Efficient Stack Allocation for Tail-Recursive Languages

Chris Hanson  
MIT Artificial Intelligence Laboratory

1 Introduction

The Scheme dialect of Lisp [9] is properly tail-recursive—it relies entirely on procedure calls to express iteration. In Scheme, a tail-recursive procedure call (that is, a call in which the calling procedure does not do any further processing of the returned value) is essentially a goto that passes arguments, as was first pointed out by Steele [13]. In a properly tail-recursive language, there is no need for any explicit iteration constructs such as do or while; these can all be defined in terms of ordinary procedure calls.

As elegant as tail-recursion may be from the perspective of the programmer or the theoretician, it poses challenges for the compiler designer. One of the crucial decisions in the design of a compiler is the formation of a strategy for memory allocation and deallocation. An important aspect of this strategy is the treatment of memory locations used to hold the bindings of local variables. Because local variables play a significant role in most computer languages, their treatment can have a noticeable impact on a program's execution speed and run-time space requirements. Compilers for many block-structured languages use a simple strategy when allocating local variables: stack allocation. This strategy is supported by hardware on many computers, and by software mechanisms such as the access link and the display. However, the standard methods for implementing stack allocation assume that the language is not tail-recursive, and a straightforward application of these methods to a tail-recursive language can result in non-tail-recursive compiled code.

This paper describes stack-allocation techniques for compiling tail-recursive languages. We do not claim that these are the only techniques that can be used to solve the problem, nor do we compare them to other techniques. Instead, we use our techniques as a concrete example to demonstrate that it is possible to implement stack allocation of local variables without sacrificing tail recursion, which to our knowledge has not previously been shown.

We have implemented these techniques in the MIT Scheme compiler [11]. Although efficiency is a secondary issue for this paper, it is nonetheless important: we show that their performance is comparable to implementations of non-tail-recursive languages. In particular, the code sequences generated for tail-recursive procedure calls are as efficient as those that implement the special-purpose iteration constructs of non-tail-recursive languages.

Sections 2 and 3 present a standard implementation of stack allocation and demonstrate why it fails for a tail-recursive language. Section 4 shows how the implementation can be modified to remedy this failure. The modified implementation is somewhat impractical, so section 5 presents a refinement that results in a simple and practical implementation; this is the paper's main contribution. Section 6 shows additional static analysis techniques that can be used to make the implementation more efficient, and section 7 compares this final implementation to a non-tail-recursive one, in which iteration is accomplished using explicit iteration constructs rather than ordinary procedure calls.

2 General Framework

We'll follow an empirical approach in this paper, starting with techniques from a well-established compiler text and making successive modifications until we achieve the desired results. Our source language will be Scheme, because it is tail-recursive, widely known, and fun to program in.

Standard texts on compilation, such as [2] inform us that stack allocation of run-time storage is a very common technique, applicable to such lexically-scoped languages as Algol, Pascal, and C; this is a good sign, as Scheme is closely related to Algol. The stack is allocated in segments called activation records, each containing the information associated with a single procedure invocation. Each activation record consists of a collection of fields. The particular stack discipline described in chapter 7 of [2] uses seven fields, of which we need only four:

Actual Parameters The argument values of the called procedure.

Saved Machine Status The return address indicating where to continue execution of the calling procedure.

Access Link A pointer to the activation record of the procedure that is the lexical parent of the called procedure. This link is used to implement lexical variable reference.

Temporaries

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 089791-368-X/90/0006/0106 $1.50
The remaining three fields won't be used:

Local Data Because Scheme has no local declarations, it needs no local data. The role played by these declarations in other languages is subsumed by Scheme's lambda bindings.

Control Link A pointer to the activation record of the procedure that invoked the called procedure. This field is normally used for three purposes:

- Analogous to the access link, the control link can be used to implement "deep access" in a dynamically-bound language.
- Non-local control transfers can be implemented by searching the control link chain to find a particular activation record.
- Procedures that accept varying numbers of arguments can use the control link to find out how many arguments they were passed.

