LISP AND SYMBOLIC COMPUTATION: An International Journal 3:3: 229–244, 1990

A New Approach to Procedures with Variable Arity*

R. KENT DYBVIG (*dyb@iuvax.cs.indiana.edu*)
ROBERT HIEB (*hieb@iuvax.cs.indiana.edu*)
Indiana University Computer Science Department
Bloomington, IN 47405

(Received: September, 1989)
(Revised: January, 1990)

Keywords:

Abstract. This article presents a convenient and efficient procedural interface that allows the definition and use of procedures with optional arguments and indefinite numbers of arguments without resorting to the use of a language-dependent data structure in which to store the arguments. This interface solves many of the problems inherent in the use of lists in Lisp and Scheme to store indefinite numbers of arguments. Simple recursion can be used to consume such arguments without the complexity problems that are caused by the use of the Lisp procedure `apply` on argument lists. A natural extension to the interface to support multiple return values is presented.

1 Introduction

Many programming languages provide primitive procedures that are defined for variable numbers of arguments. Typically, however, the programmer is not provided with a convenient way to create new variable-arity procedures. Although Common Lisp [6] and Scheme [5] both allow the programmer to define variable-arity procedures, the resulting definitions are often unreadable or inefficient. Furthermore, the arguments to procedures that accept an indefinite number of arguments are packaged in a list; this commitment to a particular data structure reduces the generality of the mechanism and complicates the semantics of the procedural interface.

There are two broad classes of variable-arity procedures: (1) procedures that accept an indefinite number of arguments, and (2) procedures that accept a limited but variable number of arguments. The second category can

*This material is based on work supported by the National Science Foundation under grant number CCR-8803432 and by Sandia National Laboratories under contract number 06-06211. A preliminary version of this article was presented at the 1988 ACM Conference on Lisp and Functional Programming.

be further divided into two subclasses: (a) procedures that have optional arguments with default values, and (b) procedures that perform related but distinct actions depending upon how many arguments they receive. In this article we describe a procedural interface that handles each of these classes in a convenient and efficient manner, without resorting to storing the arguments in a language-dependent data structure. We also describe a natural extension of this interface to handle multiple return values.

In a 1965 paper, Landin calls ISWIM "an attempt to deliver Lisp from its eponymous commitment to lists" [3]. Although the concept of Lisp without lists seems paradoxical, most Lisp dialects provide alternative data structures and mechanisms for defining new data structures. It should be possible, ideally, to create a dialect of Lisp without lists. We feel that it is an important feature of our proposal, therefore, that the procedural interface does not depend on the list data structure.

Some languages have gone to the extreme of totally immersing the procedural interface in the data structures of the language. For instance, Hewitt's PLASMA supports a mechanism superficially similar to ours, but defines all procedures to accept one argument. The argument may be an arbitrary list structure that the target procedure decomposes by pattern matching [2]. However, a more general solution is to proceed in the opposite direction, to totally divorce the fundamental control structures of a language from data structures and operations on those data structures. In clarifying the philosophy behind ISWIM, Landin states, "Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things... [ISWIM] is a byproduct of an attempt to disentangle these two aspects... So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives."

When a data structure becomes part of the procedural interface other problems arise due to the interaction between operations on the data structure and the interface. We shall see that the combination of variable-arity procedures and lists in Lisp is not well designed, and can lead to apparently elegant programs with severe performance problems. Furthermore, the implementation of the variable-arity interface is restricted, since the implementor is not always free to choose the most appropriate internal representation. Divorcing the procedural interface from a concrete internal representation allows for significant optimizations in common instances.

