back after the evaluation of b (step 3). This is due to the fact that the value cells of $id_f,v_1,\ldots,v_n$ are not necessarily contiguous or ordered, since the programmer is free in his choice of non-standard identifiers. We will now show how standardizing helps to simplify the stack situation.

First, let us introduce the notion of the "standardized program". In a standardized program the identifiers of each function are "numbered" in the following sense: Let \$ be a letter of the LISP-alphabet, arbitrarily chosen, but fixed, so that all \$i, $i \in \mathbb{N}_0$, are admissible non-standard identifiers. Then the i-th variable of each function is \$i, and the function identifier is \$0. An example is given below. Let us postpone the problem of free variables for a moment.

Now, let p be a "closed" LISP- program, i.e. a program in which no user-defined function has free non-standard identifers. By renaming all non-standard identifiers systematically within their scopes, p can be transformed into a unique standardized program p^s. We call p^s the "standardized version of p". Since p^s is unique, the transformation $s: p \to p^s$ is a function; it is called the "standardization".

Example :

The standardized version of the well known function LAST

```
        (LABEL LAST (X) (COND ((NULL (CDR X)) (CAR X)) (T (LAST (CDR X)))))
```

is

```
        (LABEL $0 ($1) (COND ((NULL (CDR $1)) (CAR $1)) (T ($0 (CDR $1)))))
```

Obviously, a closed program p and its standardized version p^s are functionally equivalent.

Let us now briefly explain the standardization of non-closed programs. As an example we consider the following program which concatenates two given lists.

Example :

The standardized version of

```
  (LABEL AP (X Y)
     ((LABEL HELP (Z) (COND ((NULL Z) X) (T (CONS (CAR Z) (HELP (CDR Z)))))) Y))
```

is

```
   (LABEL $0 ($1 $2)
      ((LABEL $0 ($1) <$1>
         (COND ((NULL $1) (FREEVAL 1)) (T (CONS (CAR $1) ($0 (CDR $1)))))) $2))
```

Here the additional variable list <$1> tells the interpreter to construct the closure (CLOSURE (LABEL $0 ($1) ...) (val($1))), where val($1) is the value of $1 (i.e. the value of X in the original program). This causes val($1) to be "frozen in the environment of its defining occurrence". Finally, the pseudo call (FREEVAL 1) directs the interpreter to take the first value, which is val($1), from that closure. With this handling of free non-standard identifiers, we obtain :

Theorem :

Let p be any closed or non-closed LISP-program.
Then p and p^s are functionally equivalent.

This means that any program p can be standardized without changing its semantics. Let us now look at the interpretation of p^s. Due to shallow binding each identifier $i is always associated with the same location loc($i). For simplicity, we assume that loc($i) = i, which means that the value of the i-th variable of any function is always stored at the same address : i. So when a function f of n variables is called by (f a_1 ... a_n) or (id_f a_1 ... a_n), our shallow binding system creates the following state of the memory (cf. figure 1) :

heap

************	p_f	a_1'	....	a_n'	************
	0	1	....	n	

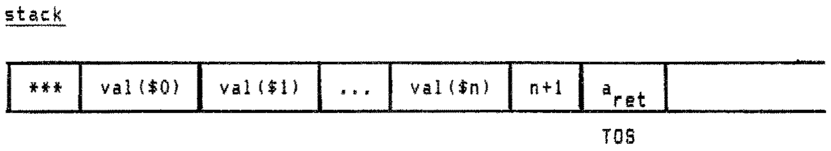
Note that all the value cells are now contiguous and ordered !

stack

***	val($0)	0	val($1)	1	...	val($n)	n	n+1	a_{ret}
									TOS

- figure 2 -

Here it is obviously redundant to store the addresses 0,1,...,n.
So the stack segment can be reduced to :



- figure 3 -

This shows that the standardization reduces the overhead expenses of calling <u>any</u> function by approximately 50 per cent.

In the following sections, the technique of shallow binding plus standardization, as proposed above, will be called "standardized shallow binding" for short.