None of these purposes is directly relevant to our discussion. The MIT Scheme implementation that we will discuss uses alternative techniques to solve all of these problems, so we won't use this field.

Returned Value We'll assume that all procedures return at most one value, and that a register is used to hold a procedure's returned value, so this field won't be needed.

Although Scheme is closely related to Algol, it differs in that Scheme's procedures are first-class objects. The major consequence of this difference is that the activation records for some Scheme procedures must be allocated on the heap, and deallocated by garbage collection. Fortunately, use of stack allocation and heap allocation may be combined in the same implementation, and there are methods for determining when stack allocation is applicable [11]. The other relevant difference between Scheme and Algol is that Scheme has first-class continuations, but even so, standard techniques permit the use of stack allocation [4]. This paper ignores these issues and focuses on the relationship between stack allocation and tail recursion.

Now let's choose an example program and simulate its run-time behavior for our proposed compiler. The example we choose is deliberately simple; it shows the basic relationship between stack allocation and tail recursion. The procedure accumulate, shown in Figure 1, is a simple iterative procedure that is sometimes used to implement the variablearity equivalent of a binary procedure (e.g. +), where the set of arguments is represented as a list.

To examine the run-time behavior of our example, we'll use a visual model showing the arrangement of activation records on the stack at particular moments during execution. By choosing appropriate moments, we'll see only those allocation records that we're interested in, not uninteresting ones such as those generated for calls to primitives. Also, we'll "crop" the snapshots to eliminate any activation records associated with the caller of the example procedure.

Each activation record is represented by a box, and the boxes are "stacked" one on top of another; the top of the stack is the topmost box. Within each box, the fields of the activation record are represented as follows:

```
(defn accumulate
  (lambda (binary-op initial iters)
    (if (null? iters)
      initial
      (letrec ((loop
                    (lambda (value iters)
                      (if (null? iters)
                        value
                        (loop (binary-op value (car iters))
                              (cdr iters))))))
        (loop (car iters) (cdr iters))))))
```

Figure 1: The accumulate procedure.

- Each actual parameter is represented by a line containing name = value.
- An access link, if any, is represented by a line containing access link, with an arrow showing where the access link points to. The arrow is suppressed in the outermost activation record, because we're not concerned with how "top-level" variables are accessed.
- The return address, if any, is represented by a line containing return address.
- The activation record's temporary storage is not shown.

3 A Traditional Implementation

Now we choose the compiler's procedure call and return sequences. Because our stack-allocation techniques are specifically designed for non-tail-recursive languages, we initially ignore tail recursion, and later we'll modify the implementation to introduce it. So we begin with the following procedure call sequence:

- Push the return address.
- Push the access link.
- Push the called procedure's arguments.
- Jump to the called procedure's start address.

and the following return sequence:

- Pop the topmost activation record off the stack.
- Jump to that record's return address.

Figure 2 shows the activation records generated by this call sequence when accumulate is called with some typical arguments: (a) shows the initial activation record for the procedure accumulate; (b) shows the additional record generated by the letrec expression that binds loop; and (c) through (e) show the records generated by successive invocations of the procedure loop. The figure doesn't show the effect of the return sequence, which simply pops the records off the stack one at a time.
As we expect for a non-tail-recursive language, Figure 2 shows the typical pattern associated with a recursive process: the number of activation records increases in proportion to the number of times we invoke loop. If our compiler was properly tail-recursive we would instead see the pattern of an iterative process: the number of activation records would be constant as we went around the loop (see p. 32 of [1] for a discussion of recursive and iterative processes).

4 A Tail-Recursive Implementation

Now let's see what must to be done to make our compiler produce tail-recursive code. The basic problem is that we are storing too much information on the stack and must discard some. We'll begin this section by refining our concepts of "activation record" and "procedure call". Then we'll discuss what information we should discard, when it can be discarded, and why discarding it is sufficient to make our implementation tail-recursive. We'll finish the section by detailing the tail-recursive call and return sequences.