We begin by describing the syntax and semantics of our new procedural interface, treating it as an extension to Scheme. We then devote a section to programming examples; these examples demonstrate the new interface and show that it leads to elegant solutions for common programming problems. We follow this with a discussion about implementation problems and

$$
\begin{aligned}
\lambda^*\textit{-expression} &\rightarrow (\textbf{lambda*}\ \textit{clause clause}\ \ldots) \\
\textit{clause} &\rightarrow [\textit{formals expression}] \\
\textit{formals} &\rightarrow (\textit{variable}\ \ldots) \mid (\textit{variable}\ \ldots\ \textbf{\&}\ \textit{rest-variable}) \\
\textit{procedure-call} &\rightarrow (\textit{expression expression}\ \ldots) \\
&\mid\ (\textit{expression expression}\ \ldots\ \textbf{\&}\ \textit{rest-variable}) \\
\textit{expression} &\rightarrow \lambda^*\textit{-expression} \mid \textit{procedure-call} \mid \textit{variable}
\end{aligned}
$$

Figure 1: Syntax for λ^*.

strategies. We then describe how the syntax and semantics of the interface can be extended to allow multiple return values. In the final section we make some concluding remarks, compare our proposal with other approaches, and discuss possible extensions to the interface.

2 Syntax and Semantics

Our procedural interface requires the introduction of a new syntactic form, λ^*, and extension of the application syntax of Scheme. The grammar in Figure 1 defines the syntactic rules for λ^* and the extended application syntax. Although a complete semantics for Scheme with the extended procedural interface is outside of the scope of this article, Figure 2 provides a formal semantics for the subset expressed by the grammar in Figure 1. The generalization to a semantics with continuations, stores, and the extended value domain necessitated by a full treatment of Scheme is straightforward. A description of the syntax and semantics of Scheme can be found in [5]. The significant feature of the semantics for λ^* is that environments map variables to sequences of values rather than to single values.

A λ^* expression with only one clause is comparable to a Scheme λ expression and evaluates to a procedure. When such a procedure is applied, its formals are bound to the actual parameters, and the corresponding expression is evaluated in the new environment containing these bindings along with the bindings in effect when the procedure was created. For example, `(lambda* [(x) x])` defines the identity function. When more than one clause is present, the first clause whose formal parameters accept the actual parameters on a given call is applied. Each clause in a λ^* expression may be thought of as a separate procedure. A clause whose formals specification is of the form $(x_1 \ldots x_n)$ accepts exactly n actual parameters. A clause whose formals specification is of the form $(x_1 \ldots x_n \,\&\, r)$ accepts any number of actual parameters greater than or equal to n. All parameters in excess of n are bound to r; r is referred to as a *rest variable*, and the parameters bound to r are referred to as *rest values*. It is an error if no clause accepts the actual parameters, and the run time system should trap the error and invoke an appropriate exception handler.

Rest values are accessed by passing them to a procedure using the extended application syntax. A rest variable can refer to zero or more values. If r is a rest variable and refers to zero values, then an application $(e_1\, e_2 \ldots\, e_n\, \mathbf{\&}\, r)$ is equivalent to $(e_1\, e_2 \ldots\, e_n)$. In general, if r refers to values $v_1 \ldots v_m$, then $(e_1\, e_2 \ldots\, e_n\, \mathbf{\&}\, r)$ is equivalent to $(e_1\, e_2 \ldots\, e_n\, v_1 \ldots v_m)$. Rest variables can appear only after an ampersand in the final position of a procedure call, and only rest variables are permitted to appear after ampersands.

Since access to the rest values is allowed only through procedure calls, we can guarantee that rest values are protected against side effects and that they are passed by procedure calls without allocating new storage locations. We shall see that this latter feature is necessary to ensure that the apparent time and space complexity of an algorithm is not altered by the implementation of the procedural interface.

3 Programming Examples

In this section we provide some simple programming examples to exhibit the power and elegance of the λ^* construct. In the process, we compare λ^* solutions with those possible in Scheme and Common Lisp.

