An Efficient Implementation of Multiple Return Values in Scheme

J. Michael Ashley R. Kent Dybvig

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{*jashley, dyb*}*@cs.indiana.edu*

Abstract

This paper describes an implementation of the new Scheme multiple values interface. The implementation handles multiple values efficiently, with no run-time overhead for normal calls and returns. Error checks are performed where necessary to insure that the expected number of values is returned in all situations. The implementation fits cleanly with our direct-style compiler and stack-based representation of control, but is equally well suited to continuation-passing style compilers and to heap-based run-time architectures.

1 Introduction

In this paper we describe an implementation of the new Scheme multiple values interface that handles multiple values efficiently without penalizing code that does not use them or sacrificing our fast implementation of first-class continuations. Full run-time error checking is performed to insure that the correct number of values is received in all situations. We also describe how the implementation can be adapted to handle Common Lisp's multiple values interface.

The implementation first rewrites all direct calls to the primitives defined by the interface into internal forms that can be optimized by the compiler. These forms are further rewritten to eliminate uses of multiple values in all situations except across procedure calls. Values returned from procedure calls are handled much like values passed to procedures, simplifying the communication of multiple values to unknown consumers. Separate multiple- and single-value return points are used to eliminate most of the error checking overhead for multiple-value calls and all run-time overhead from normal calls. The implementation generates especially efficient code when the multiple value consumer is known.

Although we assume a direct-style compiler with a stack-based run-time architecture, our results are applicable to compilers that convert their input to continuation-passing style and to systems that employ a heap-based run-time architecture.

This paper is organized as follows. Section 2 describes the Scheme multiple values interface and argues for full error checking in situations whose behavior is left unspecified by the adopted proposal. Section 3 presents our implementation of the interface. Section 4 discusses the performance characteristics of our implementation. Section 5 discusses related work. Section 6 summarizes and concludes the paper.

2 Multiple Return Values

A proposal to include a multiple values interface in the successor to the Revised[4] Report on Scheme [1] has been approved [10]. Two procedures, *values* and *call-with-values*, comprise the interface.

(*values* v_1 ...)

(*call-with-values producer consumer*)

The procedure *values* accepts any number of arguments and simply passes (returns) the arguments to its continuation. The *producer* argument to *call-with-values* may be any procedure accepting zero arguments, and *consumer* may be any procedure. *call-with-values* applies *consumer* to the values returned by invoking *producer* without arguments. The following simple examples demonstrate how *call-with-values* and *values* interact:

```
(call-with-values (lambda () (values 1 2)) +)  → 3
(call-with-values values (lambda args args))  → ()
```

In the second example, *values* itself serves as the producer. It is receives no arguments and thus returns no values.

The more realistic example below employs multiple values to divide a list nondestructively into two sublists of alternating elements.

```
(define split
  (lambda (ls)
    (if (or (null? ls) (null? (cdr ls)))
        (values ls '())
        (call-with-values
          (lambda () (split (cddr ls)))
          (lambda (odds evens)
            (values (cons (car ls) odds)
                    (cons (cadr ls) evens)))))))

(split '(a b c d e f))  →
    (a c e)
    (b d f)
```

At each level of recursion, the procedure *split* returns two values: a list of the odd-numbered elements from the argument list and a list of the even-numbered elements.

The continuation of a call to *values* need not be one established by *call-with-values*, nor must only *values* be used to return to a continuation established by *call-with-values*. In particular, (*values* v) and v are equivalent in all situations. For example:

(+ (*values* 2) 4) $\rightarrow$ 6

(**if** (*values* #t) 1 2) $\rightarrow$ 1

(*call-with-values*
  (**lambda** () 4)
  (**lambda** (x) x)) $\rightarrow$ 4

Similarly, *values* may be used to pass any number of values to a continuation that ignores the values[1], as in:

(**begin** (*values* 1 2 3) 4) $\rightarrow$ 4

Because a continuation may now accept zero or more than one value, reified continuations obtained via the procedure *call-with-current-continuation*(*call/cc*) may also accept zero or more than one argument:

(*call-with-values*
  (**lambda** ()
    (*call/cc* (**lambda** (k) (k 2 3))))
  (**lambda** (x y) (*list* x y))) $\rightarrow$ (2 3)

The multiple values proposal leaves unspecified the case in which a continuation expecting exactly one value receives zero values or more than one value. For example, the behavior of each of the following expressions is not specified:

(**if** (*values* 1 2) x y)

(+ (*values*) 5)

Similarly, since there is no requirement in Scheme to signal an error when the wrong number of arguments is passed to a procedure[2], the behavior of each of the following expressions is not specified:

(*call-with-values*
  (**lambda** () (*values* 2 3 4))
  (**lambda** (x y) x))

(*call-with-values*
  (**lambda** () (*call/cc* (**lambda** (k) (k 0))))
  (**lambda** (x y) x))

Each implementor of Scheme's multiple values interface must decide what should happen in the unspecified cases. It is natural for an implementation to handle cases where an unexpected number of values are passed to the consumer in a *call-with-values* call in whatever manner the implementation normally handles cases where procedures receive an unexpected number of arguments. Like most implementations, our implementation signals errors when procedures receive an unexpected number of arguments, and we maintain this semantics for *call-with-values*.