3. Low Cost Function Calls

Studying the performance of interpreters which apply shallow binding or standardized shallow binding, it can be observed that most of their execution time is spent on saving and restoring environments. For many calls, however, it is not necessary to save and restore the environment at all (*). This is quite obvious for certain recursive calls (e.g. tail recursive and covered tail recursive calls - see [Fe/Li86] for definitions), and a lot of studies have been done on that topic within the last years ([Gr78], [Pe78], [Fe84], [Sa-Ja84], [Fe/Li/Si86]). In this section we will take a general approach to the optimization of function calls, which is not restricted to recursive functions at all. We define a class of function calls which have the property (*) (see above) and will therefore be referred to as "low cost calls".

First of all we have to introduce the notion of the "relevant local context" of a function call.

<u>Definition</u> :

Let p be a LISP-program, c a function call in p and b the body of the smallest function f containing c.
Then the <u>relevant local context of c</u> (for short: rlc(c)) is defined to be b excluding
(1) c itself
(2) all the syntactic expressions to the left of c
(3) if c is contained in a conclusion of a conditional form cf, all the clauses of cf which are to the right of c.

<u>Example</u> :

Let us consider the following non-standard function MAP123. Below, MAP123 is presented three times. In each case the relevant local context of the underlined function call is printed in bold type.

```
1)      (LABEL MAP123 (F1 F2 F3 E)
           (COND  ((NULL E)   NIL)
                  ((ATOM E)   (F1 E))
                  ( T         (CONS
                                (F2 (CAR E))
                                (MAP123 F3 F3 F3 (CDR E)) )) ) )

2)      (LABEL MAP123 (F1 F2 F3 E)
           (COND  ((NULL E)   NIL)
                  ((ATOM E)   (F1 E))
                  ( T         (CONS
                                (F2 (CAR E))
                                (MAP123 F3 F3 F3 (CDR E)) )) ) )

3)      (LABEL MAP123 (F1 F2 F3 E)
           (COND  ((NULL E)   NIL)
                  ((ATOM E)   (F1 E))
                  ( T         (CONS
                                (F2 (CAR E))
                                (MAP123 F3 F3 F3 (CDR E)) )) ) )
```

Let f, b and c be as in the definition above. Then the relevant local context of c contains all those syntactic forms which may still be evaluated within b after c is executed. This means that only the values of those non-standard identifiers which occur free in rlc(c) may still be accessed within b (as e.g. MAP123, F3 and E in the first example). On the other hand, the values of the function identifier and the variables of f will not be needed anymore after the evaluation of b. So we can state

Corollary :

Let p, f and c be as above. Let c be a call of a non-standard function g. Let Idf(f) resp. Idf(g) be the set consisting of the variables and the function identifier of f resp. g, and let Free(rlc(c)) denote the set of all non-standard identifers occurring free in rlc(c). If

(C1) $Idf(g) \subseteq Idf(f)$ and

(C2) $Idf(g) \cap Free(rlc(c)) = \emptyset$

then the value of any identifier $v \in Idf(g)$ is not accessed anymore after the execution of c.

So, if (C1) and (C2) hold for a call of a function g, the saving and restoring of the environment can be omitted for g. Consequently, if (C1) and (C2) hold for all the non-standard functions which are called by c, then c can always be executed without saving and restoring. Therefore we define :

Definition :

Let p, f and c be as above. Then c is called a low cost call iff for each non-standard function g in p the following holds:

(c is a call of g => (C1) and (C2) hold for g)

Examples :

The recursive call of MAP123 is a low cost call.
Further, every tail recursive and covered tail recursive call is a low cost call, since f = g (so Idf(f) = Ifg(g)) and rlc(c) never contains any identifiers at all, so Free(rlc(c)) is the empty set. This means that all the optimizations discussed in [Gr78], [Pe78], [Fe84], [Fe/Li/Si86] and many cases discussed in [Sa-Ja84] are included here. Examples of non-recursive low cost calls will be given lateron in this paper.

However, the example MAP123 also shows the principal problem about low cost calls: if we try to determine whether the calls (F1 E) and (F2 (CAR E)) are low cost calls or not, we run into trouble since we cannot decide which function(s) g will be bound to F1 resp. F2 during the interpretation. In the example above this is because the context of the function is not given. But even if MAP123 is defined within another function so that the context of its declaration is known, the following theorem holds:

Theorem :

In general it is statically undecidable whether a function call c in a LISP-program p is a low cost call or not.

A proof is given in [Fe85] by showing that the decidability of low cost calls would imply the decidability of correct parameter transmission, which is known to be undecidable for languages like ALGOL or LISP (see [La73], [Li/Si79]) which allow functional arguments and free variables in inner functions.