In Scheme, I/O procedures typically take an optional port argument. The call `(read-char)` returns a character from the "current input port," while the call `(read-char p)` reads a character from the port `p`. Without the ability to define variable-arity procedures we would either have to provide two procedures, say `read-char` and `port-read-char`, or force the programmer to always supply the port. Either alternative places a burden on the programmer, who must deal with extra procedure names or extra procedure arguments. Using λ^*, it is straightforward to provide procedures that accept optional arguments:

```
(define read-char
  (lambda*
    [(p) (port-read-char p)]
    [() (port-read-char (current-input-port))]))
```

We can take advantage of optional arguments to combine the Scheme `string-copy` and `substring` procedures. The procedure call `(string-copy s)` returns a copy of the string `s`, and `(substring s start end)` returns a copy of the section of `s` from `start` to `end`. Assuming `substring` always returns a

Syntactic variables:

$$e \in \mathit{expressions}$$
$$x \in \mathit{variables}$$
$$r \in \mathit{rest\text{-}variables}$$
$$f \in \mathit{formals}$$

Domains:

$$v \in \mathit{values} = \mathit{values}^* \to \mathit{values}$$
$$\epsilon \in \mathit{values}^*$$
$$\rho \in \mathit{environments} = \mathit{variables} + \mathit{rest\text{-}variables} \to \mathit{values}^*$$

Semantics Functions:

$$\begin{aligned}
\mathcal{E} &: \mathit{expressions} \to \mathit{environments} \to \mathit{values} \\
\mathcal{M} &: \mathit{formals} \to \mathit{values}^* \to \mathit{booleans} \\
\mathcal{R} &: \mathit{environments} \to \mathit{formals} \to \mathit{values}^* \to \mathit{environments}
\end{aligned}$$

Semantic equations:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho &= (\rho x)\downarrow 1 \\
\mathcal{E}[\![(e\, e_1 \ldots\, e_n)]\!]\rho &= \mathit{strict}(\mathcal{E}[\![e]\!]\rho)\langle(\mathcal{E}[\![e_1]\!]\rho), \ldots, (\mathcal{E}[\![e_n]\!]\rho)\rangle \\
\mathcal{E}[\![(e\, e_1 \ldots\, e_n \;\&\; r)]\!]\rho &= \mathit{strict}(\mathcal{E}[\![e]\!]\rho)(\langle(\mathcal{E}[\![e_1]\!]\rho), \ldots, (\mathcal{E}[\![e_n]\!]\rho)\rangle \S (\rho r)) \\
\mathcal{E}[\![(\textbf{lambda*}\, [f_1\, e_1] \ldots\, [f_n\, e_n])]\!]\rho &= \lambda\epsilon\,.\, \mathcal{M}[\![f_1]\!]\epsilon \to \mathcal{E}[\![e_1]\!](\mathcal{R}\rho[\![f_1]\!]\epsilon)\,, \\
&\qquad \vdots \\
&\qquad \mathcal{M}[\![f_n]\!]\epsilon \to \mathcal{E}[\![e_n]\!](\mathcal{R}\rho[\![f_n]\!]\epsilon)\,, \\
&\qquad \text{error} \\
\mathcal{M}[\![(x_1 \ldots\, x_m)]\!]\langle v_1, \ldots, v_n\rangle &= (m = n) \\
\mathcal{M}[\![(x_1 \ldots\, x_m \;\&\; r)]\!]\langle v_1, \ldots, v_n\rangle &= (m \leq n) \\
\mathcal{R}\rho[\![(x_1 \ldots\, x_n)]\!]\langle v_1, \ldots, v_n\rangle &= \rho[\langle v_1\rangle/x_1] \ldots\, [\langle v_n\rangle/x_n] \\
\mathcal{R}\rho[\![(x_1 \ldots\, x_n \;\&\; r)]\!]\langle v_1, \ldots, v_n, v_{n+1}, \ldots\rangle &= \rho[\langle v_1\rangle/x_1] \ldots\, [\langle v_n\rangle/x_n][\langle v_{n+1}, \ldots\rangle/r]
\end{aligned}$$

Figure 2: Semantics for λ^*.

new string, `string-copy` is redundant, since `(string-copy s)` is the same as `(substring s 0 (string-length s))`. The only reason to have `string-copy` in the language is to allow the programmer to avoid providing the additional arguments. By allowing `substring` to provide defaults for missing arguments (as in Common Lisp), `string-copy` can be omitted from the language:

```
(define substring
  (lambda*
    [(s start end)
     (let ([s2 (make-string (- end start))])
       (do ([j 0 (+ j 1)] [i start (+ i 1)])
           ((= i end) s2)
           (string-set! s2 j (string-ref s i))))]
    [(s start) (substring s start (string-length s))]
    [(s) (substring s 0 (string-length s))]))
```

