A SIMPLE AND EFFICIENT IMPLEMENTATION OF HIGHER-ORDER FUNCTIONS IN LISP

December 1984

Technical Note 339

By: Michael P. Georgeff, Program Director
Artificial Intelligence Center
Computer Science and Technology Division

Stephen F. Bodnar
Department of Computer Science
La Trobe University
Bundoora, Victoria, 3083, Australia

This work was largely done while the authors were members of the Computer Science Department, Monash University, Clayton, Victoria, Australia.

SRI International
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Contents

Abstract

A relatively simple method for handling higher-order functions (*funargs*) in LISP is described. It is also shown how this scheme allows extension of the LISP language to include partial application of functions.

The basis of the approach is to defer evaluation of function-valued expressions until sufficient arguments have been accumulated to reduce the expression to a nonfunctional value. This results in stacklike environment structures rather than the treelike structures produced by standard evaluation schemes. Consequently, the evaluator can be implemented on a standard runtime stack without requiring the complex storage management schemes usually employed for handling higher-order functions.

A full version of LISP has been implemented by modifying the FRANZ LISP interpreter to incorporate the new scheme. These modifications prove to be both simple and efficient.

§1 Introduction

Higher-order functions[1] play a very important role in functional programming languages such as LISP. However, their implementation has always been expensive in both storage and time (e.g., [Bobrow and Wegbreit 1973], [Greenblatt 1974], [Baker 1978]), and many of the more common versions of LISP (e.g., FRANZ LISP [Foderaro 1980]) do not fully support them.

As is well known, in a statically scoped language (e.g., ALGOL) the value of a function depends not only on its code, but also on the environment at the time the function is defined. Thus, a functional value is usually represented by a construct, called a *funarg* or *closure*, which specifies both the function code and the environment of definition. However, such funargs can give rise to environment structures which are treelike rather than stacklike, and this considerably complicates storage management.

Higher-order functions prove troublesome even in LISP, despite the fact that LISP conventionally uses dynamic binding. The problem is that, while dynamic scoping serves satisfactorily in most cases, the usual intent of the programmer in writing a function-valued expression is to have static scoping.

The most straightforward way to implement such treelike environment structures is by means of a heap. However, this requires that environment extensions be copied from the runtime stack to the heap and back again, and the allocation and reclamation of such blocks is time consuming.

An alternative to the use of a heap is a spaghetti stack [Bobrow and Wegbreit 1973], where activation frames are retained on the stack until the environment component they contain is no longer needed. Although this approach is preferable to using a heap, it is still much less efficient than a standard runtime stack and can consume a large amount of stack space. More importantly, the technique is poor when shallow binding is used to represent the environment structure (e.g., [Baker 1978]) as the changes to the value cells on function entry and exit are computationally expensive.

Another approach is to allocate binding environments on the stack, moving them to the heap only when necessary [McDermott 1980]. In some cases, this avoids using the heap, but the scheme requires checks on each function call to determine whether or not a transfer to the heap is required. This is expensive, even in the case when no funargs are involved.

In this report we describe an alternative method for implementing higher-order func-

[1]That is, functions that have functional arguments or return functional values.

tions in LISP and similar languages. The implementation is based on the function-deferring scheme proposed by Georgeff [Georgeff 1982, 1984]. We first describe the standard approach to the implementation of funargs. We then describe the strategy of function deferral and construct a LISP interpreter based on this scheme. We also show how this scheme makes it possible to extend the LISP language to provide for partial application of functions. Finally, we describe how to modify the FRANZ LISP (Opus 36b) interpreter so that it will properly handle higher-order functions and partial application.

§2 Evaluation of LISP Expressions

2.1 The Standard Approach

We will be primarily concerned with the applicative part of LISP. Expressions in applicative LISP are one of three kinds: ***constants*** and ***variables***, denoting such things as primitive functions and integers; ***lambda abstractions***, consisting of a list of binding variables and a body, and which are used to define functions; and ***applications***, consisting of an operator and zero or more operands, and which represent the application of a function to its arguments.

The value of an expression depends on the values of the variables occurring *free* in the expression. The structure that provides the values of the free variables is called an *environment*.

Let e be an expression to be evaluated under static scoping in an environment E. Then there are three cases to consider:

- If e is a constant, its value is known immediately and, if e is a variable, its value is obtained directly from the environment E.
- If e is an abstraction, evaluation of the body of the abstraction is temporarily deferred. The value of the abstraction is thus represented by a pair consisting of the abstraction itself (i.e., the code defining the function), and the environment E. Such a pair is called a *funarg* or *closure* [Landin 1964].[2]
- If e is an application, its value is obtained by first evaluating the operands of the application, then applying the value of the operator to these values. If the operator evaluates to a

[2]In fact, we do not need to form a funarg if the abstraction is in operator position. In such cases, we can optimize evaluation and directly apply the abstraction to the values of the operands (see [Burge 1975]).

primitive function, the result of the application is predefined (by the system). Otherwise the operator must evaluate to a funarg. The binding variables of the funarg are then bound to the values of the operands, the environment of the funarg extended to include these new bindings, and the body of the funarg evaluated in this new environment.

However, LISP conventionally uses dynamic scoping. In this case, the body of an abstraction must be evaluated in the application environment (i.e., the environment at the time of application of the abstraction), rather than in the environment of definition (i.e., the environment at the time of creation of the abstraction). It is then sufficient to represent the value of an abstraction by the code itself (rather than a funarg), as there is no need to save the environment of definition.

When static scoping of an expression *is* desired, a special construct, called **function**, must be used. The **function** construct forces the creation of a funarg as described above.

A LISP computation will usually generate a stacklike environment structure which is extended on entering the body of an abstraction and restored on return. However, because funargs "capture" environments, evaluation of statically scoped function-valued operands can create additional branches in this structure. This results in an environment structure that is treelike rather than stacklike.

Statically scoped function-valued operands do not always destroy the stack structure. If a functional expression is a constant, variable, or lambda abstraction, the environment structure remains stacklike. This is because the environment that is "captured" by the funarg is guaranteed not to be an extension of the current environment — indeed, it is exactly the current environment. However, when a funarg is returned as the result of an application, the "captured" environment may well be an extension of the current environment, thus creating a treeelike structure. Such funargs will be called *upward funargs* to distinguish them from those functional arguments (*simple funargs*) that do not destroy the stack discipline.

To illustrate the standard approach to evaluation, let us consider the following example. Assume we have the function definitions

```
(defun appl (f x)
        (f x))
(defun fnplus (x)
        (function (lambda (y) (+ y x)))
```

In this case, we desired static scoping for the functional value of **fnplus**, and therefore used the **function** construct to force creation of a funarg.

Now consider applying the function **appl** to the function-valued expression **(fnplus 10)** and the constant **20**, i.e.,

(appl (fnplus 10) 20)

The standard strategy for evaluating this expression is as follows:

1. The operands of **appl** are evaluated in the current (global) environment.

2. The first operand is the expression **(fnplus 10)**. The operand of this expression (i.e., **10**) is evaluated, yielding the value 10.

3. The function **fnplus** is applied to this argument. This results in the binding variable (**x**) of the function being bound to 10. The environment is extended to include this new variable-value pair and the body of **fnplus** evaluated in this new environment.

4. As the body of **fnplus** is specified to be function-valued (by means of the **function** construct), evaluation yields the funarg <FUNARG **(lambda (y) (+ y x))** *env*>, where *env* is the current environment (called the *binding environment* of the funarg). In this environment, the variable **x** is bound to the value 10.

5. This funarg is returned as the value of the first operand of **appl**. Note that at this stage the environment of the funarg includes a binding (i.e., **x** to 10) that is not present in the current environment and that, under a stack organization, would have been popped and thus rendered inaccessible.

6. The second operand of **appl** (i.e., **20**) is now evaluated, yielding the value 20.

7. The binding variables of **appl** are now bound to the current arguments: **f** is bound to the funarg <FUNARG **(lambda (y) (+ y x))** *env*> and **x** to 20.

8. The environment is extended to include these new variable-value pairs and the body of the function **appl** (i.e., **(f x)**) is entered.

9. The operand, **x**, of this expression is evaluated in the current environment (called the *application environment*), yielding 20.

10. The operator, **f**, is evaluated, yielding the funarg <FUNARG **(lambda (y) (+ y x))** *env*>.

11. This funarg is now applied to the argument 20. First, the environment is restored to the binding environment of the funarg (i.e., *env*).

12. As the body of the funarg is an abstraction, the binding variable of the abstraction (**y**) is bound to 20 and the environment *env* extended. The body **(+ y x)** of the abstraction is evaluated in this new environment, eventually yielding the result 30.

13. The application environment (i.e., the environment that existed at the time of application of **appl**) is now restored.

14. The application of **appl** finally returns the result 30, and the environment is restored to the top level.

2.2 Deep Binding and Shallow Binding

To find the value of a variable at a given stage of the computation, the environment structure needs to be searched from the current environment block to the root for the first occurrence of the variable.

There are two standard means of implementing this environment structure. The first technique involves pushing each new variable-value pair onto an environment stack (often called the *bindstack*) as new bindings are created, and popping the stack as we return to previous environments. The search for a variable then simply involves looking up the stack through each environment block until the first occurrence of the variable is found. This scheme is known as *deep binding*. Unfortunately, in most cases the search will examine many environment blocks before the desired variable is found, which can be very inefficient.

Functional arguments and values do not present too many difficulties under deep binding. When a funarg is applied, the environment must be restored to the binding environment of the funarg. This process is known as *context changing* or *context swapping*, and is relatively efficient under deep binding. If upward funargs occur, environment structures are treelike and storage management is more complicated. However, the context swapping mechanism is unaffected.

An alternative scheme is known as *shallow binding*. Under this scheme, each variable is associated with a *value cell* that contains the current value of the variable. Thus, finding the value of a given variable is straightforward, but the binding and unbinding mechanisms are more complex. When binding a variable we have to replace the current value cell with the new binding, saving the old binding (on a bindstack) so that it may be restored when we return. The unbinding operation restores the value cell to its previous state. The bindstack is therefore much like the bindstack under deep binding, except that the most recent value of each variable is kept in a value cell rather than on the bindstack.

However, functional arguments and values complicate the shallow binding process considerably. We cannot simply switch to a different environment as we can under deep binding, but must re-establish the value cells that were current when the environment was created. This involves swapping the contents of the value cells with the values retained on the bindstack, up to the point at which the new environment was created. Thus, whereas deep binding is time-unbounded in accessing variables and time-bounded in context changes, shallow binding is time-bounded for variable access and time-unbounded for context changes. The reason shallow binding is usually preferred is that context changes occur far less frequently than variable accesses.

Let us now consider a stack implementation of shallow binding in more detail and, in particular, examine how functional arguments are handled. As we have previously mentioned, evaluation of a functional argument yields a closure that contains a reference to the current environment. In the case of shallow binding, this is simply implemented as a pointer to the current top of the bindstack. It is usually known as the ***binding context pointer***.

When a funarg is applied, we need to restore the value cells to their state at the time of the funarg's creation. In the case of a simple funarg, this involves *unwinding* the binding stack back to the binding environment, swapping value cell contents on the way. By the time we reach the binding environment (indicated by the binding context pointer), the value cells have been restored to their values at the time of creation of the funarg, and the segment of the stack between the original application environment and the binding environment retains the bindings that were made between creation of the funarg and its application. This stack segment must be saved to allow restoration of the original application environment. The body of the funarg is then entered and, after evaluation, the environment at the time of application of the funarg is restored by *winding* the binding stack back to the original application environment. A more detailed description of this process can be found in [Allen 1978].

Upward funargs are much more difficult to implement. Not only do we have to construct a scheme for representing a treelike environment structure rather than a stacklike one, but the mechanism for rebinding from the current application environment to the binding environment of the funarg is much more eomplex. The critical problem is that of discovering the path between the current environment and the binding environment. One way of doing this is first to search from the binding environment towards the root of the environment tree until the currently active branch is intersected. Next, the binding stack is unwound from the current active environment to this intersection, and then from there back to the binding environment of the funarg. On completion of the application, we then wind back to the original application environment. More details can be found in [Allen 1978] and [Baker 1978].

2.3 The Function-Deferring Strategy

Most previous approaches to the implementation of higher-order functions in LISP have concentrated on making the management of the treelike environment structures as efficient as possible (e.g., [Bobrow and Wegbreit 1973, Baker 1978]). However, the approach described in this report avoids the creation of such structures. Consequently, the relatively simple stack management schemes used for handling simple funargs can be used without modification.

The basis of the scheme is to change the usual execution strategy for function application and evaluation, so that the evaluation of every function-valued expression is deferred until sufficient arguments have been accumulated to allow reduction to a basic (i.e., nonfunctional) value.

This is accomplished by temporarily representing the value of such an expression by the expression itself together with the current environment, just as in standard evaluators the value of a lambda abstraction is represented by the abstraction itself together with the current environment. And, just as with lambda abstractions, this code is evaluated (in the saved environment) only at the time the functional value is actually applied.

Let us describe this process in more detail. Consider a function-valued applicative expression, $e = (e_0\ e_1 \ldots e_n)$, to be evaluated in an environment E. We could represent this directly as a new type of funarg consisting of the expression e and the environment E. However, for reasons of efficiency and to better match the conventional order-of-evaluation rule, it is preferable to evaluate the operands e_1 to e_n of the expression prior to forming the funarg. Thus, this new type of funarg consists of the (unevaluated) operator e_0, a list of the values of the operands e_1 to e_n, and the current environment E. The operator part is called the *body* of the funarg and the list of operand values is called the *arglist* of the funarg. We can consider the old type of funarg that represented a lambda abstraction to be a special case of this new funarg, i.e., the case in which the arglist is empty.

Such a funarg is applied in much the same way as the old type of funarg: the environment is restored to the binding environment of the funarg and body of the funarg evaluated in this environment. However, it differs in that the arguments in the arglist of the funarg must first be appended to the list of current arguments. This is simply because application of the body of the funarg to these arguments was previously deferred, and now is the time to do that application.