For a continuation expecting a single value, one approach is to ignore the extra values when more than one value is received and to generate a special value when no values are received, as in Common Lisp [8]. Another approach is to ignore extra values but signal an error when no values are received. We feel that either approach tends to mask programming errors without adding significantly to the flexibility of the language. Thus, we have chosen to signal an error both when no values are received and when more than one value is received by a continuation expecting a single value. As shown in the following section, this error checking can be implemented efficiently, with no run-time overhead for single-value calls and returns and minimal overhead for multiple-value calls and returns.

[1] This statement conflicts with the original proposal as worded, which left the behavior in this case unspecified, but subsequent electronic discussions appear to have resulted in a consensus in favor of the behavior described here.

[2] The formal semantics contained in Section 7.2 of the Revised^4 Report signals an error in such cases, but the text of the report does not appear to say whether passing the wrong number of arguments is an error, much less that an error must be signaled.

3 Implementation

The multiple values interface requires that *call-with-values* and *values* be implemented as procedures. With the procedural interface, closures may need to be created for the producer and consumer, and up to four procedure calls must be made, to *call-with-values*, to the *producer*, to *values*, and to the *consumer*. This overhead is especially unfortunate since a major motivation for including a multiple values interface in Scheme is efficiency relative to other techniques for communicating multiple values, *e.g.*, via lists, assignments, or continuation-passing style.

Therefore, when primitive integrations are enabled, our compiler recognizes and optimizes direct calls to both *call-with-values* and *values*. Section 3.1 describes how these calls are converted into internal forms and how the internal forms are further rewritten to eliminate in many cases the unnecessary closures and procedure calls inherent in the procedural interface. After rewriting, some uses of *call-with-values* and *values* are effectively eliminated, and the remaining uses are handled by an adaptation of the procedure call interface.

Section 3.2 describes the procedure call interface along with efficient strategies for verifying correct return value counts. Section 3.3 describes how procedural values for *call-with-values* and *values* are implemented. Section 3.4 describes adjustments necessary to implement first-class continuations in the presence of multiple values and multiple-argument continuations. Section 3.5 describes how variable-arity consumers are supported efficiently. Finally, Section 3.6 describes how our techniques can be adapted to support the Common Lisp multiple values interface.

3.1 Rewriting *values* and *call-with-values*

Calls to *values* are converted into the internal **mv-values** form:

(*values*) $\Rightarrow$ (**mv-values**)

(*values* e) $\Rightarrow$ e

(*values* e_1 e_2 e_3 ...) $\Rightarrow$ (**mv-values** e_1 e_2 e_3 ...)

Calls to *values* with exactly one argument are not converted to **mv-values**, since (*values* e) must be treated the same as e. Calls to *call-with-values* are converted into the internal **mv-call** form:

(*call-with-values producer consumer*) $\Rightarrow$
  (**mv-call** (*producer*) *consumer*)

All uses of **mv-call** are normalized via syntactic rewrites so that the producer is either an expression that evaluates to a single value (such as a constant, variable, **set!** form, or

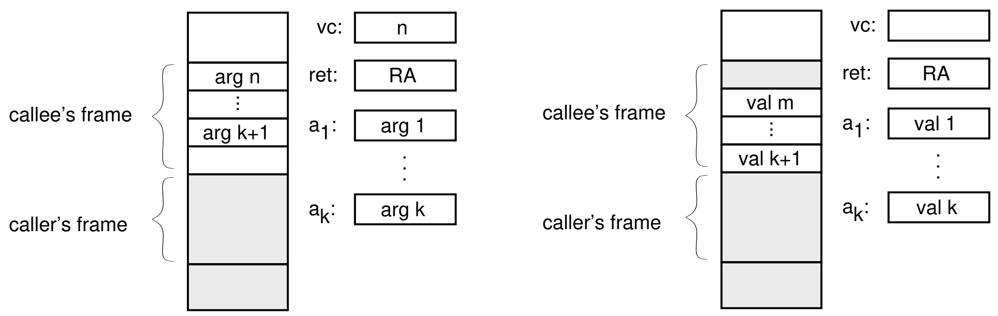

Figure 1. The diagram on the left illustrates the control state just before control transfers to the callee. The diagram on the right shows the control state just before control returns to the caller.

open-coded primitive application), a use of **mv-values**, or an application. This is done by pushing the use of **mv-call** into the producer so that the **mv-call** form occurs in tail position with respect to the producer. In particular, if the *producer* from the original *call-with-values* application is a **lambda** expression (as it often will be), the **mv-call** can be rewritten as follows:

(**mv-call** ((**lambda** () *e*)) *consumer*) $\Rightarrow$
((**lambda** () (**mv-call** *e consumer*)))

or more simply:

(**mv-call** ((**lambda** () *e*)) *consumer*) $\Rightarrow$
(**mv-call** *e consumer*)

When the **mv-call** producer is known to evaluate to a single value or is an **mv-values** expression, the **mv-call** may be rewritten into a simple application:

(**mv-call** e_s *consumer*) $\Rightarrow$
(*consumer* e_s)

(**mv-call** (**mv-values** *e* ...) *consumer*) $\Rightarrow$
(*consumer e* ...)