The traditional model's notion of an activation record is somewhat clumsy for the discussion that follows. As we shall soon see, an activation record's return address must sometimes be separated from the rest of the record. To facilitate this, we change our model by splitting each activation record into two parts: a control record containing a return address, and an environment record containing bindings and an access link; either kind of record may also contain temporaries. We'll use the term activation record to refer to either kind: what we formerly called an activation record will now be thought of as a combination of a control record and an environment record. Note that this doesn't change the contents of the stack, but only how we interpret them.

Another useful refinement to our model is to classify each procedure call as either a subproblem or a reduction. When the calling procedure will perform more computation after the called procedure has finished executing, the procedure call is a subproblem, otherwise it is a reduction. This classification is easy to implement in a compiler as it is a surface property of the program text. For example, procedure calls appearing as the last expression of a 'lambda' or 'letrec' body or as one of the arms of an 'if' expression are reductions. Procedure calls appearing as arguments to other procedure calls or as the predicate of an 'if' expression are subproblems. In accumulate, the calls to the procedures null?, car, cdr, and binary-op are subproblems and the calls to loop are reductions. (The letrec expression also expands into a reduction, regardless of how the recursion is implemented.)

With these refinements, we can say that achieving tail recursion requires (1) avoiding pushing unnecessary control records, and (2) popping environment records when they are no longer needed.

Unnecessary control records are those that correspond to reductions. It is easy to see that we need never push a control record for a reduction, because the action of its return address is known at compile time: it pops the caller's environment and control records, then jumps to the return address of the latter. Thus if we omit the control record from every reduction, the return sequence should be modified to pop environment records off the stack until it reaches a control record, then pop that control record and jumps to its return address. On the other hand, we must push control records for subproblems, because the compiler doesn't usually know what the caller will do after the return. 2 Referring to Figure 2(e), this control-record optimization eliminates all but one of the return addresses shown, the exception being that of the bottom record.

Now let's try to eliminate some of the environment records. As a consequence of the control-record optimization just described, the return sequence now discards any environment records that are above the topmost control record on the stack. But some environment records can and should be discarded sooner. In Figure 2(e), the second and third environment records from the stack's top can play no useful role in the computation, as neither their variable bindings nor their access links can be reached. Each of these two records becomes useless at the procedure call to loop that occurs within loop itself, indicating that we should attempt to detect and eliminate useless environment records during the call sequence. As our example indicates, we can sometimes find useless records at a reduction, however, we'll never find any useless environment records at a subproblem, because all of the records are needed to continue the calling procedure after the called procedure returns.

So which environment records can we pop at a reduction? At most, we can pop all environment records between the stack's top and the nearest control record; environment records below that control record may not be popped because they are needed by the caller who pushed the control record. However, we can't always pop all of the environment records above the control record, because some of these may be in the callee's access chain and thus will be needed by the callee. So what we need is a simple algorithm to decide which of these environment records, if any, is in the access chain of the callee.

Our popping algorithm is a loop that examines the topmost record on the stack: if that record is a control record or the environment record that the callee's access link points to (i.e. the callee's lexical parent), then we're done, otherwise we pop the record and continue the loop.

The algorithm is based upon the following claim: if two environment records are adjacent on the stack, then the access link of the upper record points to the lower record. The claim implies that all of the environment records between the topmost control record and the stack's top are part of a single access chain. If the callee shares part of this access chain, it will be the lower part, so we must pop the upper, unshared part before pushing the callee's new environment record.

This claim about adjacency is true because environment records are only pushed by procedure calls, and furthermore an environment record can only be adjacent to another one when the upper one has been pushed by a reduction, since a subproblem always pushes a control record before pushing its environment record. A reduction's record-popping algorithm guarantees that the new environment record's adjacent record is either a control record, or is the callee's lexical parent record. In the former case the new environment record will not be adjacent to any environment record, while in the latter case it will be adjacent to its lexical parent. In both cases our claim is supported.