This scheme ensures that the environment structure will always be stacklike, as the evaluation of function-valued operands cannot extend the current environment. In essence, the values of function-valued operands are forced to be simple funargs. However, we are not quite out of the woods: what happens when we actually come to apply a functional value to its arguments?

Under the function-deferring scheme, evaluation of applicative expressions is always deferred unless the expression is basic-valued. Therefore, to answer the above question, we need only consider basic-valued expressions. Furthermore, if the operator of such an expression evaluates to a basic-valued function, it can simply be applied to the values of the operands in the normal way. However, we do have difficulties if the operator denotes a function-valued function.

Let $e = (e_0\, e_1 \ldots e_n)$ be a basic-valued expression and let e_0 denote a function-valued function. As e_0 is function-valued, and the expression is basic-valued, the function must only apply to $m < n$ of the operands (i.e., the function must have arity $m < n$). The problem now is that in applying e_0 to $e_1, e_2, \ldots e_m$, we may create a functional value whose environment of definition is an extension of the application environment. If a standard runtime stack were used, when this functional value is returned (to be later applied to the remaining arguments), the environment of definition would be popped and thus rendered inaccessible.

But let us look carefully at the sequence of events in the application. Let E_a denote the application environment, and assume that, under the standard LISP call-by-value scheme, the operator e_0 and all the operands $e_1, e_2, \ldots e_n$ have been evaluated. First, e_0 is applied to $e_1, e_2, \ldots e_m$, creating possibly an environment E_f which is an extension of E_a. If a functional value F is created in this new environment (E_f), any representation of F will need to save this environment of definition. When F is returned as a value, the environment E_a will be restored and F applied (in the environment E_a) to the remaining (evaluated) operands $e_{m+1} \ldots e_n$. Upon application, the defining environment of F (i.e., E_f) will need to be restored, and the body of F applied to its arguments in this new environment. Thus the return to the environment E_a was wholly unnecessary — it would have been preferable to remain in the environment E_f, passing the remaining arguments *down* to F, and only returning when the application was complete (i.e., all the arguments consumed).

There are two consequences of doing things this alternative way. First, it is much more efficient as unnecessary environment swapping is avoided. Even with the standard evaluation schemes, this is a useful optimization. Neither need we explicitly create a functional value to represent F, as we can apply F to its arguments immediately.

Second, and more importantly in our case, is that the environment in which evaluation is taking place is never restored to a state which does not include the binding environment of any funargs. Consequently, a standard runtime stack can be used for managing evaluation and storing environment structures.

Just as in standard LISP, we need to use the **function** construct to indicate static scoping. However, it is now used differently: the **function** construct is used simply *whenever one has a function-valued operand* for which static scoping is desired. That is, the **function** construct is applied to function-valued expressions that occur in *operand* position. In contrast, in standard versions of LISP, the **function** construct is used at the point a function is to be *returned* as a value from within a function-valued function.

To reinforce these ideas, let us consider how the function-deferring scheme operates on the example given in the previous section. We will assume the functions to be as before, except, as mentioned above, in the way that the **function** construct is used:

```
(defun appl (f x)
       (f x))


(defun fnplus (x)
       (lambda (y) (+ y x)))
```

The expression to be evaluated is

```
(appl (function (fnplus 10)) 20)
```

Here the **function** construct is used to signify a *functional argument* or *function-valued expression*. As before, it forces the creation of a funarg.

When the above expression is evaluated, the following steps will occur:

1. The operands of **appl** are evaluated in the current environment.
2. The first operand is the expression **(function (fnplus 10))**. The operand of this function-valued expression (i.e., **10**) is first evaluated, yielding the value 10.
3. The (new type of) funarg <FUNARG **(lambda (x) (lambda (y) (+ y x)))** (10) *env*> is then formed.
4. This funarg is returned as the value of the first operand of **appl**. Note that the environment structure remains stacklike.
5. The second operand of **appl** (i.e., **20**) is now evaluated, yielding the value 20.
6. The binding variables of **appl** are bound to the current arguments: **f** is bound to the funarg <FUNARG **(lambda (x) (lambda (y) (+ y x)))** (10) *env*> and **x** to 20.
7. The environment is extended to include these new variable-value pairs and the body of

appl (i.e., (**f x**)) is entered.

8. The operand, **x**, of this expression is evaluated in the current [application] environment, yielding 20.

9. The operator, **f**, is evaluated, yielding the funarg <FUNARG (**lambda** (**x**) (**lambda** (**y**) (**+ y x**))) (10) *env*>.

10. This funarg is now to be applied to the current list of arguments, at this stage containing the value 20. The arglist of this funarg, containing the argument 10, is first appended to the argument list. The binding environment of the funarg (i.e., the environment at the time of the funarg's creation) is now restored, and the body of the funarg (i.e., the lambda abstraction (**lambda** (**x**) (**lambda** (**y**) (**+ y x**)))) is entered.

11. The binding variable **x** is bound to the first argument in the current argument list (i.e., 10), and the environment extended to include this new variable-value pair. The body of the lambda abstraction is now evaluated.

12. The body of this abstraction is itself an abstraction (i.e., (**lambda** (**y**) (**+ y x**)). However, unlike standard approaches, we do *not* create a funarg of this abstraction and return it (to the old environment) as the result of the application to the first argument. Instead, the binding variable **y** is directly bound to the remaining argument in the argument list (20). The current environment is then extended by this variable-value pair and the body of the abstraction (i.e., (**+ y x**)) evaluated in this new environment. The evaluation eventually yields the result 30.

13. The environment at the time of the application of **appl** is restored.

14. The application of **appl** finally returns the result 30, and the environment is restored to the top level.

As can be seen by comparing this with the standard evaluation strategy (Section 2.1), in essence the only change is the order in which subexpressions are evaluated. Thus, under the standard order of evaluation, the abstraction ((**lambda** (**x**) (**lambda** (**y**) (**+ y x**))) is applied to the argument 10 at Step (3), whereas under the function-deferring strategy the application is done later, at Step (11). Of course, in the presence of side effects, the different order of evaluation could result in an expression yielding different values.

2.4 Partial Application

One simple alternative to defining function-valued functions by abstracting a functional value is to use partial application (e.g., [Burstall et al. 1971]). Partial application allows multi-adic functions whose range is a basic (i.e., nonfunctional) value to be partially applied to a subset of their arguments, thus in effect yielding a functional value. As it turns out, most of the useful function-valued functions can be simply represented as partial applications of standard multi-adic functions. For example, consider that we want to add an integer a to each element of a list l of integers. One way to do this is to map the function that adds a to its argument over the list l. Thus, in LISP we would have

i. (mapcar (function (lambda (x) (plus a x))) l)

But, with partial application, we could write this more simply as

ii. (mapcar (function (plus a)) l)

Not only is this somewhat easier to read, but it also allows for more efficient evaluation. In expression (i), the lambda form prevents evaluation of the variable **a** until application of the abstraction. Thus **a** will be re-evaluated for every element in the list l. With partial application, however, the variable **a** (and, more generally, any expression standing as an argument) need only be evaluated once. Of course, because of the difference in evaluation times, the two above expressions may yield different values in the presence of side effects.

The technique of deferred evaluation of function-valued arguments can be extended to support partial application without any major change in the evaluation mechanism. Partial application simply results in the creation of a funarg, with the values of the partially applied arguments forming the arglist of the funarg and the function itself forming the body. This allows us to extend the LISP language to include partial application without significantly complicating the interpreter.

§3 A Function-Deferring LISP Interpreter

3.1 The **eval** Function

We now indicate how the technique of function deferral can be accomplished by giving a modified version of the standard LISP **eval** function. (Where we refer to standard LISP, we mean LISP 1.5 [McCarthy et al. 1965], or what is often called full LISP.) The eval function we use is based on that given in [Allen 1978]. The definition is recursive, thus simplifying the handling of control and letting us concentrate on the essential aspects of function deferral. However, the environment is represented as a global data structure.

The standard definition of **eval** is as follows:

```
function eval (exp)
  if (is-const exp) then (denote exp)
  elseif (is-var exp) then (lookup exp)
  elseif (is-cond exp) then (evcond (argc exp))
  elseif (is-funval exp) then (mkfunarg exp)
  elseif (is-application exp) then
          (apply (eval (func exp)) (evlis (args exp)))
  else (error)
end.
```

The above language is a variant of LISP close to Common LISP [Steele 1984]. Indentation (rather than brackets) is used to specify program structure at the top level.

The recognizers **is-const** and **is-var** test for constants and variables, respectively.[3] The function **denote(exp)** returns the internal represention of the constant **exp** and **lookup(exp)** returns the value of the variable **exp** in the current environment. The recognizer **is-cond (exp)** tests for a conditional expression **exp** with clause list **argc(exp)**, and **evcond** evaluates the clauses. The recognizer **is-funval(exp)** checks for an expression **exp** whose head is the function construct (**function**), and **is-application(exp)** checks for an applicative expression **exp** with operator (function part) **func(exp)** and operands (arguments) **args(exp)**. The constructor **mkfunarg** creates a funarg and **evlis (l)** maps **eval** over each of the elements of the list l (i.e., evaluates each of the arguments in l, forming a list of the results). The function **apply(f args)** applies the evaluated function **f** to the list of evaluated arguments **args**.

Note that the above **eval** function allows a wider class of LISP expressions than most LISP evaluators. In conventional evaluators, only the operands (arguments) in the applicative expression are evaluated; the operator (function part) is assumed to be either a primitive function, a lambda abstraction or a variable denoting one of these.[4]

For the function-deferring evaluator, we assume the use of the function construct

[3]Under the dynamic binding convention of LISP, lambda abstractions are considered to be constants.

[4]Exceptions are the evaluation scheme proposed by [McCarthy et al. 1965] and that used for SCHEME [Steele and Sussman 1978]. The LISP variant described by [Henderson 1980] also evaluates the functional part of applicative expressions, but assumes the resulting value will be a closure representing a lambda abstraction. MACLISP [Touretsky 1975] allows some function-valued forms but not others. Common LISP [Steele 1984] and later versions of Franz LISP (e.g., Opus 38.79) allow the functional part of an expression to be evaluated by using the **funcall** construct.

to defer evaluation of function-valued operands. This requires modifications to the funarg constructor, **mkfunarg**, and the function that applies a funarg to its arguments. We will consider those modifications shortly. However, we also need to modify the evaluator so that functional values occurring in *operator* position are not returned as values, but are instead directly applied to their arguments to produce a basic value (see Section 2.3).

To do this, when **eval** is called to evaluate the operator of an applicative expression, it must also be given the list of arguments to which the value of the operator is to be applied. Thus this list of arguments, called **arglists**, becomes an additional parameter to **eval**. When the value of the operator is eventually found, it is directly applied to the arguments in **arglists** rather than returned as a functional value to be applied at a higher level in the recursion. (As we will see later, if **arglists** is empty, the call on **apply** simply returns its argument unchanged.)

There is one additional complication. If we are to allow niladic functions, then it is necessary to remember the structure of the applications in the expression being evaluated.[5] Thus, instead of simply accruing a list of arguments, **arglists** contains a list of argument *lists*, one from each application. This also helps in debugging programs, as the application of a function to an incorrect number of arguments can then be readily detected.[6]

The function-deferring **eval** is given below. The full interpreter, implemented in LISP, is given in Appendix 1.

```
function  eval (exp arglists)
  if (is-const exp) then (apply (denote exp) arglists)
  elseif (is-var exp) then (apply (lookup exp) arglists)
  elseif (is-cond exp) then (apply (evcond (argc exp)) arglists)
  elseif (is-funval exp) then (apply (mkfunarg exp) arglists)
  elseif (is-application exp) then
            (eval (func exp) (cons (evlis (args exp)) arglists))
  else (error) end.
```

If **arglists** is empty, **eval** simply returns the value of **exp** as in the original **eval** (remembering that, in this case, the calls to **apply** do nothing). If **arglists** is non-empty, and **exp** is a constant, a variable, a conditional or a function construct, then **eval** simply applies the value of the expression **exp** (corresponding to the operator of the original expression) to the

---

[5]In the previous **eval** function, this structure was captured in the form of the recursive calls to **apply**.

[6]In the evaluator given in [Georgeff 1982, 1984], niladic functions were disallowed, thus providing a simpler structure for **arglists**.

arguments in **arglist**. In this way it is no different from the original **eval** which called **apply** directly. However, it differs from the standard definition when **exp** is itself an applicative expression. In this case, it simply recurses on itself, collecting up all the [evaluated] arguments of the application in **arglists**.

The technique of providing **eval** with a list of evaluated arguments (**arglists**), together with use of the **function** construct to defer evaluation of function-valued expressions, prevents funargs from being returned to higher levels in the recursion. Consequently, the environment structure is always stacklike and relatively simple environment management schemes can be employed. The provision of an argument list also allows for partial application, as described later.

Now we can look at the construction of funargs. In the function-deferring evaluator, a funarg **f** consists of a function part **ffun(f)**, a list of evaluated arguments **farg(f)** and an environment **fenv(f)**. The funarg constructor **mkfunarg** is defined as

```
function mkfunarg (exp)
  if (is-application exp) then
    (list 'FUNARG (func exp) (list (evlis (args exp))) env)
  (list 'FUNARG exp nil env)
end.
```

The variable **env** denotes the current environment. Note that, when the expression **exp** is an application, the arguments occurring in **exp** are all evaluated. As discussed previously, this is primarily to provide consistency with the LISP call-by-value convention, as well as to improve efficiency. However, if other evaluation strategies were desired (e.g., call-by-name, call-by-need) evaluation of the arguments could be deferred.