After these and similar rewrites, only **mv-call** expressions in which the producer expressions are applications remain.

The compiler performs an additional transformation when possible that is analogous to recognizing when a direct **lambda** application is equivalent to a **let** expression:

(**mv-call** *expr* (**lambda** (*id* ...) *body*)) $\Rightarrow$
(**mv-let** *expr* (*id* ...) *body*)

As with **let** expressions, the overhead from creating a closure and making a procedure call is avoided by treating **mv-let** as a local binding operator.

At this point, we can eliminate each occurrence of **mv-values** that is not in tail position with respect to the enclosing **lambda** expression. If it appears in a statement context, we need ensure only that its subexpressions are evaluated for their effects, if any, and can therefore convert the form into a **begin** expression. In any other context, evaluating to zero or more than one value would result in a run-time error, so we are free to replace the form with a **begin** expression that generates the appropriate error message after evaluating the **mv-values** subexpressions. Thus, **mv-values** is reduced to a mechanism for returning zero values or more than one value from a procedure call.

After rewriting, the following forms of **mv-call**, **mv-let**, and **mv-values** remain:

(**mv-call** (e_{proc} e_{arg} ...) *expr*)

(**mv-let** (e_{proc} e_{arg} ...) (*id* ...) *body*)

(**mv-values**)

(**mv-values** e_1 e_2 e_3 ...)

In order to implement these forms as well as normal calls and returns, we must consider the mechanisms by which procedure calls are set up, values are returned, incorrect numbers of values are detected, and values are provided to **mv-call** and **mv-let** consumers.

Terminology: Throughout the remainder of this paper, *multiple-value continuations* are continuations established for the producer by **mv-call** and **mv-let**. *Statement continuations* are continuations that ignore the values passed to them; these continuations are established by **begin** or implicit **begin**. *Single-value continuations* are continuations expecting exactly one value. Although a continuation established by **mv-call** or **mv-let** may accept one value if the consumer accepts one argument, such continuations are nevertheless considered to be multiple-value continuations. A *single-value return* is a return of exactly one value, and a *multiple-value return* is a return of zero values or more than one value.

3.2 Procedure call interface

Our implementation's general procedure call mechanism requires the caller to place the values of the first few (k) argument expressions into argument registers and the remaining values onto the stack, load a "value count" register (vc) with the number of arguments, and transfer control to the code for the procedure. The return address is also passed in a register, but a hole[3] is left at the base of the callee's frame, just below the stack arguments, in which the callee may save the register if the callee itself makes any nontail calls. On return, the first k values are placed in the corresponding argument registers, and the remaining values are placed at the base of the callee's stack frame. (See Figure 1.)

[3] Holes in the frame do not confuse the garbage collector since a live pointer mask is kept behind every return point to indicate which words of the frame must be traced during collection [3].

The code generated for an **mv-call** expression evaluates the consumer expression, saves the resulting value in a temporary stack location at the top of the frame, performs the producer call, and transfers control to the consumer. If the **mv-call** expression is in tail position, the call to the consumer is a tail call, and the stack return values must be shifted to the base of the current frame before control is transferred. (The register values need not be shifted.) The argument count check performed by the consumer suffices to verify that the correct number of values has been returned.

The code generated for an **mv-let** expression performs the producer call and evaluates the **mv-let** body with the **mv-let** variables bound to the register or stack locations that contain the resulting values.

There are a variety of strategies for verifying that the number of values returned matches the number of values expected. We describe three strategies here. The third strategy is the one used by our implementation. The first two may be reasonable choices in some circumstances, and are included for completeness. The first requires all procedure returns to pass back a return count along with the values, and for all procedure call return points to check this count. The second eliminates this overhead for single-value returns into single-value or statement continuations, while adding additional overhead to multiple-value returns. The third also places no run-time overhead on single-value returns into single-value or statement continuations and eliminates most of the overhead for multiple-value returns as well.

3.2.1 Register-based return count

In the first and simplest strategy, all procedure call returns set the value count register *vc* to the number of values returned, and procedure call return points verify that the expected number of values have been received. Return points representing single-value continuations verify that exactly one value has been returned, but otherwise proceed as usual. Return points representing statement continuations, which ignore the returned values, simply proceed as usual.

An **mv-let** return point checks the return count to verify that the appropriate number of arguments has been returned and then proceeds to evaluate its body. An **mv-call** return point simply invokes the consumer. The count of return values returned in *vc* serves as the count of argument values which the consumer uses to verify that the correct number of arguments is received. If the call to the consumer is not a tail call, the temporary location used to store the consumer becomes the first word of the consumer's frame, *i.e.*, the location used by the consumer to store its return address if necessary.

This strategy is simple to understand and implement, but it suffers from the drawback that single-value calls and returns are slowed by the setting and checking of the return count register.