Let's summarize how the new call and return sequences affect the stack. The subproblem call sequence doesn't dis-
Figure 2: Activation records created by evaluation of the expression (accumulate + 0 '(1 2 3)) when accumulate is compiled using non-tail-recursive methods.
informally show that our new call and return sequences are the stack, and makes that environment the current one. If then allocates a new environment frame for the callee and return address and a pointer to the current environment, tail-recursive. For a subproblem, the interpreter pushes a pushes nothing, discarding the current environment pointer recursive interpreter, such as that described in [l], we can environment frame. For a return, it discards the current interpreter's return addresses, it is clear that the interpreter andence between them is that the compiled code allocates the on the stack.

Our new return sequence is:

- If the stack's topmost record is an environment record other than the callee's lexical parent, pop it. Continue this process until reaching either a control record or the lexical parent.
- Push the access link and bindings that were computed earlier.
- Jump to the called procedure's start address.

Our new return sequence is:

- If the stack's topmost record is an environment record, pop it. Continue this process until the topmost record on the stack is a control record.
- Pop the control record off the stack and jump to its return address.

Figure 3 shows the effect of these new code sequences; it should be compared to Figure 2. Clearly accumulate now executes as an iterative process.

5 A Practical Implementation

The call and return sequences described in section 4 give correct tail-recursive behavior, but a straightforward implementation of them is both complicated and inefficient. Not only must we be able to distinguish environment and control records at run time, implying that each record must somehow be marked to indicate its type, but the compiler must emit record-popping loops for each reduction and return sequence; such loops are likely to consist of several machine instructions, and their execution time will be linear in the number of records popped. If we could statically determine how many records to pop, the code to pop them would merely change the stack pointer. This would execute in constant time, and on most machines would be a single instruction. Furthermore, without the need to dynamically distinguish record types, we would save the space needed to mark each record's type.

It is clear that compiler analysis to determine the discardable records is justified. However, despite its initial ap- pell, there are two serious drawbacks associated with static analysis: it's difficult and expensive to do well, and it's sometimes theoretically impossible to statically determine how many records to pop. So before looking at compiler analysis, we should first find a reasonably efficient mechanism that implements the record popping dynamically. Such a mechanism can be used either as an alternative to static analysis, or as a backup method when static analysis fails.

The popper mechanism, invented by Guilleremo Rozas [11] and implemented in two different versions of the MIT Scheme compiler, is one possibility. This mechanism works by adding a "popper" field to each record, which contains a tiny program that dynamically decides whether the record needs to be popped; the program takes arguments that provide it with contextual information. The compiled code jumps into the popper field of the topmost record. The code in that field either jumps to the popper of the next record, after modifying the arguments if necessary, or else stops and adjusts the stack pointer.

Rozas was dissatisfied with the performance and complexity of the popper mechanism, and a subsequent dis- cussion between him and Jonathan Rees, sparked by a pa- per contrasting stack allocation to heap allocation [3], led to their discovery of a simpler and more efficient mecha- nism [10], which has since been implemented by Richard Kelsey [8] for the Scheme 48 system. The author modified their mechanism to eliminate certain disadvantages (while introducing others) and implemented the modified mecha- nism in the MIT Scheme compiler in December 1987. The remainder of this section will describe the modified mechanism. We'll discuss the original mechanism later, as it re- mains an interesting possibility for future work.

Recall that earlier we examined the uses of the control- link field and concluded that it was unnecessary. Rozas and Rees made the discovery that a tail-recursive implementa- tion could take advantage of this field even when a non-tail-recursive implementation could not. The reason is that control links, which chain together the stack's control records, can be used to identify the topmost control record on the stack. In our original terminology, an activation record's control link points to the return address in the activation record immediately below. In our new terminology, the control link of an environment record points to the nearest control record that is below the environment record on the stack. The control link of the stack's topmost environment record points to the topmost control record.2

2Note that, except during the call and return sequences them- selves, the topmost record on the stack is always an environment record.
Figure 3: Activation records created by evaluation of the expression (accumulate + 0 (1 2 3)) when accumulate is compiled using tail-recursive methods.
We can restate the record-discarding algorithms in a slightly different form to help show how the control link can be used: the return sequence discards all environment frames above the topmost control record, while the reduction sequence discards all environment frames above either the topmost control record or callee's lexical parent record, whichever is closer to the top of the stack. We can implement the first case by moving the control link into the stack pointer, and the second by comparing the control link and the callee's access link, moving the topmost one into the stack pointer. Both of these are simple to implement and reasonably efficient.