3.2 The **apply** Function

We will now consider the **apply** function. The **apply** function used in conventional LISP interpreters (e.g., [Allen 1978]) is essentially as follows:

```
function apply (fn args)
  if (is-car fn) then (car (arg1 args))
  elseif (is-cdr fn) then (cdr (arg1 args))
  elseif (is-cons fn) then (cons (arg1 args)(arg2 args))
 .....
  elseif (is-lambda fn) then (apply-abs (body fn) (vars fn) args)
  elseif (is-funarg fn) then
             (apply-funarg (fbody fn) (fargs fn) (fenv fn))
  else (error)
end.
```

The recognizers **is-car**, **is-cdr** and **is-cons** check for the primitive functions CAR, CDR and CONS respectively, and the "...." represents the code for the remaining primitive functions supported by the LISP system. The functions **arg1** and **arg2** take the first and second arguments, respectively, in the list of arguments **args**. The recognizer **is-lambda** checks for a lambda abstraction and applies it via the call on **apply-abs**, and **is-funarg** tests for a funarg and applies it via the call on **apply-funarg**. The selectors **body**, **vars**, **fbody**, **fargs** and **fenv** select the appropriate components of the structures to which they are applied.

The **apply** function for the function-deferring interpreter differs primarily in that it receives a list of argument *lists*, and must recurse on itself until this list is fully consumed. This is necessary because the function **eval** no longer calls **apply** when it encounters an applicative expression. The new **apply** function is as follows:

```
function apply (fn arglists)
  if (null arglists) then fn
  else
    let args = (first arglists)
         remains = (rest arglists)
      if (is-car fn) then (apply (car (arg1 args)) remains)
      elseif (is-cdr fn) then (apply (cdr (arg1 args)) remains)
      elseif (is-cons fn) then (cons (arg1 args)(arg2 args))
      .....
      elseif (is-lambda fn) then (apply-abs (body fn) (vars fn) arglists)
      elseif (is-funarg fn) then
             (apply-funarg (fbody fn)(append (fargs fn) arglists) (fenv fn))
      else (error)
end.
```

The let construct is just a convenient form of lambda binding: it binds the variable to the left of each "==" sign to the value of the expression on the right, and then evaluates each expression (form) in the body of the construct (i.e., the expression following the binding part) with these bindings.

Note that some primitives (e.g., CONS) cannot produce functional values, so that in these cases the result of the application need not be re-applied to any remaining lists of arguments. (An error could be flagged if **arglists** contained more than one argument list.)

The only remaining functions we need consider are **apply-abs** and **apply-funarg**. The function **apply-abs** applies a lambda abstraction in the usual way: the body of the lambda abstraction is evaluated in an environment formed by binding the variables (parameters) of the lambda abstraction to the arguments in **arglist**. It differs from standard, however, in that any remaining (unpaired) arguments are supplied to **eval** as the eval **arglist**. This list of remaining arguments will be nonempty only if the abstraction is function-valued, and will be used by **eval** to complete the application.

The definition of **apply-abs** is as follows:

```
function apply-abs (body vars vals)
        (pushvars vars)
        (assign-vars vars (first vals))
        let result == (eval body (rest vals))
            (popvars)
            result
```

The function **pushvars** extends the current environment with the binding variables of the abstraction, and **assign-vars** binds the variables to their values (i.e., the values in the first list of arguments). The function **popvars** restores the environment to its state prior to the call.

The function **apply-funarg** is also very similar to conventional funarg application — the environment is set to that of the funarg and the funarg evaluated therein by applying the function part of the funarg to the list of given arguments. In this case, however, the function part may itself be an expression and thus needs to be evaluated by **eval**. The call on **eval** will in turn cause the evaluated functional part to be applied to the given arguments. Here is its definition:

```
function apply-funarg (body args envbinding)
   let envappln = root
     (setenv envbinding)
     let  result = (eval body args)
          (setenv envappln)
          result
```

The function **setenv(env)** sets (unwinds) the environment to **env**. The variable **root** is the current environment.

3.3 Binding Strategies

The above evaluation mechanism makes no committment regarding the binding strategy to be used. However, unlike standard schemes for supporting functional values, the environment structure for the function-deferring interpreter is always stacklike rather than treelike. This is because, by deferring the evaluation of function-valued expressions, upward funargs (applicative expressions that evaluate to a function) are in essence transformed into simple funargs (where no application takes place). This means that the environment management schemes that work for simple funargs will work for the function-deferring interpreter, and the more complex schemes required for handling upward funargs in the standard interpreters can thus be avoided. In particular, the function-deferring scheme allows the environment structure to be built on the runtime stack rather than the heap. The gain in efficiency is especially great if a shallow-binding strategy (see Section 2.4) is used, as the swapping of value cell contents can be considerably reduced in comparison with that of conventional LISP interpreters.

Although not realized in the above interpreter, further gains in efficiency can be achieved by storing the **eval** arglists on the runtime stack. Furthermore, in the evaluation of function-valued expressions, the function-deferring scheme allows funargs to be created and retained on the stack without moving any of the evaluated operands. In contrast, schemes involving heap allocation of environment structures are required to copy argument values to the heap (and later, back again) as these arguments become part of an extended environment.

In particular cases, the gain in efficiency resulting from simpler environment structures and the avoidance of heap operations can be offset by multiple evaluation of function-valued expressions. This can happen when the expression to be evaluated contains multiple applications of a function-valued variable (i.e., one that is bound to a funarg). Because the evaluation of such expressions is deferred, multiple occurrences of a function-valued variable result in multiple evaluations of the functional expression to which it is bound. Conversely, conventional interpreters for full LISP need perform but a single evaluation of the functional expression. However, conventional interpreters must also do a context swap for each application of the resulting functional value, and this can be just as expensive as re-evaluation.[7]

3.4 Partial Application

As mentioned previously, the above interpreter can be extended to allow evaluation of expressions involving partial application.

The only effect that partial application has on the above interpreter is that the arguments to a function may not all occur in the first argument set in the arglists given to **apply**. Thus it is necessary to modify **apply** to allow functions to be applied to sufficient argument sets in **arglists** to provide all the arguments they need. The simplest way to do this is to restructure **arglists** so that all the arguments needed by the function do in fact occur in the first argument set. Thus **apply** becomes

```
function apply (fn arglists)
  if (null arglists) then fn
  else
    let newarglists == (restructure (adicity fn) arglists)
      let args == (first newarglists)
          remains == (rest newarglists)
        if (is-car fn) then (apply (car (arg1 args)) remains)
        elseif (is-cdr fn) then (apply (cdr (arg1 args)) remains)
        elseif (is-cons fn) then (cons (arg1 args)(arg2 args))
        .....
        elseif (is-lambda fn) then (apply-abs (body fn)(vars fn) newarglists)
        elseif (is-funarg fn) then
            (apply-funarg (fbody fn)(append (fargs fn) arglists) (fenv fn))
        else (error)
end.
```

[7]Some comparisons of the conventional approach and the function-deferring approach can be found in [Georgeff 1982, 1984].

Note that if the function is a funarg, **arglists** is not restructured — this can be done when the body of the funarg is evaluated.

The function **restructure** needs to be given the adicity of the function to be applied (i.e., the number of arguments it requires). For lambda abstractions, this can be achieved by counting the number of binding variables, but for primitive (system) functions requires that their adicity be explicitly specified. In a "pure" LISP system, this is straightforward.

However, a problem arises in conventional LISP systems because some functions (e.g., lexprs and various system functions) can take an arbitrary number of arguments.

For example, consider the following conditional expression:

```
((cond (x (function car))
       (y (function cdr)))
  (f z))
```

The intent here is that, depending on the values of **x** and **y**, the conditional expression will yield the function **car** or **cdr** which in turn will be applied to the value of the expression **(f z)**. The problem is that, if **cond** itself can be partially applied, an alternative interpretation is that **(f z)** could be taken as another argument to **cond**. Indeed, in this case the problem is further compounded because, under the intended interpretation, the expression **(f z)** should be evaluated before evaluation of the conditional expression, while under the alternative interpretation, **(f z)** should remain unevaluated until used by **cond**.

To handle this, we either need to restrict such functions to being basic valued, or forbid them from being partially applied. The latter alternative is the one chosen here, as partial application is essentially a notational convenience whereas function-valued functions have semantic import.

The function **restructure** is as follows:

```
function restructure (m arglists)
  if (lesseq m (length (first arglists))) then arglists
  else (restructure m (join (first arglists) (rest arglists)))
end.
```

The function **join** restructures **arglists** by appending the first argument set in **arglists** to the second.

§4 Implementation in FRANZ LISP

In this section we outline the structure of the FRANZ LISP interpreter (hereafter called FRANZ) and describe the changes necessary to implement the function-deferring scheme for handling functional arguments and values.

4.1 The Structure of FRANZ LISP

The **eval** function used by FRANZ is conceptually very similar to the LISP interpreter described in Section 3. However, function calls are minimized to improve efficiency. The **eval** function thus performs all the evaluation, instead of sharing the work with **apply**. All function applications (like **car** and **cdr**) are removed from **apply** and are bound to the appropriate atoms as bcd (binary-coded-decimal) functions. The only time **apply** is called is when the function **apply** is explicitly applied in a user-defined expression. A detailed description of **eval** is given in Appendix 2.

The environment structure is implemented by using shallow binding. However, FRANZ does not have any mechanisms for supporting statically-scoped functional arguments or values.

FRANZ uses three stacks for managing evaluation. One of these is the C runtime stack, which is used by the LISP kernel for storing return addresses, non-LISP arguments to subroutines, and saved registers.

The other two stacks are used exclusively for interpreting LISP expressions. The first is used for representing the environment structure and is called the *bindstack*. As mentioned above, shallow binding is used.

The second stack is called the *namestack*. It is used for storing the values of the operands involved in function application (i.e., for passing parameters to LISP functions). It simply contains all current activations of the argument lists that, in the interpreter of Section 3, were passed as the arglist parameter of the **apply** function. It also holds local data within LISP functions and doubles as a temporary storage area for LISP data that must be protected from garbage collection.

To ensure that functions get the right number of arguments, it is important that the extent of each arglist on the namestack be known. This is achieved by using a variable (**lbot**) that is [effectively] local to **eval**, and setting it to the top of the stack just prior to pushing the arguments at this level of evaluation. Thus, at each call to **eval**, the variable **lbot** denotes the base address of the current arglist.

The base address of each arglist is also needed for other purposes. In any function application, the list of operands will be scanned from left to right, and hence the operand values will be placed on the stack with the *rightmost* operand on top. This causes no problem for the primitive (system) LISP functions, which expect their arguments in this order. However, for user-defined lambda abstractions, the binding variables are also scanned from left to right, and so the first variable to be bound will be the *leftmost* one. This means that the binding of variables must start at the *base* of the current arglist, successive bindings being made as we move *up* the namestack to the top.

For the sake of efficiency, both the bindstack and the namestack use contiguous memory elements; allocation and deallocation are then implemented simply by respectively incrementing and decrementing a pointer.

Let us now consider the implementation of the deferred-evaluation scheme. A detailed description of the modified **eval** function can be found in Appendix 3.

4.2 Saving the Bindstack

FRANZ has no facilities for simple funargs, let alone upward funargs. Thus, the first thing that has to be implemented is a mechanism for handling simple funargs. Once this is done, we can consider how to implement upward funargs, using the function-deferring scheme.

Implementation of simple funargs requires that, when a funarg is applied, the bindstack be unwound to the binding environment of the funarg and the application environment saved for restoration at the conclusion of the application. This is quite straightforward and can be achieved using a single stack (for example, see [Allen 1978]). However, the existing structure of the FRANZ LISP bindstack is not suitable for such a solution, and consequently a second bindstack is introduced for retaining the old variable bindings when the primary bindstack is unwound. This second bindstack is called the *saved-bindstack*.

It is clear that, as the bindstack is unwound, the saved-bindstack will grow, and that, as the bindstack is restored, the saved-bindstack will shrink. This behavior allows both stacks to be conveniently placed at either end of the same address space, both growing towards each other.

An example of the development of both bindstacks during the application of a funarg is given in Figure 1. The example is the same one as that considered in Section 2, where we had the following function definitions:

```
(defun appl (f x)
        (f x))
(defun fnplus (x)
        (lambda (y) (+ y x)))
```

The expression to be evaluated is

```
(appl (function (fnplus 10)) 20)
```

First, the operands of **appl** are evaluated and pushed onto the namestack (Figure 1.1).

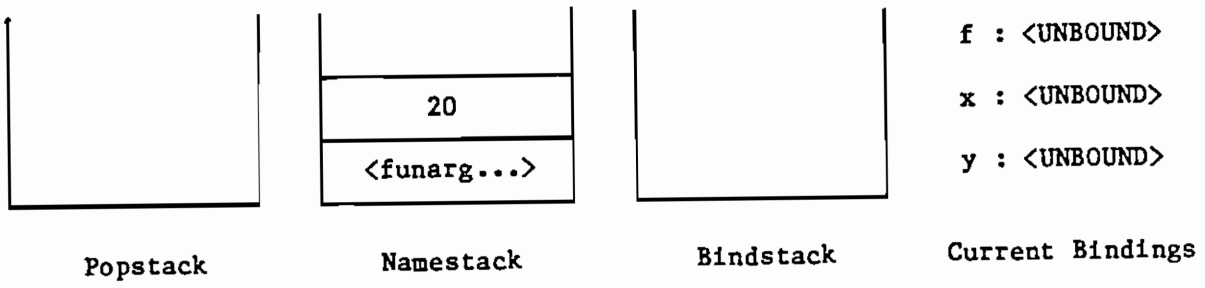

Figure 1.1. Development of the Bindstack

The variable **x** is then bound to 20 and **f** is bound to the funarg <FUNARG **(lambda (x) (lambda (y) (+ y x)))** (10) *env*>. (Note that the operand values remain on the namestack until we return from the instance of **eval** that put them there.)

The body of the function **appl** is then entered. First, the value of **x** is pushed onto the namestack for the call to **f** (Figure 1.2).