It is not necessary for `substring` to assume that the third argument is missing when only two arguments are provided. We could just as easily define a version of `substring` that supplies a default value for the second argument in that case:

```
(define substring
  (lambda*
    [(s start end)
     (let ([s2 (make-string (- end start))])
       (do ([j 0 (+ j 1)] [i start (+ i 1)])
           ((= i end) s2)
           (string-set! s2 j (string-ref s i))))]
    [(s end) (substring s 0 end)]
    [(s) (substring s 0 (string-length s))]))
```

Alternatively, we can prevent confusion by requiring neither or both of the endpoints:

```
(define substring
  (lambda*
    [(s start end)
     (let ([s2 (make-string (- end start))])
       (do ([j 0 (+ j 1)] [i start (+ i 1)])
           ((= i end) s2)
           (string-set! s2 j (string-ref s i))))]
    [(s) (substring s 0 (string-length s))]))
```

A procedure in which the first argument rather than the second argument can be considered optional is "-" defined as both a unary and binary procedure:

```
(define -
  (lambda*
    [(x y) (binary- x y)]
    [(x) (binary- 0 x)]))
```

It is interesting to compare this with Scheme and Common Lisp versions of "-". In Scheme, a procedure must take optional arguments in a list and destructure the list to obtain their values:

```
(define -
  (lambda (x . l)
    (cond
      [(null? l) (binary- 0 x)]
      [(null? (cdr l)) (binary- x (car l))]
      [else (error '- "too many arguments")])))
```

Here the list destructuring and explicit error handling make the code difficult to follow and make generation of efficient code difficult. Although Common Lisp provides an optional argument mechanism that makes it unnecessary to package all optional arguments in lists, it does not easily handle "-", since its mechanism is oriented toward defaulting trailing arguments. In order to determine whether two arguments have been supplied, a *supplied-predicate parameter* as well as an *initialization form* must be provided, obscuring the intent and effect of the code:

```
(defun - (x &optional (y 0 pred))
  (if pred
      (binary- x y)
      (binary- y x)))
```

We leave it as an exercise for the reader to convert the above versions of `substring` to Scheme and Common Lisp. Although Common Lisp handles the first version in a straightforward manner, solutions to the latter two that do full error checking are inelegant in both Scheme and Common Lisp.

So far our examples have not dealt with procedures that accept indefinitely many arguments. Consider the following definition for "+":

```
(define +
  (lambda*
    [() 0]
    [(x & r) (binary+ x (+ & r))]))
```

Here we define (+) to be zero, and let the zero-argument clause be the base case. The second clause takes care of additional arguments by setting up a simple recursion; since each call to "+" decreases the number of arguments by one, the base case must eventually be reached.

It is not necessary that the base case be a zero-argument clause. For "+", we can define a tail-recursive version that uses a two-argument clause as the base case, while still supporting zero and one argument calls:

```
(define +
  (lambda*
    [() 0]
    [(x) x]
    [(x y) (binary+ x y)]
    [(x y & r) (+ (binary+ x y) & r)]))
```

The usefulness of rest values depends on an efficient implementation. If the rest values are moved on each recursive call, an algorithm that appears to be linear with respect to the number of arguments is actually quadratic. The following Scheme definition for "+" illustrates this problem:

```
(define +
   (lambda l
      (if (null? l)
          0
          (binary+ (car l) (apply + (cdr l))))))
```

Again the list destructuring interferes with clarity, but there is also a serious performance problem caused by the use of `apply` to recursively sum the list. The call-by-value semantics of Scheme demands that a fresh list be provided on each procedure call with a rest-list interface. This ensures that side effects to an existing list do not affect the arguments to a procedure. Although a compiler may be able to prove in simple cases that it is safe to use the same list across calls, an implementation cannot guarantee such behavior in general, and different implementations of the same language will vary widely in their handling of such optimizations. Consequently, an algorithm that appears to be linear with respect to the number of arguments is likely to be quadratic in some or all implementations. However, since algorithms like the one above are more elegant than those that contain inner recursions to explicitly reduce a list without the aid of `apply`, and since the copying problem is not likely to occur to most programmers, the end result is likely to be elegant programs with inexplicably poor performance.