3.2.2 Stack-based return count

In the second strategy, full responsibility for error checking is placed on **mv-call**, **mv-let**, and multiple-value returns.

A flag is placed in the instruction stream behind every return point indicating whether or not multiple return values are accepted. This flag is set to true for both multiple-value continuations and statement continuations, and false for single-value continuations. In the first two cases, the stack location directly below the return address is reserved to hold the count of values returned. Both **mv-let** and **mv-call** arrange to seed this slot with the value 1.

Code for a multiple-value return checks the flag in the code stream behind the return point to insure that multiple values are accepted. If not, an error is signaled. Otherwise, the return value count is placed in the return count slot below the return address (rather than in the *vc* register as before) before control returns to the caller. Code for a single value simply returns without changing the return count slot. The return count slot need not be set in this case since it is seeded with 1 for multiple-value continuations and ignored by statement continuations.

Return points representing single-value and statement continuations proceed without any checks. Single-value continuations are guaranteed to receive a single value, and statement continuations do not care how many values are returned. An **mv-let** return point retrieves the return count from the stack, verifies that the appropriate number of arguments has been returned, and evaluates the body. An **mv-call** return point retrieves the consumer from its temporary location, moves the return value count from the stack into *vc*, and transfers control to the consumer. If the call to the consumer is not a tail call, the location used to indicate the number of return values becomes the location used to store the consumer's return address.

While this strategy is more complicated, it has the advantage that the overhead for handling multiple return values has been eliminated for normal calls and returns. The drawback is that the overhead for handling multiple values has increased. Multiple-value returns must check a flag before returning and must write the return count to memory rather than to a register. Multiple-value continuations must load the count from memory.

3.2.3 Separate multiple-value return point

Our final strategy improves the performance of multiple-value returns and continuations without sacrificing the speed of normal calls and returns. In place of the flag in the second model, an alternate multiple-value return point is situated at a fixed offset behind the normal (single-value) return point in the instruction stream. Procedure calls returning one value return to the single-value return point. Calls returning zero values or more than one value place the number of values being returned into *vc* and return to the multiple-value return point.

If control returns to the single-value return point of a single-value continuation, a single value must have been returned and execution proceeds without any check. If control returns to the alternate return point of a single-value continuation, a jump is made to an error routine signaling that multiple values were returned to a single-value continuation.

For statement continuations, the single-value return point again performs no check, and the multiple-value return point simply falls through to the single-value return point, achieving the desired effect that any number of values returned are ignored.

If control returns to the single-value return point of an **mv-call**, *vc* is loaded with the value 1 and control is transferred to the consumer. The code at the multiple-value return point jumps around the load of *vc* with 1, directly to the code that transfers control to the consumer.

The single-value return point of an **mv-let** can simply load *vc* with 1, as in **mv-call**, and fall through to the code for the return count check and **mv-let** body. Alternatively,

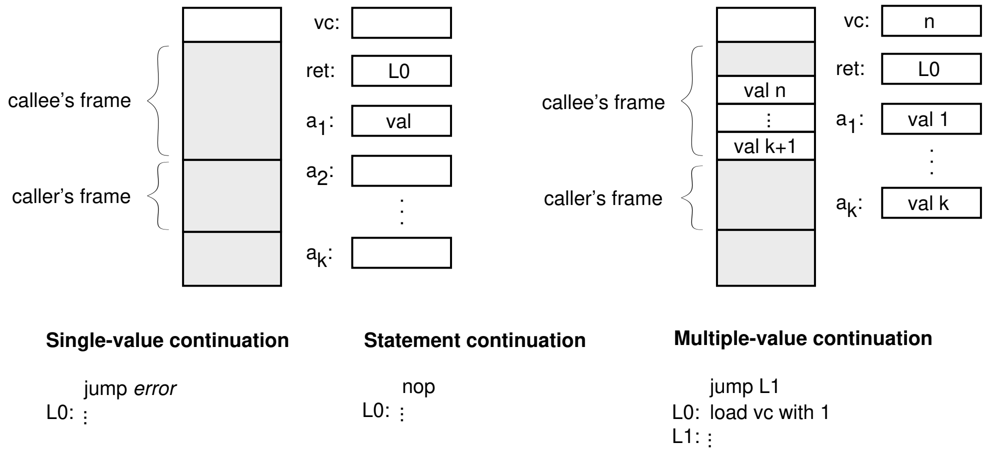

Figure 2. There is now a multiple-value return point behind each normal (single-value) return point. The diagram on the left illustrates the control state just before a single-value return. The diagram on the right shows the control state just before a multiple-value return.

the single-value return point can jump directly to the body (if one value is accepted) or to the error signaling code. The multiple-value return point of an **mv-let** jumps over the single-value return point to the return count check.

Figure 2 illustrates the stack layouts for both multiple- and single-value returns along with the code sequences found at the various return points.

The virtues of this model are that normal calls and returns proceed without overhead, and no checks are required to prevent multiple-value returns into single-value continuations. It is tempting to "optimize" the mechanisms in the previous models by skipping the error checks necessary to verify that a single value was returned to a single value continuation, but this model has no test to eliminate. Signaling errors when possible even in optimized code is an essential part of our design philosophy, and preserving the error check at no extra cost allows us to remain consistent with this philosophy without sacrificing efficiency.