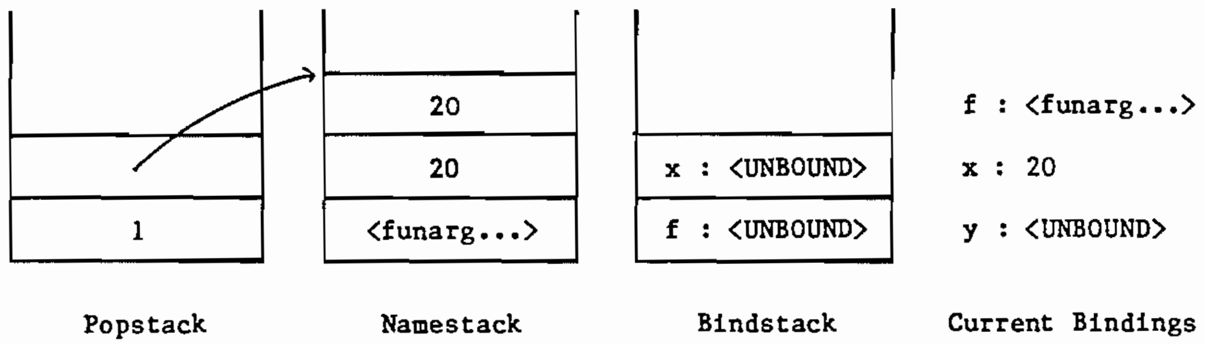

Figure 1.2. Development of the Bindstack (cont.)

As **f** is bound to a funarg, the environment must next be restored to the binding environment of the funarg (*env* in this case). As we want to recover the old bindings later, they are placed on the saved-bindstack (i.e., at the other end of the bindstack space) until needed. Now the arglist held by the funarg is also placed on the namestack (Figure 1.3).

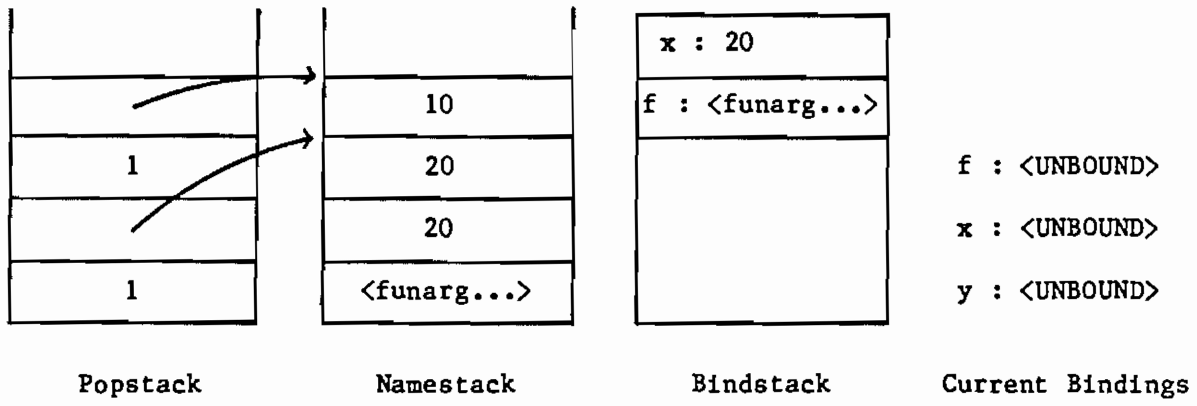

Figure 1.3. Development of the Bindstack (cont.)

Evaluation continues in the normal manner; the first lambda of the funarg is given the first block of arguments on the namestack, binding **x** to 10. The second lambda is given the second block of arguments, binding **y** to 20. Finally, the values of **y** and **x** are pushed onto the namestack for the call to "+" (Figure 1.4).

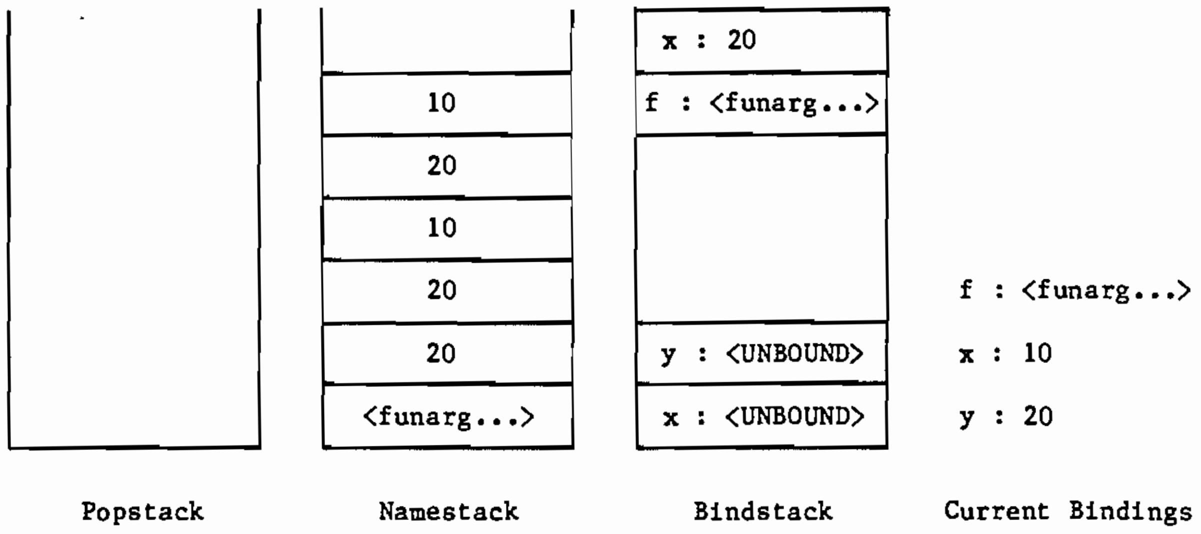

Figure 1.4. Development of the Bindstack (cont.)

The bcd function "+" returns 30, whereupon **eval** returns, taking 10 and 20 off the namestack. The two lambdas of the funarg return, and the evironment prior to the lambda application is restored (Figure 1.5).

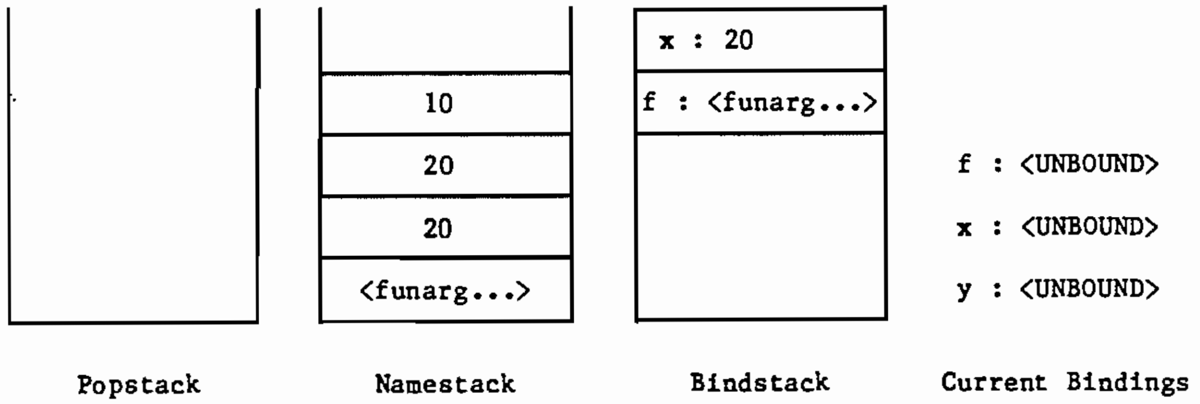

Figure 1.5. Development of the Bindstack (cont.)

At this stage, the initial application environment is restored by using the old bindings on the saved-bindstack (Figure 1.6). The funarg returns, all the calls to **eval** finish, and the value 30 is returned.

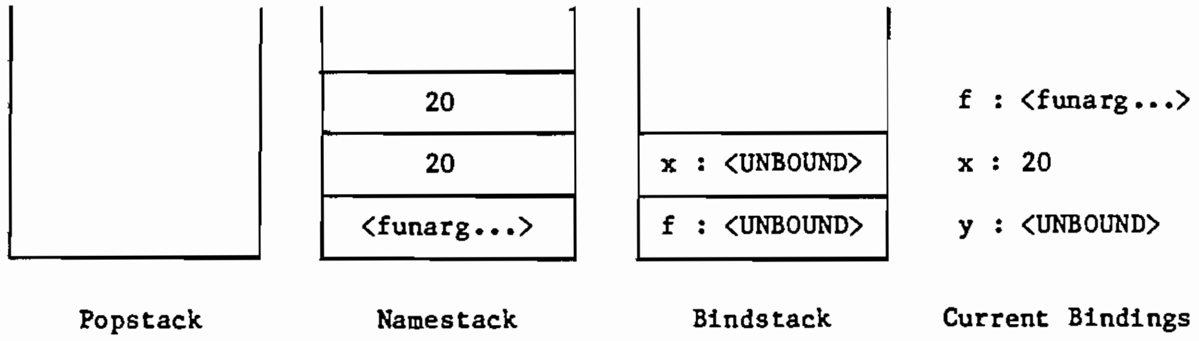

Figure 1.6. Development of the Bindstack (cont.)

4.3 Handling Function Application

Having modified the environment representation to support the handling of simple funargs, we now need to consider what changes are necessary to allow implementation of the deferred-evaluation scheme for upward funargs. The major change that needs to be made is to allow the arglist to continue accruing arguments until application of a function is possible (as is done in the interpreter described in Section 3).

Under the proposed evaluation scheme, the execution of a function-valued function is deferred until there are enough arguments for the expression to reduce to a basic (i.e., nonfunctional) value. Only when sufficient arguments have been accumulated is the function-valued function applied. The resultant function is then applied to the remains of the accumulated arguments, and so on, until a basic value is finally returned as the value of the expression. With this approach, by the time a functional value is applied, it is already in its correct binding environment.

The only difficulty with implementing this scheme is that, at the time any function is to be applied, it must be capable of reaching down the namestack far enough to collect all the arguments it needs for the application. In the standard FRANZ LISP interpreter, the current arglist corresponds to the block of arguments pushed onto the namestack at the current call to **eval**, and thus the base address of this arglist is represented by the top of the stack prior to pushing the arguments. However, under the function-deferring scheme, the arglist must include enough arguments to allow reduction to a basic value, and the base of this arglist may not correspond to the top of the stack at the *current* call to eval. It corresponds, in fact, to the top of the stack when the evaluation of the enclosing basic-valued expression is initialized. This is not difficult to implement, requiring only a simple change in the way the base address of the current arglist is determined.

However, in addition to the foregoing, we still need to store the address of the top of the stack on each call to **eval**. This is necessary to enable the correct binding of variables and to distinguish argument blocks from temporary information that is also stored on the namestack. Furthermore, this information must be globally accessible, and therefore cannot be represented as a simple variable local to the **eval** function.

The address of the top of the namestack on each call to **eval** could very easily be placed on the namestack along with other temporary information. However, so as to minimize changes in the current data structures, this information is stored instead on another stack, called the *popstack*. Thus, in effect, the popstack records the storage locations of the relevant argument blocks on the namestack.

The popstack is used only in evaluating expressions whose operator part is either a funarg or a function-valued applicative expression. In these cases, two entries are pushed onto the popstack for every set of arguments placed on the namestack. The first is the address of the top of the set and the second is the number of arguments in the set. This effectively gives the base address of the block.

Figure 2 contains an example of the development of the popstack and namestack during evaluation of the basic-valued expression

((f1 arg1) arg2 arg3)

Assume the current situation is as shown in Figure 2.1.

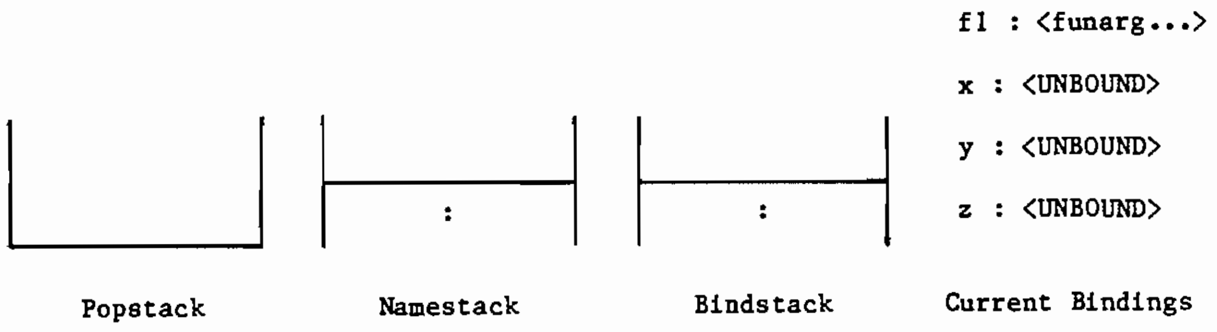

Figure 2.1. Development of the Popstack and Namestack

To evaluate the expression, the operands **arg2** and **arg3** are evaluated, yielding *arg2'* and *arg3'* respectively. These values are placed on the namestack. Because the operator **(f1 arg1)** is a function-valued applicative expression, the popstack is used to record the extent of the argument block at this level of evaluation (Figure 2.2).