The final example of this section uses λ^* to create a guarded memory cell:

```
(define make-cell
   (lambda (initial-value guard)
      (let ([contents (guard initial-value)])
         (lambda*
            [() contents]
            [(new-value) (set! contents (guard new-value))]))))
```

The definition of `make-cell` relies on the first-class status of procedures in Scheme. An invocation of `make-cell` returns a procedure with a private variable, `contents`. When this procedure is invoked with no arguments it returns the value of the private variable; when it is invoked with one argument it assigns the variable to a new value. The `guard` procedure may be used to coerce the new value to a value appropriate for the memory cell, or it may be used to signal an error if the new value is not appropriate. The curious aspect of this use of λ^* is that the `new-value` argument does not have a default value. Instead, two different but related actions are performed depending upon the number of arguments.

4 Implementation

The difficulty of and methods for implementing λ^* depend upon the characteristics of the language as a whole, especially its rules for the scope and extent of variable bindings. Certain features of Scheme make an efficient implementation difficult to achieve in general. The indefinite extent of variable bindings and the lack of strong typing for procedures both negatively impact efficiency. However, the λ^* construct can be made at least as efficient as the standard Scheme and Common Lisp interfaces. Furthermore, an implementation can detect many circumstances where more efficient strategies are possible. We can at least guarantee that (1) storage for rest values is allocated on the stack whenever the compiler can detect that they have dynamic extent, (2) the expense of choosing among λ^* clauses is borne only by procedures with multiple clauses, and (3) the expense of handling rest values is borne only by procedures expecting to receive rest values. The expense of choosing among λ^* clauses and of handling rest values is rarely significant, and whenever the compiler knows the interface of the called procedure at the point of call, as with calls to locally-defined procedures, these expenses are often avoided entirely. For traditional languages such as Pascal, where all variables have dynamic extent and procedure interfaces are always known, the more efficient strategies are always applicable.

There are two constraints that all implementations must obey. The first is that rest values must be protected against side effects, since a rest variable may occur in more than one procedure call. For instance, we might have both `(g & r)` and `(f & r)` for some rest variable `r`. Suppose `f` is `(lambda* [(x & s) e])`, and `e` assigns `x`. This side effect cannot be allowed to affect the values `g` receives. This problem is easily solved by copying the values for non-rest variables into their own storage locations; this copying can be avoided for variables that are never modified.

The second constraint is that procedure calls must not copy rest values unless there is no possibility that the procedure call is either directly or indirectly recursive. This constraint is more subtle since it is related to the complexity rather than to the correctness of the computation. The discussion of "+" in Section 3 illustrates how unrestricted copying of rest values can turn an apparently linear algorithm into a quadratic one.

A simple implementation can be derived directly from the semantics presented in Figure 2 by treating value sequences as heap-allocated linked lists. If the portion of the value list referenced by a rest variable is not copied when passed on procedure call, and all assignable formal parameters are given fresh locations when a procedure is invoked, both of the above constraints are satisfied. Although such an implementation would be suitable for an interpreter, many modern implementations of Lisp-like languages use a stack for passing parameters to save the expense of heap storage allocation and reclamation. Even in the presence of rest variables, a stack can still be used in many easily recognizable cases. Allocating the value list on the stack when the parameters do not have indefinite extent prevents the heap management overhead. Procedure calls can be further expedited by arranging parameters properly at the point of call, so that non-rest values are put directly in place on the stack. When it is known that the sequence of values associated with a rest variable has dynamic extent and is monotonically non-increasing in length across procedure calls, the sequence can be treated as a stack-allocated vector, resulting in further savings of storage space and access time.