3.3 Procedural versions of *values* and *call-with-values*

The compiler can rewrite uses of *call-with-values* and *values* when they are applied directly to arguments, but we must still provide procedural definitions for them. Once the compiler recognizes direct calls to *call-with-values*, a procedural definition of *call-with-values* is trivial:

```
(define call-with-values
  (lambda (x y)
    (call-with-values x y)))
```

Although this definition appears to be circular, the use of *call-with-values* is recognized by the compiler and rewritten into the corresponding **mv-call** form.

A trivial definition for *values*:

```
(define values
  (lambda args
    (apply values args)))
```

would be circular since the compiler recognizes only direct calls to *values*. We cannot list the arguments individually and call *values* directly since *values* accepts an indefinite number of arguments. We thus implement *values* as a primitive library routine. This routine first moves the stack arguments, if any, to the base of the frame, leaving the register arguments in place. It then transfers control to the normal (single-value) return point if one argument is received ($vc = 1$) and to the multiple-value return point otherwise. The argument count in *vc* serves as the return value count in the latter case.

3.4 Multiple values and first-class continuations

Our strategy fits cleanly with our implementation of first-class continuations [7]. When a continuation is captured, the current stack segment is split into two smaller segments. The first segment is the part of the stack in use when the continuation is captured, and the second segment is the unused portion of the stack. A distinguished return address, the address of a continuation handler, is placed at the base of the second segment; this handler effects an implicit continuation invocation if control returns through it. The displaced return address is placed in a data structure that also links the two segments and maintains other bookkeeping information. Execution continues using the second stack segment as the current stack segment.

Whether a captured continuation is invoked explicitly or implicitly, control is transferred to one of two entry points into the continuation handler: the single-value entry point or the multiple-value entry point. Since the handler serves as an artificial return point, the multiple-value entry point is at the same fixed offset behind the single-value entry point as for ordinary procedure call return points. The procedure representing a continuation branches to the appropriate entry depending upon the number of arguments received.

The handler copies the saved stack segment[4] into the

[4] The segment to be copied may be split before it is copied if its size exceeds a predetermined limit on the size of segments the continuation

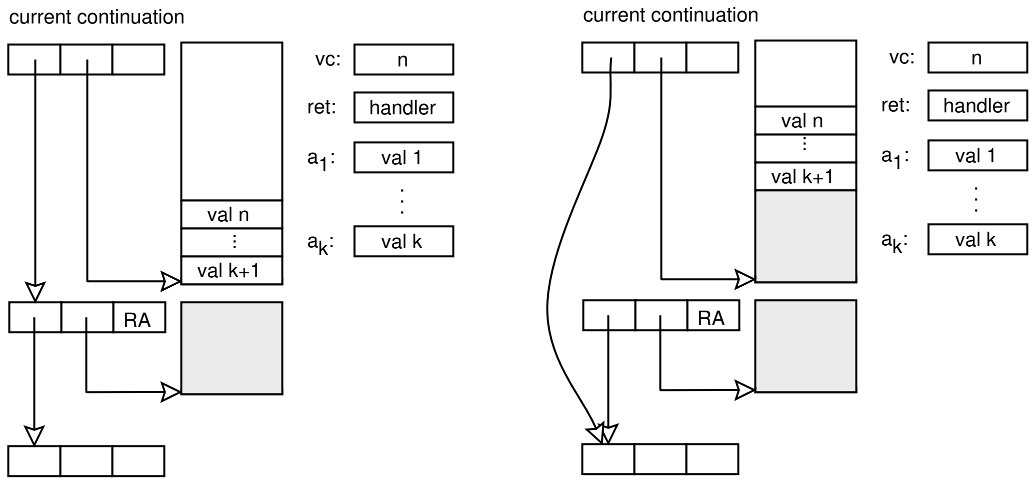

Figure 3. These diagrams show the control state just before and after an implicit continuation invocation. When a continuation is invoked, the saved stack segment is copied to the current stack segment after the stack values (if any) are moved to their new locations.

current segment and returns to the displaced return address. The only difference between the multiple- and single-value entries into the continuation handler is that the multiple-value entry must first move the stack return values to just beyond where the saved stack segment will be inserted into the current segment. (See Figure 3.)

3.5 Variable-arity consumers

Scheme permits procedures to accept arbitrary numbers of arguments through its "dot" interface. It is possible to extend **mv-let** to accommodate the dot interface. In place of a simple check to verify that the exact number of values expected has been received, **mv-let** verifies that the minimum number expected has been received and packages the remaining arguments in a list. This packaging can be performed out-of-line to reduce code size.

Our implementation supports a multiple-arity procedure call interface with **case-lambda** [4, 12]. A **case-lambda** expression takes the following form:

(**case-lambda** (*idspec body*) ...)