This theorem remains true even if p is supposed to have correct parameter transmission ([Fe85]).

Now, at this crucial point, the standardization proposed in the previous section provides the key to a practical solution:

Let us consider a call c of a function g, e.g. c = $(g\ a_1\ \ldots\ a_n)$, in a standardized program p, and let c have correct parameter transmission, i.e. g is a function of n variables. Since p is standardized, the function identifier of g is \$0, and the variables of g are \$1,...,\$n, so Idf(g) = {\$0,\$1,...,\$n}. (Note that in a non-standardized program the set Idf(g) is in general undecidable [Fe85] !).
Since both the set Idf(f), f being the smallest function containing

c, and the set Free(rlc(c)) can be constructed simply by scanning the program p, (C1) and (C2) can be computed, so we get the following theorem ([Fe85]) :

Theorem :

Let c be a non-standard function call in a standardized program p which has correct parameter transmission. Then it is decidable whether c is a low cost call or not.

This solves the decidability problem under the assumptions that
- the program p is standardized, and
- the program p has correct parameter transmission, and
- the call c is a non-standard function call.

For practical purposes, none of these assumptions are restrictive at all, since the standardization is simply a matter of variable allocation; programs with incorrect parameter transmission terminate with an error anyway; and the shallow binding mechanism is never applied to standard function calls. So the theorem provides a practical solution to the decidability of low cost calls.

Examples :

In the standardized version of MAP123, all the three calls (F1 E), (F2 (CDR E)) and (MAP123 F3 F3 F3 (CDR E)) are low cost calls. Note, however, that neither (F1 E) nor (F2 (CDR E)) are recursive! Further, crossed (covered) tail recursive calls of functions f and g are low cost calls in the standardized version, if f and g have the same number of variables (see e.g. ODD and EVEN in [Sa-Ja84]).

Finally, we give an efficient algorithm that detects all the low cost calls in a given standardized program p^s in just one pass:

The Detection Algorithm

In addition to the standardization, a little transformation is done to function calls: Any call $c = (x\ a_1\ \ldots\ a_n)$ is transformed into $(x\ \ t\ \ a_1\ \ \ldots\ \ a_n)$ where t is a pointer to the variable list of the smallest function f containing c. This simple transformation will be helpful later on (see step 5 below). Now, p^s is travered right to left, according to the following rules:

1. Skip constants.
2. Enter each function definition f with an empty set S of addresses. After leaving f, go on only with the set of the free non-standard identifiers of f.
3. While traversing f, pick up every non-standard identifier and add it to S.
4. When entering a conditional form cf in f, stack S. Let S′ be the accumulated set when entering a conclusion co with-

in cf. Copy S from TOS (top of stack) and stack S'. Use S when traversing co.
Let S" be the accumulated set when leaving co. Pop S' and go on with the union of S' and S".
After leaving cf, pop S and dispose of it.

5. Now, let S be the accumulated set when reaching a call $c = (x\ t\ a_1 \ldots\ a_n)$ within f. Then S = Free(rlc(c)). Calculate (C1) and (C2) as described above, using t to determine Idf(f). If both conditions are satisfied, replace t by T, otherwise replace t by F.

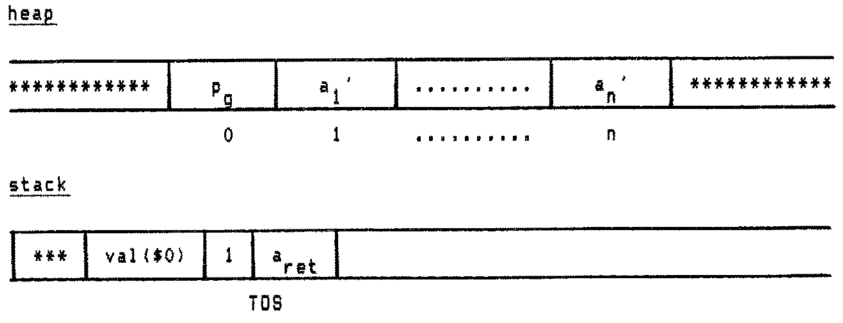
Now every low cost call in p^S is marked T, every non-low cost call is marked F. So the interpreter can execute function calls most efficiently in the following way:

The Interpretation of Function Calls

When the interpreter encounters a call $c = (x\ m\ a_1 \ldots a_n)$, x and $a_1, \ldots, a_n$ are evaluated as is usually done. The marker m is either T or F. The evaluation of x yields a function g. The interpreter checks for correct parameter transmission and terminates with an error if g is not an n-ary function. If g is a standard function, g and $a_1', \ldots, a_n'$ are passed to APPLY as is usually done, and m is ignored. Otherwise the following two cases have to be distinguished:

Case I : m = F
In this case c is not a low cost call, so the interpreter proceeds as before, pushing the current values of all v ∈ Idf(g), as shown in the previous figures.