The major challenge to an efficient implementation is handling calls to unknown procedures. The simplest solution is to force procedures that may be unknown to some caller to always accept heap-allocated linked parameter lists. A better solution is to provide two entry points for such procedures, one for procedure calls that do not pass along a rest list and one for calls that do. Procedures that expect their parameters on the stack are provided with an extra entry point that calls a library routine to unfold a parameter list onto the stack. Procedures that expect their parameters in a linked list are provided with an extra entry point that calls a library routine to move parameters from the stack into a linked list. This copying will be done at most once for a given parameter list if procedures that pass rest values to unknown procedures maintain the values as a linked list.

5 Multiple Return Values

The section on multiple return values in *Common LISP: The Language* [6] begins, "Normally, multiple values are not used." Although this may be because multiple values are not often useful, a further reason might be that none of the ordinary Lisp constructs are easily adapted for receiving multiple values. Fortunately, the λ^* interface adapts easily to multiple return values. Two capabilities are necessary: the capability to receive multiple values and the capability to return multiple values. For the receipt of multiple values we extend the syntax of procedure calls to allow an arbitrary expression evaluating to zero or more values to follow the ampersand. In Common Lisp, a special primitive `values` is used to signal the return of multiple values. Although such a primitive could be adopted here, we think it more natural to extend λ^* expressions to allow them to directly return multiple values. We do this by allowing the body of a clause in a λ^* expression to consist of zero or more expressions; the values of all the expressions are returned as the result of an invocation of the procedure. In addition, a clause body is permitted to contain an ampersand in the penultimate

$$
\begin{aligned}
\lambda^*\textit{-expression} &\rightarrow (\textbf{lambda*}\ \textit{clause clause}\ \ldots) \\
\textit{clause} &\rightarrow [\textit{formals body}] \\
\textit{formals} &\rightarrow (\textit{variable}\ \ldots) \mid (\textit{variable}\ \ldots\ \textbf{\&}\ \textit{variable}) \\
\textit{procedure-call} &\rightarrow (\textit{expression body}) \\
\textit{body} &\rightarrow \textit{expression}\ \ldots\ \mid \textit{expression}\ \ldots\ \textbf{\&}\ \textit{expression} \\
\textit{expression} &\rightarrow \lambda^*\textit{-expression} \mid \textit{procedure-call} \mid \textit{variable}
\end{aligned}
$$

Figure 3: Syntax for λ^* with multiple return values.

position; the expression following the ampersand may evaluate to zero or more values. The modified syntactic rules are shown in Figure 3. A formal semantics for the modified language appears in Figure 4. The most significant change is the evaluation of expressions to sequences of values rather than to single values.

It is straightforward to simulate Common Lisp's `values` form, which is just the multiple value identity function:

```
(define values (lambda* [(& r) & r]))
```

We can also define a simple procedure to force the return of the first value from an expression that returns one or more values:

```
(define first (lambda* [(x & r) x]))
```

So we can write, for instance, `(p (first & e))`, where `p` is an expression that evaluates to a procedure of one argument and `e` is an expression evaluating to one or more values.

The effects of multiple return values are far reaching, and important semantic issues must be resolved when they are added to a language. One such issue is what to do about expressions that do not evaluate to a single value in single-value contexts—in the above syntax, all expression contexts that do not immediately follow an ampersand. The approach adopted by Common Lisp is to ignore extra return values, as with `first` above, and to supply a default value when no value is returned. However, this approach hides rather than exposes errors. Since it is easy to force the return of a single value when necessary, we prefer to make unexpected or missing values an error. There are other contexts where single values are ordinarily required. In Scheme, conditional and assignment statements are the other major contexts that must be considered. For conditionals, it is reasonable to insist that the test expression evaluate to a single value, and let the context of the conditional determine the context of the branches. Little power is lost by restricting assignments to single-valued expressions. However, since variables can refer to multiple values, it is reasonable to allow multi-valued

Syntactic variables:

$$e \in expressions$$
$$b \in bodies$$
$$x \in variables$$
$$f \in formals$$

Domains:

$$v \in values = values^* \to values^*$$
$$\epsilon \in values^*$$
$$\rho \in environments = variables \to values^*$$

Semantic functions:

$$\begin{aligned}
\mathcal{E} &: expressions \to environments \to values^* \\
\mathcal{M} &: formals \to values^* \to booleans \\
\mathcal{R} &: environments \to formals \to values^* \to environments \\
\mathcal{X} &: bodies \to environments \to values^*
\end{aligned}$$