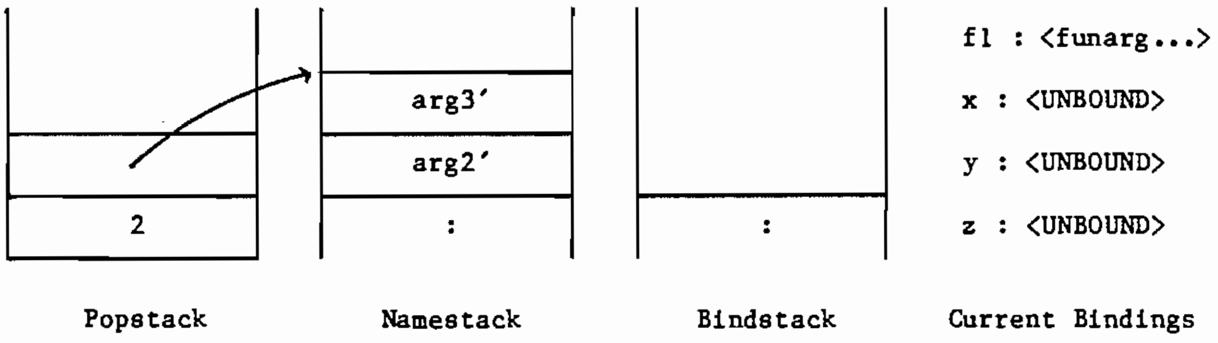

Figure 2.2. Development of the Popstack (cont.)

Then the subexpression **(f1 arg1)** is evaluated. The operand **arg1** is first evaluated, yielding *arg1'*, which is pushed onto the namestack. The extent of this argument block is recorded on the popstack (Figure 2.3).

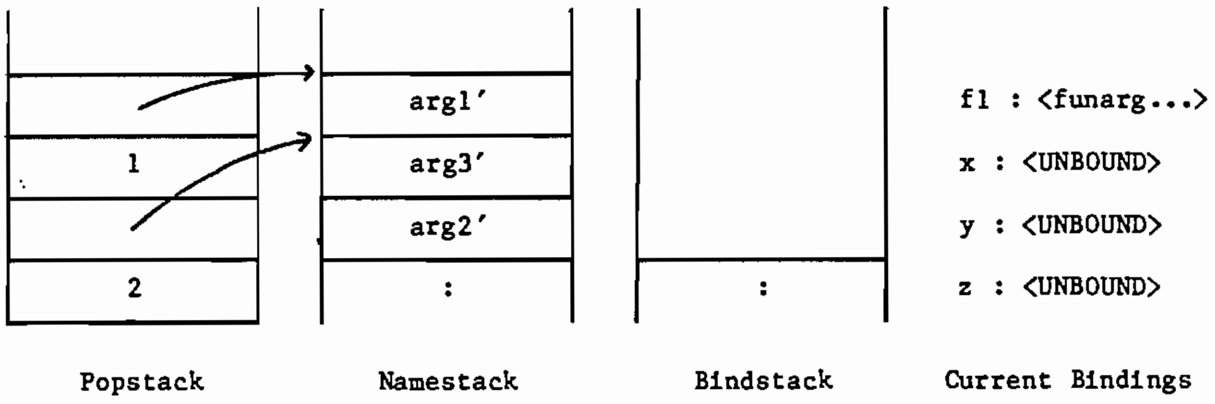

Figure 2.3. Development of the Popstack (cont.)

At this point, the function **f1** is to be applied to its arguments on the namestack. Let us assume that **f1** is bound to the funarg <FUNARG **(lambda (x) (lambda (y z) (plus x y z)))** nil env>.

The first thing that happens is that any arguments in the arglist of the funarg are pushed onto the namestack. In this case, the arglist of the funarg is empty and so nothing is done. Next, the binding environment of the function is restored (as described above), and the body of the funarg is entered. Now the binding variables (formal parameters) get bound to the arguments in the first argument block on the namestack, pointed to by the popstack. In this case **x** gets bound to the value of **arg1** (*arg1'*). The corresponding information on the popstack is popped off, since the argument block is no longer needed. Thus the arguments are no longer accessible. Note, however, that the actual values remain on the namestack (i.e., the top of the namestack is unchanged) until we return from the instance of **eval** that put them there.

The environment is extended to include this new variable-value pair and the body **(lambda (y z) (plus x y z))** of the abstraction is evaluated. The situation is shown in Figure 2.4. The binding variables are bound to the arguments in the next argument block, pointed to by the popstack. In this case, the variables **y** and **z** are now bound to the values of **arg2** (*arg2'*) and **arg3** (*arg3'*). The corresponding information on the popstack is popped off, as this argument block is also no longer needed.

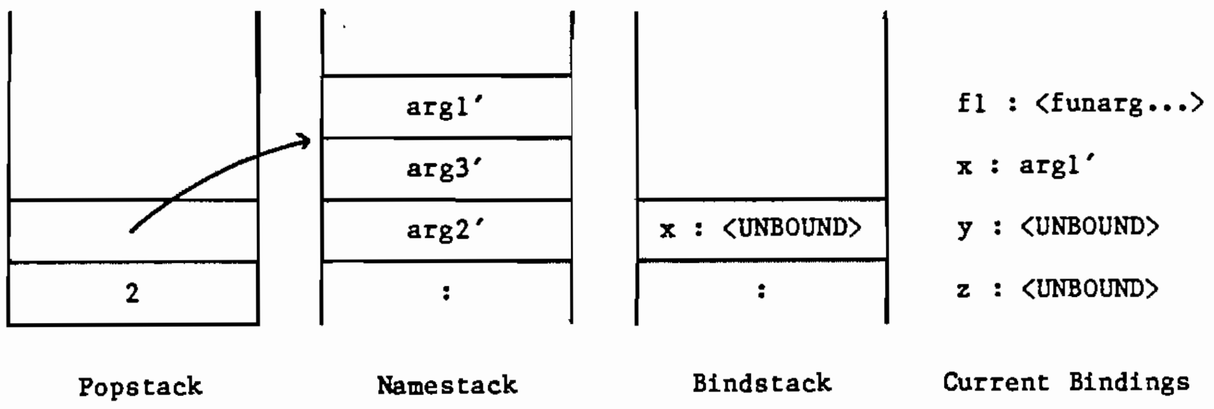

Figure 2.4. Development of the Popstack (cont.)

The environment is extended by these new variable-value pairs and the body **(plus x y z)** evaluated in this new environment. The values of the variables **x**, **y**, and **z** are placed on the namestack, but the corresponding block address information is not placed on the popstack, as the expression under evaluation is of standard form (Figure 2.5).

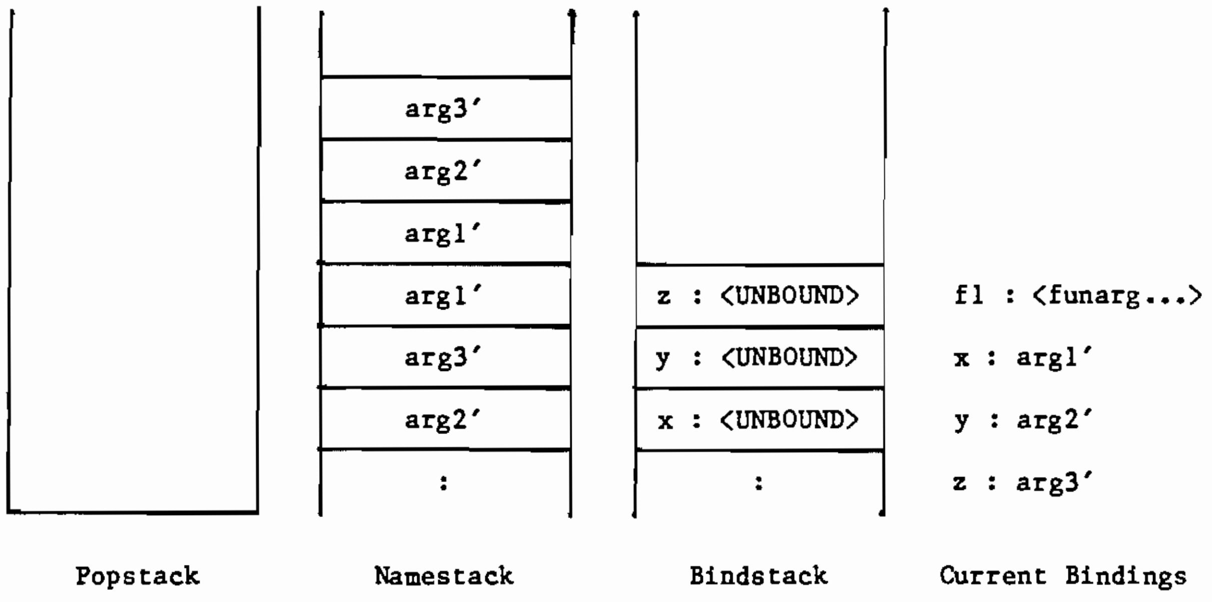

Figure 2.5. Development of the Popstack (cont.)

The primitive **plus** function is then applied to the arguments on the namestack and the result returned as the value of **eval** at this level. This value is then returned as the result of each higher-level call to **eval** (restoring environments on the way) and eventually placed on the namestack as the value of the original expression (Figure 2.6).

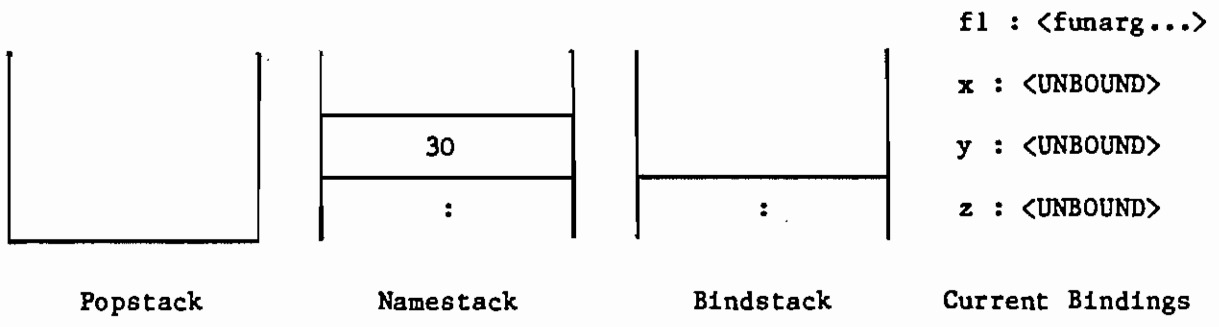

Figure 2.6. Development of the Popstack (cont.)

4.4 Providing for Partial Application

As discussed in Section 3.4, for functions that can produce functional results, the interpreter must know the exact number of arguments expected by the function. This required two changes in the FRANZ LISP interpreter: one to mark those functions unsuitable for partial application, the second to provide information on the number of arguments required by functions.

Marking particular functions as unsuitable for partial application is straightforward, but it is more difficult to provide information on the number of arguments required by LISP functions. For user defined functions this is a trivial matter, as the number of binding variables can simply be counted. However, because system primitives are in binary, it is necessary to associate with each bcd function the number of formal parameters it requires, or a special value (taken to be **anynumber**) indicating that the function will accept any number of arguments (as, for example, is required by lexprs).

4.5 Implementation

The scheme outlined above has been implemented by modifying the FRANZ LISP (Opus 36b) interpreter. The modified version of the evaluator is given in Appendix 3. The modifications were both simple to make and did not affect the efficiency of interpretation.

There are, however, some restrictions in the current implementation. First, if the function **function** is applied to an expression, it will evaluate the operands occurring in the expression in the current environment and will save that environment pending evaluation of the operator. Thus the operator is treated as a lambda or lexpr discipline (i.e., its arguments are evaluated). There is no provision for the other disciplines like nlambda and macro.

One solution is that **function** can take a second argument describing the discipline of the operator. Another possible solution is to have variations of the function **function**, say, **function** for lambda function bindings, **nfunction** for nlambdas and **mfunction** for macros.

Second, the current system does not allow compilation of function-valued applicative expressions, function-valued arguments (with static scoping) or partial applications; only the interpreter can evaluate such expressions. To rectify this deficiency, compiled LISP code has to be altered so that the number of arguments is known and the last expression evaluated can use unapplied arguments from previous applications, as in the case of interpreted code. In addition, of course, the interpreter for the compiled code would have to be modified along the same lines as the LISP interpreter. Alternatively, the compiler could first transform such expressions into a form that avoids upward funargs (see [Georgeff 1984]).

Third, in the current implementation, some constructs cannot be used freely with function-valued functions. Nonlocal gotos cannot be used with function-valued functions or partial application. Partial application of array references has also been ignored. Furthermore, the use of do and **prog** will not allow arguments from previous (enclosing) applications to be employed by expressions within their bodies, as the last expression executed is a return statement.

§5 Examples

In this section we provide some examples of the use of function-valued expressions and partial application. We assume that the reader is familiar with the standard FRANZ LISP language as described in the FRANZ LISP Manual [Foderaro 1980].

There are two major extensions of the language: the use of the **function** construct to preserve static scoping of function-valued expressions, and the introduction of partial application.

5.1 Function-Valued Operands

In the extended version of FRANZ LISP, the **function** construct is used to preserve static scoping of function-valued expressions. It is thus different from the **function** construct in standard FRANZ LISP, which does not preserve static scoping. It will work in the same way as the **function** construct in MACLISP [Touretsky 1978], but is more general in that it will also work for upward funargs — which MACLISP does not. Furthermore, it differs from the use of the **function** construct in various versions of full LISP, in which it is used at the point a function is to be returned as a value from within a function-valued function.

In the extended version of FRANZ LISP, the function construct is used simply *whenever one has a function-valued operand* for which static scoping is desired. That is, the **function** construct is applied to function-valued expressions that occur in operand position.

Some examples are given below:

```
(defun increment (x)
        (lambda (y)
                (+ x y)))

(defun twice (f)
        (lambda (x)
                (f (f x))))

(defun double-increment (x)
(twice (function (increment x))))

(mapcar (function 1+) '(1 2 3))

            → (2 3 4)

(mapcar (function (lambda (x) (+ x 2))) '(1 2 3))

            → (3 4 5)

(mapcar (function (double-increment 3)) '(1 2 3))

            →  (7 8 9)
```

```
(mapcar (function (twice (function (increment 3)))) '(1 2 3))

        → (7 8 9)
```

Note that, since scoping rules are irrelevant for primitive functions, it was not necessary to use the **function** construct for the operand "1+"; "quote" would have sufficed equally well. However, it is good practice and yields more readable code if the function construct is used for all function-valued operands.