Case II : m = T
In this case c is a low cost call, so the saving and restoring of variable bindings are omitted, and we get the following simplified state of the memory:



Note that val($0) is a pointer to the "previous closure", i.e. to the closure of the smallest function f containing c.

- figure 4 -

Here the "1" indicates that, after the evaluation of c, no values have to be restored, except for the pointer to the previous closure, which must be restored in order to be able to retrieve the proper values of the free variables of f furtheron. However, in many cases, even these entries can be omitted:

- For recursive calls, the current closure is the same as the previous one.
- For calls of closed functions the current closure does not contain any values and will never be accessed.

In these cases, the contents of the value cell 0 do not need to be updated, and the control can be passed to EVAL by a simple jump instead of a subroutine jump. Thus we can even omit to stack val($0), the "1" and the return address a_{ret}. So, summing up, we can state:

For any execution of a low cost call the stack either does not grow at all or grows by only three entries, which is just as little as for a function of zero variables (cf. figure 3).

4. Conclusion

We have developed an optimized interpreter for static scope LISP. Its efficiency is the result of combining the technique of standardized shallow binding - which reduces the overhead expenses of any function call by roughly 50 per cent - with a method of detecting and executing low cost calls, which eliminates most of the remaining overhead expenses (for low cost calls). The interpreter has been simulated in INTERLISP on an IBM 4381 and is currently being implemented in C on different microcomputers.

Further research will be done in the area of low cost calls, e.g. concerning the order in which arguments should be evaluated. Permutations of this order might provide for further improvements by artificially increasing the number of low cost calls.

References

[Ba78] Baker, H. G.
Shallow Binding in LISP 1.5
CACM, Vol. 21 No. 7, pp. 565-569, July 1978

[Ba80] Bauchrowitz, N.
Vergleich einer operationellen mit einer denotationellen Semantik für LISP
Diplomarbeit am Institut für Informatik und Praktische Mathematik der Universität Kiel, 1980

[Fe84]	Felgentreu, K.-U. Implementierung eines schnellen LISP-Interpretierers Diplomarbeit am Institut für Informatik und Praktische Mathematik der Universität Kiel, February 1984
[Fe85]	Felgentreu, K.-U. Decidability Problems concerning the Optimization of Function Calls Bericht Nr. 3/85-I, Institut für Instrumentelle Mathematik der Universität Münster, 1985
[Fe/Li86]	Felgentreu, K.-U., Lippe, W.-M. Dynamic Optimization of Covered Tail Recursive Functions in Applicative Languages (to appear in the Proceedings of the ACM Computer Science Conference February 1986)
[Fe/Li/Si86]	Felgentreu, K.-U., Lippe, W.-M., Simon, F. Optimizing Static Scope LISP by Repetitive Interpretation of Recursive Function Calls (to appear in IEEE Transactions on Software Engineering)
[Gr78]	Greussay, P. Iterative Interpretation of Tail-Recursive LISP Procedures in Ecole de la Recherche, Universite de Paris, 1978
[La73]	Langmaack, H. On correct Procedure Parameter Transmission in Higher Programming Languages Acta Informatica 2, p. 110-142, 1973
[Li/Si79]	Lippe, W.-M., Simon, F. LISP/N - Basic Definitions and Properties Bericht Nr. 4/79, Institut für Informatik und Praktische Mathematik, Universität Kiel, Oktober 1979
[McC66]	McCarthy, J., et. al. LISP 1.5 Programmer's Manual MIT Press, Cambridge, Massachusetts, 1966
[Pe78]	Perrot, J. F. Principes d'implementation de processus recursifs in Ecole de la Recherche, Universite de Paris, 1978
[Sa-Ja84]	Saint-James, E. Recursion is more efficient than Iteration Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August 1984