Semantic equations:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho &= \rho x \\
\mathcal{E}[\![(e\,b)]\!]\rho &= strict(single(\mathcal{E}[\![e]\!]\rho))\mathcal{X}[\![b]\!]\rho \\
\mathcal{E}[\![(\textbf{lambda*}\,[f_1\,b_1]\,\ldots\,[f_n\,b_n])]\!]\rho &= \lambda\epsilon\,.\,\mathcal{M}[\![f_1]\!]\epsilon \to \mathcal{X}[\![b_1]\!](\mathcal{R}\rho[\![f_1]\!]\epsilon)\,, \\
&\qquad\vdots \\
&\quad\ \mathcal{M}[\![f_n]\!]\epsilon \to \mathcal{X}[\![b_n]\!](\mathcal{R}\rho[\![f_n]\!]\epsilon)\,, \\
&\quad\ \text{error} \\
\mathcal{X}[\![e_1\,\ldots\,e_n]\!]\rho &= \langle single(\mathcal{E}[\![e_1]\!]\rho), \ldots, single(\mathcal{E}[\![e_n]\!]\rho)\rangle \\
\mathcal{X}[\![e_1\,\ldots\,e_n\,\&\,e_{n+1}]\!]\rho &= (\langle single(\mathcal{E}[\![e_1]\!]\rho), \ldots, single(\mathcal{E}[\![e_n]\!]\rho)\rangle)\,\S\,(\mathcal{E}[\![e_{n+1}]\!]\rho) \\
\mathcal{M}[\![(x_1\,\ldots\,x_m)]\!]\langle v_1, \ldots, v_n\rangle &= (m = n) \\
\mathcal{M}[\![(x_1\,\ldots\,x_m\,\&\,x_{m+1})]\!]\langle v_1, \ldots, v_n\rangle &= (m \leq n) \\
\mathcal{R}\rho[\![(x_1\,\ldots\,x_n)]\!]\langle v_1, \ldots, v_n\rangle &= \rho[\langle v_1\rangle/x_1]\,\ldots\,[\langle v_n\rangle/x_n] \\
\mathcal{R}\rho[\![(x_1\,\ldots\,x_n\,\&\,x_{n+1})]\!]\langle v_1, \ldots, v_n, v_{n+1}, \ldots\rangle &= \rho[\langle v_1\rangle/x_1]\,\ldots\,[\langle v_n\rangle/x_n][\langle v_{n+1}, \ldots\rangle/x_{n+1}] \\
single &= \lambda\epsilon\,.\,(\#\epsilon = 1) \to \epsilon\downarrow 1\,, \text{error}
\end{aligned}$$

Figure 4: Semantics for λ^* with multiple return values

assignments.

Although adding multiple return values to a language does complicate the implementation of the procedural interface, it is possible to place the burden of such complications on procedure calls expecting multiple values and on procedures returning multiple values without adversely affecting simpler procedure calls. Avoiding repeated movement of the multiple values is necessary to avoid introducing complexity problems similar to those mentioned in Section 3. As a result, heap storage is necessary in some cases even in a language in which variable bindings do not have indefinite extent, since the sequence of values itself may have indefinite extent.

6 Conclusions

Scheme and Common Lisp provide procedural interfaces that make it possible to define variable-arity procedures. They are similar in that procedures accepting indefinitely many arguments receive these arguments in a list. Both languages provide an `apply` function, which applies a function to the contents of a list, but using `apply` with lists can not be considered equivalent to using a rest variable in a procedure call. The problem is that list structures can be modified, so the only way to ensure that a procedure's arguments are safe is to provide it with a fresh list on each call or to prove that the existing list is never modified. Consequently Scheme and Common Lisp cannot guarantee that the use of `apply` and lists of arguments will not result in complexity problems. This situation is analogous to Scheme's requirement that tail recursion be performed with no net growth of the control stack, so that iteration may be expressed as tail recursion. In both cases, it is not sufficient that a compiler *can* provide the optimization; in order for the feature to be generally useful, the compiler *must* provide the optimization. Furthermore, just as proper treatment of tail-recursion makes it unnecessary to include other primitive iterative control structures in a language, proper treatment of the rest variable interface eliminates the need to include primitive data structures and procedures in the language simply to support the procedural interface.