Let's design new call and return sequences that take advantage of the control link. If we store a procedure's control link in its environment record, we'll have to copy the control link every time a reduction is executed. Instead, we introduce a register to hold the control link; when we execute a subproblem, we'll save the contents of this register in the control record, along with the return address, and set the register to point to the new control record. This is a tradeoff that simplifies reductions at the expense of subproblems.

Here is the new call sequence for a subproblem:

- Push the return address.
- Push the contents of the control-link register.
- Copy the contents of the stack-pointer register into the control-link register.
- Push the access link and bindings of the called procedure.
- Jump to the called procedure's start address.

The new return sequence is:

- Copy the contents of the control-link register into the stack-pointer register.
- Pop the previous control link off the stack and put it in the control-link register.
- Pop the return address off the stack and jump to it.

The new call sequence for a reduction is:

- Compute the access link and bindings of the new environment record, saving them somewhere other than the stack.
- Compare the access link to the contents of the control-link register. Set the stack-pointer register to whichever of these is closer to the top of the stack.
- Push the access link and bindings that were computed earlier.
- Jump to the called procedure's start address.

The efficiency of these new code sequences is significantly better than that of the previous sequences: the new ones execute in constant time, and the number of machine instructions needed to encode them is small. Their major drawbacks are the need to dedicate a machine register to hold the control link, and to push an extra word for each control record.

Figure 4 shows the activation records that result from these call and return sequences. The notation "CL" and an arrow is used to indicate the contents of the control-link register, and a saved control link is indicated by a line containing control link. Comparison to Figure 3 shows that these snapshots differ only in the handling of the control link.

6 An Efficient Implementation

The control-link mechanism of section 5 provides a correct implementation of tail recursion with acceptable performance. The mechanism dynamically determines the records to discard. If we make this determination statically we can generate even more efficient code. What follows are some useful ad hoc rules, with emphasis on what can be accomplished by a simple compiler using these rules.

Static analysis readily admits two relevant improvements to our code sequences. If we predict the value of a control link, then we need not generate the instructions to load, save, and restore the register for the link; if we predict the value of an access link, we may eliminate the link from its record, reducing code size. Here "predict the value" means the value can be determined at run time by adding a constant offset to some other known quantity. A control link is predicted by determining its offset from the stack pointer, while an access link is predicted by determining its offset from the beginning of the record that it is stored in. With good predictions, the compiled code can refer to most quantities as offsets from the stack pointer. Many machines have instructions supporting this form of reference, and we'd expect a good C compiler, for example, to generate such code.

We can predict the value of a record's access link in several situations. One situation occurs commonly and is relatively easy to determine: when a given procedure is always called as a reduction, the environment record of the procedure's lexical parent is always immediately below the environment record of the procedure itself (assuming that the lexical parent record is stack-allocated; otherwise this analysis is not interesting). In accumulate, loop is such a procedure, whose parent is the letrec that binds the variable loop. The letrec also expands into such a procedure, whose parent is the procedure accumulate. In typical Scheme code most let expressions occur this way. Performing the optimization transforms such a let expression into a sequence of stack pushes.

A simple analyzer can find many of the environment records that this rule applies to. A reduction whose operator is a lambda expression is an obvious case, and a common one: a let expression expands into such code. Another common case is a lambda expression that is bound to a variable by a let or letrec expression, and every reference to that variable appears as the operator of a procedure call. Both of these cases can be located without sophisticated analysis, and represent many access links. Of course, if more complete dataflow information is available the analyzer can take advantage of it to find other cases.

Elimination of control links is harder than elimination of access links, and control links that can be eliminated are less common. For these reasons, the designer of a simple compiler may wish to omit control-link analysis entirely. However, one rule for eliminating control links is so simple, and applies to so many procedures, that most compilers should implement it: when a procedure's lexical-parent environ-
Figure 4: Activation records created by evaluation of the expression (accumulate + 0 '(1 2 3)) when accumulate is compiled using the control-link mechanism.
ment is not stack-allocated, the procedure does not need a control link: the control record is always immediately below the procedure's environment record (recall that the record below an environment record is either a control record or the lexical parent's environment record).