5.2 Function-Valued Operators

In FRANZ LISP and most other versions of LISP, the operator (function) position in an applicative expression must be a function constant (e.g., **car**), a variable denoting a function, or some kind of abstraction (e.g., a lambda abstraction). The extended version of FRANZ LISP, however, allows any function-valued expression to occur in operator position. Some examples are given below.

```
((increment 3) 2)

        → 5

(+ ((increment 3) 2) 5)

        → 10

((twice (function (lambda (x) (+ x 2)))) 5)

        → 9

((car (list (function (increment 3))(function (increment -3)))) 10)

        → 13

((cond (x (function 1+))(t (function 1-))) 10)
```

→ **11** (if x is non-nil)

→ **9** (if x is nil)

5.3 Partial Application

The extended version of FRANZ LISP also allows most functions to be partially applied to a subset of their arguments. Such partial application is not allowed in any other version of LISP. However, a similar facility exits in the language POP2 [Burstall et al. 1971]. Some examples follow:

```
(mapcar (function (+ 3)) '(1 2 3))

        → (4 5 6)
```

This expression is equivalent to the expression

```
(mapcar (function (lambda (x) (+ 3 x))) '(1 2 3))
```

However, the partially-applied form is more efficient in evaluation than the latter expression. Under partial application, the variable **x** is evaluated only once (at the time the functional argument is evaluated), whereas in the latter case the lambda form prevents evaluation of **x** until application time (thus resulting in **x** being evaluated three times). Of course, because of the difference in evaluation times, the two above expressions may yield different values in the presence of side effects.

The next example shows how partial application can be used directly in operator position (albeit, in this case, to no advantage).

```
((cons 'a) '(b c d))

        → (a b c d)
```

This expression is equivalent to

```
(cons 'a '(b c d))
```

§6 Conclusions

A simple and efficient scheme for handling both simple and upward funargs in LISP has been described and implemented. This allows the definition and use of higher-order functions that provide static scoping for free (global) variables. Furthermore, the scheme enables partial application of functions to a subset of their arguments, thus providing a simple means of generating function-valued functions from basic-valued functions.

The essence of the scheme is to defer evaluation of function-valued expressions until sufficient arguments exist to allow the expression to be reduced to a nonfunctional value. In this way, upward funargs are converted into simple funargs and a standard runtime stack can be used for managing evaluation.

Experimentation shows that, in most cases, there are less environment swappings under this scheme than in other implementations. The scheme offers the added advantage that, not only can a simple stack be used, but no awkward processing, such as garbage collection on the stack space, is needed. Indeed, given a LISP system that handles simple funargs, the modifications necessary to handle upward funargs as well are extremely straightforward.

The scheme has been implemented by modifying an existing FRANZ LISP interpreter. The implementation handles interpreted expressions only, and function-valued functions must be either of lambda or lexpr discipline. To qualify as a practical LISP system, the scheme needs to be extended to enable compilation of expressions involving function-valued functions as well as their interpretation. Techniques for doing this are described elsewhere [Georgeff 1982, 1984].

It is likely that more efficient and better-structured code could be produced if the current scheme were implemented from scratch, or if an interpreter already endowed with facilities for handling simple funargs were modified. Further simplification would result if partial application were disallowed. This feature proves more difficult to implement in a full LISP system than in an abstract interpreter where efficiency is not at issue. Although the wisdom of extending the LISP language to include partial application is open to question, it may prove a useful addition to other, less standardized, applicative languages.

Acknowledgment

The authors wish to thank Rodney Topor for his helpful comments on an earlier draft of this report and for finding an important bug in our first version of the abstract LISP interpreter.

Appendix 1: LISP Interpreter with Full Funargs

The full listing for the **eval** function described in Section 3 is given below. Note that to compare approaches to the handling of higher-order functions, the interpreter includes both the conventional scheme for handling funargs (which must allow for tree-structured environments) and the function deferring scheme described in this report (which can be implemented on a simple stack). Which scheme is used during evaluation depends on how the programmer uses the function FUNCTION (see Sections 2.1 and 2.3). Of course, the environment management procedures given below would be considerably simplified by implementing the function deferring scheme alone.

Descriptions of the important functions are given in Section 3 of this report. The environment structure is implemented using shallow binding, and is based on the method described in [Allen 1978]. The global variable **root** always points to the current environment. For ease of tracing the enviroment structure, each node in the environment is named by a unique symbol. This can be changed (and the interpreter made more efficient) by modifying the functions **ptr** and **frame**.

§1 Main Routines

```
(defun read-eval-print ()
  (prog (exp root)
        (setq root (ptr (list nil (ptr nil) 'active)))
        loop
            (print 'EXPRESSION:)
            (setq exp (read))
            (cond ((atom exp) (print (EVALNEW exp nil)))
                  ((is-exit (func exp))(return 'exit))
                  ((is-defun (func exp)) (print (declare exp)))
                  (t (print (EVALNEW exp nil))))
            (terpri)(terpri)
        (go loop)))

(defun EVALNEW (exp arglists)
  (cond ((is-const exp) (APPLYNEW (denote exp) arglists))
        ((is-var exp) (APPLYNEW (lookup exp) arglists))
        ((is-cond exp) (APPLYNEW (evcond (argsc exp)) arglists))
        ((is-funcons exp) (APPLYNEW (mkfunarg (fbody exp)) arglists))
        ((is-application exp) (EVALNEW (func exp) (cons
                                         (evlis (args exp)) arglists)))))

(defun lookup (var) (get var 'value))

(defun denote (exp)
  (cond ((is-number exp) exp)
        ((is-truth exp) exp)
        ((is-false exp) nil)
        ((is-sexpr exp) (rep exp))
        ((is-lambda exp) exp)))

(defun evcond (exp)
   (do ((x exp (rest x)))((null x) nil)
           (cond ((EVALNEW (ante (first x)) nil)
                      (return (EVALNEW (conseq (first x)) nil))))))
```

```
(defun mkfunarg (exp)
        (cond ((is-application exp)
                (list 'FUNARG (func exp)(list (evlis (args exp)))  root))
              (t (list 'FUNARG  exp nil root))))

(defun evlis (exp)
  (mapcar '(lambda (x) (EVALNEW x nil)) exp))

(defun APPLYNEW (fn arglists)
 (prog (newarglists args remains)
   (return
   (cond ((null arglists) fn)
     (t (setq newarglists (restructure (adicity fn) arglists))
        (setq args (first newarglists))
        (setq rest (rest newarglists))
        (cond ((is-car fn) (APPLYNEW (car (arg1 args)) remains))
        ((is-cons fn)       (cons (arg1 args) (arg2 args)))
        ((is-cdr fn)        (APPLYNEW (cdr (arg1 args)) remains))
        ((is-times fn)      (* (arg1 args) (arg2 args)))
        ((is-plus fn)       (+ (arg1 args) (arg2 args)))
        ((is-diff fn)       (- (arg1 args) (arg2 args)))
        ((is-null fn)      (null (arg1 args)))
        ((is-set fn)       (assign (arg1 args) (arg2 args)))
        ((is-lambda fn)    (apply-abs (body fn) (vars fn) newarglists))
        ((is-funarg fn)     (apply-funarg (fbody fn)
                                      (append (fargs fn) arglists)
                                      (fenv fn))))))))))


(defun declare (exp)
  (assign (def-name exp)
          (cons 'lambda (def-body exp)))
  (def-name exp))
```

§2 Stack Manipulation

```
(defun apply-abs (body vars vals)
  (prog (result)
        (pushvars vars)
        (assign-vars vars (first vals))
        (setq result (EVALNEW body (rest vals)))
        (popvars)
        (return result)))

(defun pushvars (vars)
    (mknode (mapcar 'val-cell vars) root))

(defun popvars ()
        (replace (typeptr root) '(binding))
        (swap-env root)
        (setq root (parent root)))

(defun assign-vars (vars vals)
  (mapcar 'assign vars vals))

(defun apply-funarg (body args envbinding)
  (prog (result envappln)
        (setq envappln root)
        (setenv envbinding)
        (setq result (EVALNEW body args))
        (setenv envappln)
        (return result)))

(defun setenv (ptr)
  (prog (nodepair)
     (setq nodepair (find-intersect ptr))
     (unwind-active (first nodepair))
     (unwind-bind (rest nodepair))))
```

```
(defun find-intersect (envptr)
  (do ((next nil ptr)
            (ptr envptr parent)
            (parent (parent envptr)(parent parent)))
       ((is-active ptr)
            (conspair ptr next))
       (swap-links ptr next)))

(defun unwind-active (envinter)
  (do ((ptr root parent)
            (parent (parent root)(parent parent)))
      ((eq  ptr envinter)
            (setq root ptr))
      (swap-env ptr)
      (replace (typeptr ptr) '(binding))))

(defun unwind-bind (envptr)
  (do ((next root ptr)
            (ptr envptr parent)
            (parent))
      ((null ptr)
            (setq root next))
      (setq parent (parent ptr))
      (swap-links ptr next)
      (replace (typeptr ptr) '(active))
      (swap-env ptr)))

(defun swap-links (x y)
        (rplaca (parentptr x) y))

(defun swap-env (ptr)
  (prog (temp)
        (setq temp (binding ptr))
        (cond ((null temp) (return nil)))
        (replace (binding ptr) (mapcar 'swap-var temp))))
```

```
(defun swap-var (var-val)
  (swap (variable var-val) (value var-val)))

(defun swap (var val)
  (prog (temp)
        (setq temp (val-cell var))
        (assign var val)
        (return temp)))
```

§3 Recognizers

```
(defun is-defun (x) (eq x 'defun))
(defun is-exit (x) (eq x 'exit))

(defun is-const (x)
  (or (is-truth x)
      (is-false x)
      (is-number x)
      (cond ((atom x) nil)
            ((is-sexpr x))
            ((is-lambda x)))))

(defun is-var (x) (atom x))
(defun is-car (x) (eq x 'CAR))
(defun is-cons (x) (eq x 'CONS))
(defun is-cdr (x) (eq x 'CDR))
(defun is-times (x) (eq x 'TIMES))
(defun is-plus (x) (eq x 'PLUS))
(defun is-diff (x) (eq x 'DIFF))
(defun is-null (x) (eq x 'NULL))
(defun is-set (x) (eq x 'SET))
(defun is-truth (x) (eq x 't))
(defun is-false (x) (eq x 'nil))
(defun is-number (x) (numberp x))
(defun is-sexpr (x) (eq (first x) 'quote))
```

Warning: The LISP reader may automatically convert "quote" to the symbol '. In this case, one would have to replace "quote" in the preceding definition with some atom (e.g., "QUOTE") which would not be modified by the LISP reader.

```
(defun is-lambda (x) (eq (first x) 'lambda))
(defun is-cond (x) (eq (first x) 'cond))
(defun is-funcons (x) (eq (first x) 'function))
(defun is-funarg (x) (eq (first x) 'FUNARG))
(defun is-application (x) (not (or (is-const x)(is-var x))))
(defun is-active (x) (eq 'active (type x)))
```

§4 Selectors

```
(defun func (x) (first x))
(defun args (x) (rest x))
(defun fbody (x) (second x))
(defun fargs (x) (third x))
(defun fenv (x) (nthelem 4 x))
(defun body (x) (third x))
(defun vars (x) (second x))
(defun argsc (x) (rest x))
(defun arg1 (x) (first x))
(defun arg2 (x) (second x))
(defun ante (x) (first x))
(defun conseq (x) (second x))
(defun rep (x) (second x))
(defun def-name (x) (second x))
(defun def-body (x) (cddr x))
(defun value (x) (second x))
(defun variable (x) (first x))
(defun frame (x) (eval x))
(defun binding (x) (first (frame x)))
(defun parent (x) (second (frame x)))
(defun parentptr(x) (cdr (frame x)))
```

```
(defun type (x) (third (frame x)))
(defun typeptr (x) (cddr (frame x)))

(defun first (x) (car x))
(defun second (x) (cadr x))
(defun third (x) (caddr x))
(defun rest (x) (cdr x))
```

§5 Constructors

```
(defun val-cell (var)
  (list var (get var 'value)))

(defun assign (var val)
  (putprop var val 'value))

(defun mknode (env parent)
  (setq root (ptr (list env parent 'active))))

(defun ptr (val)
  (prog (a)
    (setq a (gensym))
    (set a val)
    (return a)))

(defun restructure (m arglists)
    (cond ((lesseq m (length (first arglists))) arglists)
           (t (restructure m (join (first arglists) (rest arglists))))))

(defun join (l1 l2)
    (rplaca l2 (append l1 (first l2))))

(defun conspair (x y)
   (cons x y))
```

```
(defun adicity (fn)
  (cond ((member fn '(CAR CDR NULL)) 1)
        ((member fn '(CONS TIMES PLUS DIFF EQUAL SET)) 2)
        ((is-lambda fn)(length (vars fn)))
        (t 0)))

(defun lesseq (x y)
   (or (= x y) (< x y)))
```

§6 Compatibility

For MACLISP compatibility the following functions need to be defined:

```
(defun replace (x y)
   (rplaca x (car y))(rplacd x (cdr y)))

(defun nthelem (n l)
  (nth  (1- n) l))
```

For Common LISP you will need, in addition, the following function:

```
(defun putprop (var val key)
  (setf (get var key) val))
```

Furthermore, the variable **root** must be proclaimed *special*.