Each clause of a **case-lambda** expression specifies an interface to the procedure and the expression to be evaluated when the procedure is called through that interface. For example:

```
(let ((f (case-lambda
           ((x) (+ x 1))
           ((x . xs) (cons (+ x 2) xs))
           (xs 0))))
  (list (f 5) (f 1 2 3) (f)))         → (6 (3 2 3) 0)
```

The particular interface chosen for an application is the first clause that accepts the number of arguments provided. Any **lambda** expression can be rewritten into a **case-lambda** expression with only one clause.

handler is allowed to copy [7].

Our system allows **case-lambda** to be used to specify the consumer when using *call-with-values*. In order to allow the consumer to be a **case-lambda** expression, we must extend **mv-let** to handle multiple consumers. The revised syntax for **mv-let** is:

(**mv-let** *expr* ($idspec_0$ $body_0$) ... ($idspec_n$ $body_n$))

When **mv-call** appears with a **case-lambda** expression as the consumer, the compiler rewrites the expression as follows:

```
(mv-call expr
  (case-lambda                 (mv-let expr
    (idspec_0 body_0) ...   ⇒    (idspec_0 body_0) ...
    (idspec_n body_n)))          (idspec_n body_n))
```

The implementation of this extended form of **mv-let** is almost identical to the single clause case. The only difference is that a case dispatch on the number of values received must be performed to determine which clause to execute. As with **case-lambda**, the first clause that accepts the number of values provided is chosen as the consumer.

The optional argument interfaces provided by Common Lisp [8] and by other Scheme systems, *e.g.*, MIT Scheme [6], can be handled in a similar manner.

3.6 Common Lisp multiple values interface

Our strategy can be adapted to implement the Common Lisp multiple values interface efficiently. Common Lisp provides two basic interfaces for receiving multiple values: **multiple-value-call** and **multiple-value-bind**. The former is essentially equivalent to our internal **mv-call** form, except that the consumer arguments are constructed from values received from an arbitrary number of producers. The latter is essentially equivalent to our **mv-let** form, except that extra return values are ignored and the variables corresponding to missing values are set to *nil*.

In order to speed the handling of multiple-value returns

	time	*space*
mvlet	1.00	1.00
cps	1.21	2.00
cons	1.26	1.60
byref	1.20	1.10
reverse	1.53	2.10

Table 1.

into single-value continuations, zero value returns should place *nil* in the first return value location in addition to setting the return value count to 0. The multiple-value return point for a single-value continuation may then simply fall through to the single-value return point, as for statement continuations. The return points for a **multiple-value-call** with one producer behave the same as for our **mv-call** form; handling more than one producer is a straightforward generalization but may require some shuffling of values. Both the single- and multiple-value return points for a multiple-value continuation established by **multiple-value-bind** must generate one *nil* value for each return value expected by the consumer beyond those actually returned. This may be accomplished by a dispatch to the appropriate point in a sequence of move instructions inserted before the consumer code. For single-value entries, the dispatch can be replaced by a direct jump to a fixed address within the sequence of move instructions.

The minor savings relative to a similar adaptation of the strategy described in Section 3.2.1 may not compensate for the added complexity of providing two return points. The primary benefit derived from our model when full error checking is performed comes from eliminating the return count test for single-value continuations. No such check is required, however, by either of the strategies to implement the Common Lisp semantics. Furthermore, on modern pipelined architectures, the cost of moving the return value count into the *vc* register, as required by the simpler strategy, is likely to cost little or nothing.

4 Performance

The multiple values implementation has been incorporated into the *Chez Scheme* compiler and run-time system. The implementation performs well, especially when calls to *call-with-values* can be recognized internally as **mv-let** forms. Table 1 compares the performance of several versions of the *split* procedure defined in Section 2. The different variations are described below:

mvlet: values returned via *values* (original version)
cps: values returned via continuation-passing style
cons: values returned via *cons*
byref: values returned via assignments
reverse: values accumulated in reverse, then reversed

We call the original version "*mvlet*" since the use of *call-with-values* is ultimately converted into the internal **mv-let** internal form. The *reverse* version is included for completeness, but does not represent a generally applicable style for handling multiple values. Definitions for all but the original version are given in the appendix.

	time	*space*
mvlet	1.00	1.00
mvcall	1.27	1.50
procedural	1.94	2.00

Table 2.

Both cpu time and allocation costs are given, normalized to the values for the *mvlet* version. Lists of length 10 were used as input, and the tests were timed over many iterations to produce measurable data. The table shows that the performance of the multiple values interface for this simple benchmark is significantly better in terms of both cpu time and allocation costs.

As a more realistic benchmark, we rewrote three passes of our compiler that require the use of multiple values to use the new interface in place of a vector-based interface that we had been using. All occurrences of *call-with-values* are recognizable as **mv-let** forms internally. This resulted in a 20% improvement in the speed of the three passes.

We also compared the performance of a set of benchmarks that do not use multiple values (Scheme versions of the Gabriel benchmarks [5]) before and after the multiple value interface was added to the system. As expected, we found that supporting multiple values has no run-time impact on code that does not use them, although the code generated is naturally slightly larger due to the addition of multiple-value return points.