Beyond this simple rule is another that requires dataflow analysis. For each procedure, find all of the calls to the procedure. If all of the calls are subproblems then a control link is not needed: the control record will always be immediately below the procedure's environment record. If all of the calls are reductions, then the environment record of the procedure's lexical parent is always immediately below: a control link is needed only if the lexical-parent procedure needs one. If some of the calls are reductions, and some are subproblems, then a control link is needed.

The compiler analysis we've just sketched results in each procedure being marked to indicate whether or not it requires a control link or an access link. The compiler's code generator examines these marks to generate the correct call sequences on the accumulate procedure. As we can see, accumulate is optimal in that the compiler is able to eliminate all of its access and control links, except for the special access link in the bottommost record.

7 Comparison to Standard Techniques

Having described an efficient tail-recursive implementation of stack allocation, we can ask how the quality of its generated code compares to code generated by traditional compilation techniques for non-tail-recursive languages. There are two cases in which direct comparison of tail-recursive and
Figure 5: Activation records created by evaluation of the expression \((\text{accumulate} + 0 \ (1 \ 2 \ 3))\) when accumulate is compiled using static analysis to eliminate access and control links.
LIAR's output differs in two ways from that of the hypothetical compiler we have been discussing. First, it does not push an environment record containing the binding of loop—it determines that the value of this variable is always known and need not be constructed. Second, while our hypothetical compiler would have computed the callee's record, stored it somewhere (perhaps in registers), adjusted the stack, and then pushed the record, LIAR's output constructs the environment record of the callee by incrementally overwriting that of the caller (see lines 25–28). This valuable optimization causes the object code to use side-effects on stack-allocated locations, even though no explicit side-effects appear in the source code. As an alternative to this optimization we could have written the Scheme program using explicit side-effects to make it more like that of the C program. However, we cannot rewrite the C program to eliminate those explicit side-effects without changing it from an iterative process to a recursive process, because C has no iteration constructs that bind variables.

Summarizing, the differences between the compilers' outputs are entirely due to calling and representational conventions. There is no fundamental difference in their efficiency:

- GCC's output uses a frame pointer while LIAR's output does not (e.g. Fig. 8 lines 2 and 10).
- GCC uses the convention that the caller deallocates the arguments of a procedure call, while with LIAR the callee does. (See Fig. 8 line 24, Fig. 7 line 16.)
- GCC uses the jsr instruction to push the return address, jumping to the callee at the same time; LIAR uses the pea instruction and does other work before jumping to the callee. (See Fig. 8 line 23, Fig. 7 lines 19 and 23.) Note that this convention is a consequence of tail-recursion: the return address appears below a procedure's arguments so that the arguments may be popped without popping the return address.
- GCC's output allocates its local variables immediately on entry to the procedure; LIAR's output allocates them when they are needed. (See Fig. 8 line 2, Fig. 7 lines 9 and 10.)

This shows that our techniques work as well as existing non-tail-recursive techniques for simple iterative loops of this kind.

It would be interesting to do a more thorough comparison of the two compilation techniques. However, this will require writing pairs of programs that generate equivalent stack-allocation patterns in tail-recursive and non-tail-recursive languages, which is difficult. Tail-recursive procedure calls are easily used to create processes with complex mixtures of recursive and iterative stack-allocation behavior, while most non-tail-recursive languages' iteration mechanisms have little or no control over stack-allocation behavior.