§7 Basic Functions Used

To run the system, the following basic functions (or something similar) must be loaded:

```
(assign 'mapcar
  '(lambda (fcn l)
        (cond ((null l) nil)
              (t (cons (fcn (car l))
                       (mapcar fcn (cdr l)))))))
```

```
(assign 'car 'CAR)
(assign 'cdr 'CDR)
(assign 'cons 'CONS)
(assign 'times 'TIMES)
(assign '* 'TIMES)
(assign 'plus 'PLUS)
(assign '+ 'PLUS)
(assign 'diff 'DIFF)
(assign '- 'DIFF)
(assign 'null 'NULL)
(assign 'equal 'EQUAL)
(assign '= 'EQUAL)
(assign 'set  'SET)
```

Appendix 2: The Structure of FRANZ LISP

§1 Data Types

In any real LISP system, the types of objects the system can manipulate are much more complex and varied than the simple atoms and lists assumed for the simplified interpreters discussed in Section 3. FRANZ's most basic data structure is a LISP object, called *lispobj*. These lispobjs are used to represent all the data in the system. For our purposes, we need only know the following:

1. An atom, referred to as *a*. Atoms hold most user information and include a current level binding, referred to as *clb*, plus any function binding (*fnbnd*).

2. A list node (dotted pair), referred to as *dtpr*. List nodes have associated with them one pointer to a LISP object (*car*) and another pointer to the rest of the list (*cdr*).

3. A LISP object, used as a value, referred to as *l*.

4. A pointer, referred to as *bcd*, to a formal descriptor (*bfun*) for executable binary code. The bfun structure is used to store all executable binary functions; hence the name bcd (binary-coded-decimal). These structures have associated with them a pointer to the entry point (*entry*) and the discipline of the function (lambda, nlambda, etc.), referred to as *discipline*.

§2 The Structure of EVAL

The **eval** function used by FRANZ is conceptually very similar to the LISP interpreter described in Section 3. Function calls are minimized to improve efficiency; thus **eval** performs all the evaluation, instead of sharing the work with **apply**.

As discussed in section 4, FRANZ uses three stacks for managing evaluation. One of these is the C runtime stack, which is used by the LISP kernel for storing return addresses, non-LISP arguments to subroutines, and saved registers. The other two stacks, called the bindstack and the namestack, are used exclusively for interpreting LISP expressions. The bindstack is employed for representing the environment structure, using a shallow binding scheme. The namestack is used for storing the values of the operands involved in function application (i.e., for passing parameters to LISP functions). It also holds local data within LISP functions and doubles as a temporary storage area for LISP data that must be protected from garbage collection.

The structure of the FRANZ LISP eval function is given below in pseudocode. Indentation is used instead of brackets to show structure.

```
function eval(actarg)
    a = actarg;
    oldbindstack = bindstack;
    savestack();
    if isatom(a) then
        restorestack();
        return(clb(a));
    elseif isvalue(a) then
        restorestack();
        return(l(a));
    elseif isdtpr(a) then
        lbot = namestack;
        a = func(a);
        if isatom(a) then
            a = funcbind(a);
        elseif isvalue(a) then
            a = l(a);
        argptr = arglist(actarg);
        if isbcd(a) then
            if isnlambda(discipline(a)) then
                push(argptr,namestack);
            elseif ismacro(discipline(a)) then
                push(actarg,namestack);
            else {* islambda(discipline(a)) or islexpr(discipline(a)) *}
```

```
                map(pushnamestack,map(eval,argptr));
            vtemp = call(a);
        elseif isarray(a) then
            vtemp = array(a,argptr);
        elseif isdtpr(a) then
            if islambda(func(a)) then
                map(pushnamestack,map(eval,argptr));
            elseif isnlambda(func(a)) then
                push(argptr,namestack);
            elseif ismacro(func(a)) then
                push(actarg,namestack);
            elseif islexpr(func(a)) then
                map(pushnamestack,map(eval,argptr));
                push(namestack - lbot, namestack);
                lbot  = namestack - 1;
            workp = lbot;
            while not isempty(bvars(a)) do
                push(bind(next(bvars(a)),top(workp)),bindstack);
                workp = workp + 1;
            vtemp = last(map(eval,body(a)));
        unbind(oldbindstack);
        if ismacro(func(a)) then
            vtemp = eval(vtemp);
        restorestack();
        return(vtemp);
    else {* isconstant(a) *}
        restorestack();
        return(a);
```

The recognizers, constructors, and selectors used in the above procedure should be clear from their names. Some important functions are the following:

funcbind:	extracts the function binding of its argument
push:	pushes the first argument onto the stack named as its second argument
pushnamestack:	pushes its argument onto the namestack

call:	executes bcds
bind:	performs shallow binding of the first argument to the second, and returns the old binding
unbind:	restores old bindings
savestack:	saves the value of **namestack** and **lbot**
restorestack:	restores the old values for **namestack** and **lbot**

A more detailed description of these and other functions can be found in Appendix 4.

Note that, for the sake of clarity, we have made some minor modifications of the original FRANZ code: the variable **argptr** is assigned earlier and the in-line code for extracting function bindings is replaced by a call to the function **funcbind**.

The evaluation of an expression proceeds as follows:

1. Initialization

The expression to be evaluated, **actarg** (actual argument), is passed to **eval**. That expression is also assigned to the temporary variable **a**; **a** will eventually hold the actual function binding or array. The current top of the bindstack is then saved so that, when **eval** exits, the bindstack can be reinstated (thus restoring the values of locally used atoms). Similarly, the current top of the namestack and the variable **lbot** are saved by using **savestack**. These will be restored just prior to return from **eval**.

2. Type Determination and Evaluation

The second phase of evaluation hegins by determining the type of **eval**'s argument. The following actions occur, depending on the type:

atom:	Return its current level binding.
value:	Return its value.
dtpr:	Complete the third phase of processing and return the result.
constant:	Everything else is considered a constant and is simply returned.

3. Function Application

3.1. Initialization

The third phase of evaluation handles the application of functions to their arguments. First, the current top of the namestack is saved in a global variable **lbot** to indicate the bottom of the arglist. This is done so that all routines can determine how many arguments there are (by comparison with the top of the stack after pushing the arguments) and where they begin.

The operator (functional part) of the expression is then extracted and assigned to the variable **a**. If this is an atom (**isatom**), the variable **a** is reassigned to the actual function binding of the atom (**funcbind(a)**).

Finally, the variable **argptr** (argument pointer) is made to point to the actual operands to which the function is to be applied.

3.2. The Real Work

This is where the real work begins. The processing is now divided, depending on whether the functional part of the expression is a binary-coded-decimal function (**isbcd**, as in the case of compiled code), an array (**isarray**), or an expression representing an abstraction (**isdtpr**, as in the case of interpreted code).

Case 1: Binary-coded-decimal

If the function's discipline is nlambda (**isnlambda**), then **argptr** is pushed onto the namestack. This leaves the operands unevaluated and in list form on the stack.

If the function's discipline is macro (**ismacro**), then **actarg** is pushed onto the namestack. This leaves the original calling expression on the stack with the operands unevaluated.

All else is assumed to be of lambda discipline;[1] therefore, each operand is evaluated and pushed onto the namestack.

Finally, control is transferred to the bcd function by calling a special function, **call**. The result is placed in the variable **vtemp**.

Case 2: Array

The array is accessed by calling a special function **array**. The resulting array element is placed in the variable **vtemp**.

Case 3: Abstraction (dtpr)

[1]At this point, lexprs are represented and handled as lambdas.

If the abstraction's discipline is lambda (**islambda**), each operand is evaluated and pushed onto the namestack.

If the abstraction's discipline is nlambda (**isnlambda**), then **argptr** is pushed onto the namestack. This leaves the operands unevaluated and in list form on the stack.

If the abstraction's discipline is macro (**ismacro**), then **actarg** is pushed onto the namestack. This leaves the original calling expression on the stack, with the operands unevaluated.

If the abstraction's discipline is lexpr (**lexpr**), each operand is evaluated and is pushed onto the namestack. The number of arguments (the current namestack level minus the bottom of the arglist **lbot**) is then pushed onto the namestack, and **lbot** is made to point one position below the current namestack position. This in effect allows a lexpr function to have only one argument and leaves it to its own devices to extract the real arguments.

Now comes the task of binding the binding variables (formal parameters) to the arguments. First, a variable **workp** (as it does most of the work) is made to point to the first argument by using the **lbot** pointer. Then, moving *up* the namestack, each binding variable is bound to a value from the namestack and the old bindings pushed onto the bindstack. This corresponds to binding the evaluated arguments to the binding variables in a left-to-right fashion.

Finally, **eval** is mapped over all expressions appearing in the function body, and **vtemp** is assigned the result of the last evaluated expression.

3.3 Finishing Off

The old bindings are reinstated by using the saved bindstack pointer. If the function is a macro, its result is re-evaluated and assigned to **vtemp**. Finally, the top of the namestack and the variable **lbot** are restored to their original values by using **restorestack**, and **vtemp** is returned as the result of **eval**.

Appendix 3: Implementation of Deferred Evaluation

As discussed in the main body of the paper, implementation of the deferred-evaluation scheme required the introduction of two additional stacks: the saved-bindstack and the popstack. These are described in Section 4. In this section, we outline the changes in the evaluation routines.

To separate the modifications as much as possible from the original FRANZ code, the flow of control within **eval** was unchanged except for the evaluation of nonstandard functional expressions.

If the expression to be evaluated is function-valued (i.e., if there remain "un-applied" arguments from previous (enclosing) applicative expressions), **eval** redirects the processing to an auxiliary evaluator called **evalaux1**. The function **evalaux1** handles evaluation of all function-valued expressions.

In addition, if the operator (functional part) of an expression is either a funarg or an applicative expression, then another auxiliary routine called **evalaux2** is invoked to handle evaluation.

§1 Changes in **eval**

The function deferring **eval** function is given below. It is identical to the original **eval** except for the segments appearing in italics.

```
function eval(actarg)
    a = actarg;
    oldbindstack = bindstack;
    savestack();
    if not bottom(popstack) then
        vtemp = a;
        if isatom(a) then
            if not hasfnbnd(a) then
                if not hasclb(a) then goto skip;
                vtemp = clb(a)
                if not isatom(vtemp) or not hasfnbnd(vtemp) then goto skip;
            a = fnbnd(vtemp)
        elseif not isdtpr(a) then goto skip;
        restorestack();
        return(evalaux1(a));
skip:
    if isatom(a) then
        restorestack();
        return(clb(a));
    elseif isvalue(a) then
        restorestack();
        return(l(a));
    elseif isdtpr(a) then
        lbot = namestack;
        a = func(a);
        if isatom(a) then ....
        elseif isvalue(a) then ....;
        argptr = arglist(actarg));
        if isbcd(a) then ....
        elseif isarray(a) then ....
        elseif isdtpr(a) then
            if islambda(func(a)) then ....
            elseif isnlambda(func(a)) then ....
            elseif ismacro(func(a)) then ....
            elseif islexpr(func(a)) then ....
            else {* isfunarg(func(a)) or isapplication(func(a)) *}
```

```
                    vtemp = evalaux2(argptr,a);
                    goto skip1;
              workp = lbot;
              while not isempty(bvars(a)) do
                  push(bind(next(bvars(a)),top(workp)),bindstack);
                  workp = workp + 1;
              vtemp = last(map(eval,body(a)));
skip1:
          unbind(oldbindstack);
          if ismacro(func(a)) then
               vtemp = eval(vtemp);
          restorestack();
          return(vtemp);
     else {* isconstant(a) *}
          restorestack();
          return(a);
```

As before, processing can be divided into three phases. These are described below.

1. Initialization

The first phase differs from the old **eval** only when there exist unapplied arguments from previous applications, in which case **evalaux1** is called. As the addresses of all such arguments are kept on the popstack, the test for whether or not such arguments exist is simply a test for the bottom of the popstack (**bottom**). The function **evalaux1** handles the evaluation of all function-valued expressions.

2. Type Determination and Evaluation

The second phase of evaluation is the same as the old **eval**.

3. Function Evaluation

The third phase of evaluation is almost exactly the same as that of the old eval, the only difference being when the functional part of the expression is itself an expression (**isdtpr**).

If, in this case, the functional part is not an expression representing an abstraction (lambda, nlambda, macro or lexpr), then, instead of an error occurring, a call to **evalaux2** will be made. The function **evalaux2** handles application of funargs and function-valued applicative expressions.

§2 The Function **evalaux1**

We now need to describe the auxillary eval function **evalaux1**:

```
function evalaux1(actarg)
    partial = true
    oldbindstack = bindstack;
    savestack();
    if isatom(actarg) then
        actarg = list(actarg);
    a = func(actarg);
    if isatom(a) then
        if islambda(a) or isfunarg(a) or islexpr(a) then
            a = actarg
            actarg = list(actarg);
        elseif isnlambda(a) or ismacro(a) then
            error();
        else
            a = funcbind(a);
    elseif isvalue(a) then
        a = l(a);
    argptr = arglist(actarg);
    lbot = namestack;
    if isbcd(a) then
        newpopstack();
        if count(a) = 0 then
            partial = false;
        if isnlambda(discipline(a)) then
            if partial then
                error();
            else
```

```
              push(argptr,namestack);
        elseif ismacro(discipline(a)) then
            if partial then
                 error();
            else
                 push(actarg,namestack);
        else {* islambda(discipline(a)) or islexpr(discipline(a)) *}
            numargs = count(a);
            restorepopstack();
            pushallarguments();
            pop(popstack);
            pop(popstack);
            newpopstack();
        vtemp = call(a);
        restorepopstack();
    elseif isarray(a) then
        vtemp = array(a,argptr);
    elseif isdtpr(a) then
        if islambda(func(a)) then
            numargs = length(bvars(a));
            pusharguments();
        elseif isnlambda(func(a)) or ismacro(func(a)) then
             error();
        elseif islexpr(func(a)) then
            pusharguments();
            push(namestack-lbot,namestack);
            lbot = namestack - 1;
            pop(popstack);
            pop(popstack);
            push(1,popstack);
            push(namestack,popstack);
            numargs = 1;
        else {* isfunarg(func(a)) or isapplication(a) *}
            vtemp = evalaux2(argptr,a);
            goto skip1;
        savedpopstack = popstack;
        nargstack = 0;
        while numargs > nargstack and not bottom(popstack) do
            pop(popstack);
            nargstack = nargstack + pop(popstack);
        while savedpopstack > popstack do
            savedpopstack = savedpopstack - 1;
            top = top(savedpopstack);
            savedpopstack = savedpopstack - 1;
            workp = top - top(savedpopstack);
            while not isempty(bvars(a)) and workp < top do
                push(bind(next(bvars(a)),top(workp)),bindstack);
                workp = workp + 1;
        newpopstack();
        map(eval,all-but-last(body(a)));
        restorepopstack();
        vtemp = eval(last(body(a)));
skip1:
    unbind(oldbindstack);
    if ismacro(func(a)) then
        newpopstack();
        vtemp = eval(vtemp);
        restorepopstack();
    restorestack();
    return(vtemp);
```

Most of the recognizers, selectors, and constructors are as before, and the function of all new ones are briefly described in Appendix 4. However, before we begin the description of **evalaux1**, it is necessary to have a clear idea of how the popstack works.