We also studied what happens when the compiler is not able to convert calls to *call-with-values* into the internal **mv-let** form. Table 2 compares the performance of the original *mvlet* version with the following variants:

mvcall: recognition of consumer as a **lambda** expression disabled
procedural: recognition of direct calls to *call-with-values* and *values* disabled

As the table shows, the *mvlet* version has a significant edge. Comparing Tables 1 and 2, it also appears that programmers are better off, in terms of performance, using some other mechanism for returning multiple values unless the compiler can recognize the *call-with-values* application.

5 Related work

Other implementations of Scheme, notably T [13] and MIT Scheme [6], provide multiple values interfaces. We have experimented with T Version 3.1 on a Sparc processor and MIT Scheme Version 7.2 on an SGI MIPS processor. The T interpreter's semantics for multiple return values matches ours, but the compiler does not perform the run-time checks necessary to guarantee that the correct number of values is received. Their compiler seems to optimize only cases where both the producer and consumer are lambda expressions; in other cases they use the procedural interface. In MIT Scheme, the semantics for multiple return values is similar to ours, but their current implementation appears to require that *values* be used only to return to continuations created by *call-with-values*. Their system also does not appear to treat the case where the consumer is a **lambda** expression as a local binding operation.

The model described in Section 3.2.1, with the modification for zero-value returns mentioned in Section 3.6, is essentially the same as the model used by Kyoto Common Lisp [14]. Rather than intermix stack arguments with other control information on a single stack, however, they use multiple stacks with one stack reserved for passing and returning values to procedure calls. Also, their compiler targets C and therefore does not have access to the machine registers, so they use the stack for all arguments and return values.

On Sparc-based computer systems, the prescribed method for communicating whether fixed-length structures are expected and returned in C and similar languages somewhat resembles the stack-based return count model described in Section 3.2.2 [11]. The return point for a call to a procedure expected to return a structure is flagged with a particular "unimplemented" instruction, and the address of a location into which to place the returned structure is passed as a special argument on the stack. A procedure returning a structure must check to determine if the unimplemented instruction is at the return point. If it is, it copies the return value to the specified address. In either case, control returns to the instruction following the unimplemented instruction. A procedure returning a nonstructure value returns as usual, resulting in an unimplemented instruction trap.

The Spineless Tagless G-machine [9] uses vectored returns to control closure updating. Control is returned to one point if the closure needs to be updated and to another return point if it does not. Also, if the result is a constructor, and if the constructor has sufficiently few fields, the fields are returned in registers instead of in a data structure. The effect in this case is a multiple-value return to the continuation using a mechanism similar to ours.

6 Conclusions

In this paper we have described an implementation of Scheme's multiple values interface. The implementation converts all direct calls to *call-with-values* and *values* into internal forms that can be optimized by the compiler. These forms are rewritten to eliminate uses of multiple values when the producer is a **lambda** expression whose body evaluates to one value or to multiple values via a direct call to *values*. The implementation handles arbitrary consumer expressions efficiently but optimizes the common case in which the consumer is a **lambda** expression by treating the consumer call as a local binding operation, as with **let**. The first few values returned from a procedure call are placed in the same registers as the first few argument values passed to procedures, and the remainder are stored on the stack as for procedure calls. This helps make multiple-value returns efficient and simplifies the communication of multiple values to unknown consumers.

Full error checking for unexpected numbers of values is performed with no run-time overhead for normal procedure calls, *i.e.*, single-value returns to single-value and statement continuations, and little overhead for multiple-value procedure calls. This is accomplished with the use of a separate multiple-value return point placed at a fixed offset behind the normal (single-value) return point for each procedure call.

Programs that wish to ignore extra values in particular contexts can do so easily by calling *call-with-values* explicitly. The syntactic form **first** defined below abstracts the discarding of more than one value when only one is desired:

```
(define-syntax first
  (syntax-rules ()
    ((_ expr)
     (call-with-values
       (lambda () expr)
       (lambda (x . y) x)))))
```

Ignoring values in this manner is inexpensive in our implementation because of our treatment of *call-with-values* and because our implementation does not construct lists for unreferenced "rest" variables. The code generated simply verifies that at least one value has been returned and ignores the remaining values.

It is possible to implement the multiple values interface entirely in Scheme, although doing so precludes doing something sensible when a single value continuation receives zero values or more than one value. Such an implementation is also certain to be much less efficient than a native implementation of the interface. The code below demonstrates one way in which this can be done:

```
(define call-with-current-continuation)
(define values)
(define call-with-values)
(let ((magic (cons 'multiple 'values)))
  (define magic?
    (lambda (x)
      (and (pair? x) (eq? (car x) magic))))
  (set! call-with-current-continuation
    (let ((primitive-call/cc
            call-with-current-continuation))
      (lambda (p)
        (primitive-call/cc
          (lambda (k)
            (p (lambda args
                 (k (apply values args)))))))))
  (set! values
    (lambda args
      (if (and (not (null? args)) (null? (cdr args)))
          (car args)
          (cons magic args))))
  (set! call-with-values
    (lambda (producer consumer)
      (let ((x (producer)))
        (if (magic? x)
            (apply consumer (cdr x))
            (consumer x))))))
```

The special flag (*magic*) used to mark multiple values can be any unique Scheme object. When anything other than a single value is returned to a single value continuation, the continuation receives a list whose car is this flag. Without altering the compiler or macro expander's treatment of applications, it is not possible to signal an error or even to ignore extra values in this case.