8 Future Directions

We've seen a specific set of methods to implement stack allocation of local variables for a tail-recursive language. Our use of the control link is a general solution to the problem of deallocating stack-allocated environment records. The compiler optimizations admit an implementation that is comparable in performance to an implementation for a non-tail-recursive language.
Figure 7: MIT Scheme compiler output for `accumulate`.

```assembly
accunulate:
mov.1 8(%esp),%a0  # test (NULL? ITEMS)
tst.l  %a0
bne  label_18  # branch if not null
mov.1 4(%esp),%a0  # value-register := INITIAL
lea  12(%esp),%sp  # deallocate arguments
rts  # return

label_18:
mov.1 4(%a0),-(%esp)  # allocate L := (CDR ITEMS)
mov.1 (%a0),-(%esp)  # allocate VALUE := (CAR ITEMS)

lambda_4:
mov.1 4(%esp),%a0  # test (NULL? L)
tst.l  %a0
bne  label_19  # branch if not null
mov.1 (%esp),%d0  # value-register := VALUE
lea  20(%esp),%sp  # deallocate args, L, VALUE
rts  # return

label_19:
pea  continuation_2  # push return address
mov.1 (%a0),-(%esp)  # push (CAR L)
mov.1 8(%esp),-(%esp)  # push VALUE
mov.1 20(%esp),-(%esp)  # push BINARY-OP
jmp  apply_2  # invoke BINARY-OP on two args

continuation_2:
mov.1 %d0,(%esp)  # VALUE := result
mov.1 4(%esp),%a0  # L := (CDR L)
mov.1 4(%a0),4(%esp)
bra  lambda_4  # continue loop
```

Figure 8: GNU C compiler output for `accumulate`.

```assembly
accumulate:
link.w  %a6, &-8  # allocate VALUE, L
mov.l  16(%a6),%a0  # test (ITEMS == 0)
tst.l  %a0
bne  L2  # branch if not null
mov.l  12(%a0),%d0  # value-register := INITIAL
unlink  %a6  # deallocate VALUE, L
rts  # return

L2:
mov.l  (%a0),-(%a6)  # VALUE := ITEMS->CAR
mov.l  4(%a0),-(%a6)  # L := ITEMS->CDR

L3:
mov.l  -8(%a6), %a0  # test (L == 0)
tst.l  %a0
bne  L4  # branch if not null
mov.l  -4(%a6),%d0  # value-register := VALUE
unlink  %a6  # deallocate VALUE, L
rts  # return

L4:
mov.l  (%a0),-(%esp)  # push L->CAR
mov.l  -4(%a0),-(%esp)  # push VALUE
mov.l  8(%a6),-(%esp)  # push BINARY_OP
jump  apply_2  # invoke BINARY_OP on two args
add.w  %d1, %sp  # discard call frame
mov.l  %a0,4(%esp)  # VALUE := result
mov.l  -8(%a6),%a0  # L := L->CDR
mov.l  4(%a0),-8(%a6)
bra  L3  # continue loop
```

117
What we haven't seen is perhaps more interesting. This paper has just touched the surface of how the methods work, ignoring any serious analysis or proof. A thorough treatment of the topic would compare these methods to alternatives such as register allocation and heap allocation to determine the performance tradeoffs. It should also compare these techniques to lambda-lifting [7], a popular technique that solves this problem by copying environment records rather than attempting to share them. The compiler analysis would be better presented as one part of a comprehensive analysis with deep understanding of the relationships between a program's parts. The ad hoc rules we've seen here are crude approximations to this.

Another useful experiment would be to compare the performance of the original form of the control-link mechanism, as invented by Rozas and Rees, to that of the form presented here. In the original form, records are popped in the return sequence but not the call sequence. If the stack overflows, a compaction process examines the records on the stack and removes those that are useless. This form has the advantage of eliminating some code at reductions, but the disadvantages of making access-link optimization more difficult and using more memory (thereby decreasing the effectiveness of caching); a hybrid form might capture the best of both.

Clearly, much interesting work remains.

Acknowledgements

Thank you to the wonderful people who helped create this paper. Some read drafts and debugged my writing (if only program debuggers were so good!), and all had interesting and useful comments: Hal Abelson, Olivier Danvy, Mike Eisenberg, Arthur Gleckler, David Kranz, Jonathan Rees, Jerry Sussman, and Franklyn Turbak. Special thanks to Bill Rozas, who not only read the paper, but was always there with a helpful word when the ideas were being implemented. Both the paper and the MIT Scheme compiler owe a great deal to his imagination and energy.

References