Common Lisp provides additional mechanisms to make defining procedures that take optional arguments with default values more convenient. However, in cases for which there are no default values or for which an optional parameter appears before a required parameter, the Common Lisp optional interface becomes clumsy. We find λ^* expressions both simpler and more elegant in many cases.

Bellot and Jay provide a combinator-based semantics for the equivalent of the lambda calculus extended to allow variable-arity functions [1]. They support "rest" arguments by extending the syntax and semantics of the lambda calculus to handle variable-arity functions directly. They also avoid commitment to a data structure for passing extra arguments. They do not suggest the use of multiple clauses as a way to provide optional arguments and access to rest values, and they do not address the potential complexity problem caused by copying rest values.

It is tempting to view λ^* as providing a limited sort of pattern matching and argument destructuring, such as provided by ML [4]. However, λ^* is intended to provide a representation-independent way to manipulate indefinite numbers of arguments without regard to the structure of the arguments themselves. In contrast, ML supports the definition of procedures with fixed numbers of arguments (via currying), allowing pattern matching based on the structure of the arguments to be used in choosing among alternative procedure bodies. Our mechanism is orthogonal to the notion of pattern matching, though λ^* might provide a natural basis for an ML-like `pattern-lambda` that supports both variable-arity procedures and pattern-matching.

The apparent utility of rest variables would be increased by allowing ampersands and rest variables to appear anywhere in a formal parameter specification or procedure call. For example, we might wish to extend `cons` to accept an indefinite number of arguments (Common Lisp `list*`) with the following tail-recursive definition:

```
(define cons
  (lambda*
    [(x) x]
    [(& r x y) (cons & r (binary-cons x y))]))
```

In this definition, `r` refers to all the arguments except for the last two, and it appears before rather than after the other argument in the recursive call to `cons`. Although this extended interface does make solutions to some programming problems simpler, efficiency problems inherent in the interface are not easily resolved. In particular, a straightforward generalization of the implementation techniques discussed in Section 4 would not allow us to make the same guarantees about the time and space complexity of procedures using the extended interface as we can make about procedures using the more restricted interface.

We have chosen to emphasize variable-arity procedures over multiple return values. One reason is that, although the capability to return multiple values using λ^* depends upon variable-arity procedures, variable-arity procedures are independent of multiple return values. Consequently, it is reasonable to support variable-arity procedures without supporting multiple return values. Furthermore, a programmer who does not wish to use aggregate structures to return multiple values can use continuation-passing-style programming techniques to "return" multiple values in a language that allows procedures to be treated as first class objects. Since multiple return values have such far reaching effects on the syntax, semantics, and implementation of a language, and since there are other means of achieving similar results, it is not clear that the benefits of multiple return values outweigh the costs.

Acknowledgements: The authors would like to thank Matthias Felleisen, Dan Friedman, Chris Haynes, Stan Jefferson, Bruce Duba, David Wise, and the anonymous reviewers for their helpful comments on earlier versions of this article. The examples were formatted by Carl Bruggeman's Scheme TEXer.

References

[1] Bellot, P. and Jay, V. A theory for natural modelisation and implementation of functions with variable arity. In *Proceedings of the 1987 Conference on Functional Programming and Computer Architecture*, Kahn, G., editor, *Lecture Notes in Computer Science No. 274* (September 1987) 212–233.

[2] Hewitt, C. E. and Smith, B. Towards a programming apprentice. *IEEE Transactions on Software Engineering*, **SE-1**, 1 (March 1975) 26–45.

[3] Landin, P. J. The next 700 programming languages. *Communications of the ACM*, **9**, 3 (March 1966) 157–166.

[4] Milner, R. A proposal for Standard ML. *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* (August 1984) 184–197.

[5] Rees, J. A. and Clinger, W. (Ed.), The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, **21**, 12 (December 1986) 37–79.

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