As with first-class continuations, multiple return values can be implemented via a global source-level rewrite into continuation-passing style (CPS). Just as implementing first-class continuations in this manner prevents the compiler from assuming that all continuations have dynamic extent, implementing multiple return values in this manner prevents the compiler from assuming that all continuations receive exactly one value. CPS compilers that wish to flag return value count mismatches must therefore verify that the correct number of values is passed to each continuation. The same technique used to implement multiple values in a direct-style compiler such as ours may be used in a CPS compiler: the multiple-value return point behind each normal return point becomes a multiple-value entry point behind each normal continuation entry point.

A compiler might perform a CPS conversion early in the compilation process to simplify subsequent processing, then convert back into direct style just prior to code generation. The reverse CPS transformation developed by Danvy and Lawall [2] to recognize transformed calls to *call-with-current-continuation* and reified continuations should be straightforward to extend to recognize transformed calls to *call-with-values* and *values*.

Unoptimized multiple-value calls may require as many as two additional closures and four additional procedure calls. In implementations that do not optimize the interface when possible, code using the interface will likely run slower than code using other mechanisms for returning multiple values. Unless most implementations optimize the interface, *call-with-values* will be of little practical use as a portable tool for performance enhancement.

Because the interface is procedural rather than syntactic, the code that produces the values must be encapsulated in a zero-argument procedure. As well as leading to awkward-looking code, this has the effect of forcing the consumer expression to be evaluated before the values are computed. This artificial ordering constraint does not exist for ordinary calls: the consumer (procedure expression) may be evaluated either before or after the argument values are computed. The constraint can result in the generation of suboptimal code.

Because of the awkwardness, potential inefficiency, and artificial ordering constraint inherent in the procedural interface, we believe it would have been better to have standardized on a syntactic version of *call-with-values*, which we might call **with-values**:

(**with-values** *producer consumer*)

In this form, the producer is an expression evaluating to zero or more values, and the consumer is an expression evaluating to a procedure that accepts the number of values yielded by the producer. Nothing would be lost with this change: *call-with-values* can be defined trivially in terms of **with-values**.

References

[1] William Clinger and Jonathan A. Rees (Editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[2] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 299–310, 1992.

[3] R. Kent Dybvig, David Eby, and Carl Bruggeman. A segmented memory model for storage management in Scheme. Technical Report 400, Indiana University Computer Science Department, Lindley Hall 215, Bloomington Indiana, 47405, 1994.

[4] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, September 1990.

[5] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1985.

[6] Chris Hanson. *MIT Scheme Reference Manual*, November 1991. Edition 1.1 for Scheme Release 7.1.3.

[7] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[8] Guy L. Steele Jr. *Common Lisp*. Digital Press, second edition, 1990.

[9] Simon L. Peyton-Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[10] Jonathan Rees. The Scheme of things: The June 1992 meeting. *Lisp Pointers*, 4(4), October-December 1992.

[11] *System V Application Binary Interface, SPARC Processor Supplement*, 1990.

[12] Cadence Research Systems. *Chez Scheme System Manual*. Bloomington, Indiana, February 1992.

[13] *T Version 3.0 Release Notes*.

[14] Tahchi Yuasa. Design and implementation of Kyoto Common Lisp. *Journal of Information Processing*, 13(3):284–295, 1990.

A Variants of *split* used for performance comparisons

```
(define split-cps
  (lambda (ls values)
    (if (or (null? ls) (null? (cdr ls)))
        (values ls '())
        (split-cps (cddr ls)
          (lambda (odds evens)
            (values (cons (car ls) odds)
                    (cons (cadr ls) evens)))))))

(define split-cons
  (lambda (ls)
    (if (or (null? ls) (null? (cdr ls)))
        (cons ls '())
        (let ((pair (split-cons (cddr ls))))
          (cons (cons (car ls) (car pair))
                (cons (cadr ls) (cdr pair)))))))
```

```
(define split-byref
  (lambda (ls pair)
    (if (or (null? ls) (null? (cdr ls)))
        (begin (set-car! pair ls)
               (set-cdr! pair '()))
        (begin (split-byref (cddr ls) pair)
               (set-car! pair
                 (cons (car ls) (car pair)))
               (set-cdr! pair
                 (cons (cadr ls) (cdr pair)))))))

(define split-reverse
  (lambda (ls)
    (let f ((ls ls) (odds '()) (evens '()))
      (cond
        ((null? ls)
         (cons (reverse odds) (reverse evens)))
        ((null? (cdr ls))
         (cons (reverse (cons (car ls) odds))
               (reverse evens)))
        (else (f (cddr ls)
                 (cons (car ls) odds)
                 (cons (cadr ls) evens)))))))
```