As mentioned previously, the popstack has two entries per set of arguments saved on the namestack (a set of arguments corresponds to the evaluated arglist of a function). One entry gives the address of the top of the set, the second the number of arguments. This is necessary because there is extraneous information placed on the namestack between these sets and it is necessary to know where on the namestack the arguments are located.

Each argument within a set of saved arguments is pushed in a left-to-right fashion. However, the sets themselves are pushed right to left (because the operands in the outermost applications are evaluated *before* those in inner applications). Furthermore, when we come to bind variables to these values, the binding variables will be scanned in left-to-right order. Thus, to achieve the correct bindings, we begin by processing the *topmost* set of saved arguments, while initiating the binding of variables with the *bottommost* argument within that set. When all arguments in that set have been bound, we move down the stack to the next set. In effect, this achieves a left-to-right ordering of the arguments as we take them off the namestack.

Unfortunately, if partial application is allowed, the above technique cannot be used for applying bcd functions to their arguments.[1] These functions access the namestack explicitly and thus expect their arguments to be in the same order irrespective of whether or not the application is partial. In these cases, therefore, arguments previously saved on the namestack must be copied back onto the namestack in the correct order.

It is sometimes necessary to protect arguments on the namestack, since evaluating an improperly defined user function or a partially applied function could otherwise consume arguments that belong to some earlier function application. To achieve this protection, the bottom of the popstack is raised by executing **newpopstack**. When it is no longer necessary to protect the saved arguments, the bottom of the stack is reinstated to its previous position by executing **restorepopstack**.

Now we are in a position to describe **evalaux1**. Since **evalaux1** expects a function application to evaluate, it has no need for a type determination phase as in **eval**.

1. Initialization

The expresion to be evaluated (**actarg**) is passed to **evalaux1**. The current level of the bindstack is saved so that the bindstack can be reinstated when **evalaux1** exits.

2. Function Evaluation

2.1. Initialization

The next phase of evaluation is similar to the third phase in **eval**, except that extra mechanisms have to be added to extract the arguments from the namestack.

[1]Functions of lexpr discipline would present the same difficulty. However, as mentioned in Section 3.4, we do not allow lexprs to be partially applied.

Although **evalaux1** expects a function application, some or all of the arguments of the application will be on the namestack, as indicated by the popstack. When all the arguments are on the namestack, the expression passed to **evalaux1** will not be an applicative expression, i.e., it will be either an atom or an abstraction (lambda, lexpr, nlambda or macro). The first thing we must do, therefore, is convert such expressions into applicative form: this is done by forming a list of the atom or abstraction and reassigning **actarg** to this new expression. Functions of type nlambda or macro are invalid and cause an error.

At the same time, the temporary variable **a** is assigned to the operator (functional part) of the expression. Finally, the variable **argptr** (argument pointer) is made to point to the actual operands to which the function is to be applied, and the current top of the namestack is saved in the global variable **lbot**.

2.2. The Real Work

As in **eval**, the processing is now divided according to whether the functional part of the expression is a binary-coded-decimal function (**isbcd**), an array (**isarray**), or another expression (**isdtpr**).

Case 1. Binary-coded-decimal

This phase would be identical to the old **eval** if partial application were *not* allowed. However, because partial application of bcd functions leaves the arguments on the namestack in the wrong order, it is necessary to copy them back onto the namestack in the correct order (i.e., the one expected without partial application).

First, the bottom of the popstack is saved with **newpopstack**. Then it is determined whether or not the function can be partially applied. Such partial application is possible if the count field of the function (**count(a)**) is not zero.[2]

If the function's discipline is nlambda (**isnlambda**) or macro (**ismacro**) and it is partially applicable, then an error occurs (as only lambdas can be partially applied); otherwise its arguments are pushed onto the namestack, as in the old **eval**.

All else is assumed to be of lambda discipline.[3] First, the number of expected arguments is extracted through the **count** field. The bottom of the popstack is restored with **restorepopstack**. All the required arguments are then copied back onto the namestack in their correct order, using **pushallarguments**. The bottom of the popstack is then raised by

[2]The count field dictates the number of arguments required by the function, and is set to zero if the function cannot be partially applied.

[3]Lexprs are also evaluated here; as their count fields are zero, they are handled properly.

using **newpopstack**.

Finally, control is transfered to the bcd function by calling the special function (**call**). The result is placed in the variable **vtemp** and the bottom of the popstack then restored with **restorepopstack**. It is assumed that compiled code cannot represent a function-valued function.

Case 2. Array

This is the same as for **eval**.

Case 3. Function-Valued Expression (dtpr)

As in **eval**, it is necessary to distinguish whether the functional expression represents an abstraction, a funarg or an applicative expression. If it is either of the latter two, then, just as in **eval**, the auxillary function **evalaux2** is called. If, on the other hand, the expression represents an abstraction, processing will depend on the discipline.

If the abstraction's discipline is lambda (**islambda**), the number of required arguments is extracted by counting the binding variables. All the operands of the current application are evaluated and pushed onto the namestack with **pusharguments**. Note that there is no need to copy arguments from enclosing applications back onto the namestack as we did for bcd functions.

If the function's discipline is nlambda (**isnlambda**) or a macro (**ismacro**), an error occurs.

If the function's discipline is lexpr (**islexpr**), then, as in **eval**, the number of arguments (the current namestack level minus **lbot**) is pushed onto the namestack and the **lbot** pointer made to point one position below the current namestack position. It is important to note that the lexpr wants only one argument; consequently the popstack entries for all the arguments are popped and new entries are set to indicate a single argument. The variable **numargs** is then reassigned a value of one.

Now comes the task of binding the binding variables (formal parameters) to the arguments. We first make the temporary variable **savedpopstack** point to the top of the popstack. The number of saved arguments is counted in the original set by looking at the first two entries on the popstack. If there are not enough arguments the next saved set is examined. This continues until at least the correct number of arguments, or the bottom of the popstack, is reached.[4] The obtained number is now in **nargstack**.

The main loop starts with the first set of saved arguments, does the binding (at the same time pushing old bindings onto the bindstack), and then moves down the namestack to repeat the procedure on the next set of saved arguments. This continues until all the binding variables are bound to their arguments.

Note that, as in the foregoing cases, partial application considerably complicates the binding process. If partial application were not allowed, the number of arguments in each block of arguments saved on the namestack would match the number of binding variables exactly and there would be no need to loop through previous blocks of saved arguments.

Finally, **eval** is mapped over all expressions appearing in the body of the function and assigns **vtemp** to the result of the last evaluated expression. So that only the last expression will be allowed to use arguments from previous (enclosing) applications, the elements on the popstack are protected (using **newpopstack** and **restorepopstack**) from access by all expressions except the last.

2.3. Finishing Off

The old bindings are reinstated by using the saved bindstack pointer. If the function was a macro, its result is re-evaluated and assigned to **vtemp**.

Finally **vtemp** is returned as the result of **evalaux1**.

§3 The Function evalaux2

We will now consider the function **evalaux2**.

```
function evalaux2(argptr,a)
    savestack();
    pusharguments();
    if isfunarg(func(a)) then
        unwind(env(a));
        argptr = fargs(a);
        pusharguments1();
        vtemp = eval(fbody(a));
        wind();
```

[4]As we give the function a set of saved arguments (arglist) at a time, it is possible to have too many arguments. This, of course, is treated as an error.

```
else
    vtemp = eval(a);
while not bottom(popstack)
    vtemp = eval(vtemp);
restorestack();
return(vtemp);
```

The function **evalaux2** gets two arguments, the first being the operands of an expression and the second the operator (function part) of the same expression. The operands are first evaluated and pushed onto the namestack by means of **pusharguments**. Processing then depends on whether the functional part of the expression (i.e., the first argument to **evalaux2**) is a funarg or a function-valued applicative expression.

If the functional part is a funarg (**isfunarg**), then **evalaux2** will extract its environment (**env**) and unwind the bindstack to that environment. This involves reinstating bindings from the bindstack, with the old bindings being saved on the saved-bindstack. The variable **argptr** is made to point to the already evaluated arguments of the funarg. These arguments are then pushed onto the namestack by using **pushargumentsl** (which is the same as **pusharguments** except that it does not evaluate the arguments). The routine **eval** is then called with the body of the funarg as argument (which, in turn, will call **evalauxl**, thus applying the funarg to the arguments saved on the namestack). The result of the call is placed in **vtemp**. Finally the bindstack is wound back up to the calling (application) environment, using the bindings saved on the saved-bindstack.

If the function is not a funarg, it must be a function-valued application. This expression is given to **eval** (which, again, will call **evalauxl** to complete the application). The result of the call is placed in **vtemp**.

It is possible, at this stage, that the value returned is a function. If it is, there must exist unapplied arguments on the namestack to which it can be applied. Thus **evalaux2** first checks the popstack. If it indicates that there are still some unapplied arguments on the namestack, the value returned from the previous call to **eval** is assumed to be a function and **eval** is called again. This will cause the function to be applied to the unclaimed arguments remaining on the namestack. The process is continued until the popstack is empty, and the result placed in **vtemp**.

Finally, **vtemp** is returned as the result of **evalaux2**.

§4 Changes in the Function **function**

The function of **function** in FRANZ LISP is little different from **quote**'s role. It differs in that, on receiving an atom with a function binding, it returns that binding.

```
function()
   handy = first(lbot);
   if (isatom(handy) and hasfnbnd(handy)) then
      return(fnbnd(handy));
   else
      return(handy);
```

The new role of **function** is to create a funarg triple. The first component is a functional form, called the body of the funarg; the second is a list of evaluated arguments; the third is the current environment. That is, the function **function** returns funargs with the form

<FUNARG body argument-list environment>

This is achieved in the modified FRANZ LISP interpreter as follows:

```
function()
    handy = first(lbot);
    args = empty;
    if isatom(handy)
       handy = funcbind(handy);
    elseif isdtpr(handy)
       temp = func(handy);
       if isatom(temp) then
           if not isspecial(temp) then
              temp = funcbind(temp);
       args = evalaux3(arglist(handy));
       handy = temp;
    return(list('FUNARG,handy,args,bindstack));
```

The argument to **function** is first stored in the variable **handy**. The **function** is an nlambda, so its argument is in list form and unevaluated. That is why the argument needs to be extracted by taking the first element from the argument list pointed to by **lbot**.

The variable **args** is initialized by pointing to an empty list (**empty**). Processing is now divided into two, depending on whether **function**'s argument is an atom (**isatom**) or an expression (**isdtpr**).

If the argument (**bandy**) is an atom, the associated funarg <FUNARG function-binding nil env> (where env is the current environment) is returned.

If the argument is an expression, a funarg is constructed as follows. As before, the funarg's first element is "FUNARG" and the fourth element is a pointer to the bindstack indicating the current environment. The second and third elements depend on the following:

1. If the argument is an abstraction, as in **(lambda (..) ...)**, the second element is the abstraction itself and the third element is the empty list.

2. If the argument is a function application (applicative expression), the third element is the list of evaluated arguments. The second element of the funarg depends on whether the operator of the expression is an atom or not. If it is an atom, the second element is set to the function binding of the atom. If not an atom, the second element is set to the operator subexpression itself.

§5 The Function evalaux3

The function **evalaux3** simply evaluates a list of expressions, and is defined as follows:

```
function evalaux3(argptr)
    savestack();
    newpopstack();
    temp := map(eval,argptr);
    restorepopstack();
    restorestack();
    return(temp);
```

Any arguments previously saved on the namestack are protected by using **newpopstack** and **restorepopstack**.

§6 Some Possible Improvements

One possible improvement to the system is in the handling of abstractions. Consider the following expression, where **fcn** is a function of n arguments:

((fcn arg1 arg2) ... argn)

Let us now compare this with the application of a lambda abstraction:

((lambda (binding-vars) exp1) ... argn)

From a syntactic point of view, the construct "lambda" is acting in the role of a partially applied function. Indeed, the semantics are also the same if we allow the first two arguments to be quoted implicitly.

Thus a lambda construct can be looked upon as a partial application of the "function" **lambda**. The code for **lambda** can be taken out of **eval** and shallow-bound as a bcd function to the atom **lambda**. This would simplify the interpreter and improve modularity. The same implementation technique could be applied to nlambda, macro, and lexpr applications.

It is also possible to avoid the copying of arguments that is sometimes necessitated by partial application. For functions of lambda discipline, there is no need to copy arguments from previous (enclosing) applications to the top of the namestack prior to binding. The binding process knows exactly where the arguments are using information from the popstack. However, bcd functions expect their arguments in correct order, and make no use of information on the popstack. If the system were modified to compile in the access routine for the namestack when partial application is involved, then copying of the arguments would not be needed and evaluation would be made more efficient.

This also applies to foreign and system functions. However, as all foreign functions have interfacing routines with LISP, these routines can readily be changed to account for partial application. System functions can be recoded or can utilize an interfacing function.

Appendix 4: Functions and Predicates

The important functions and predicates used in the modified **eval** function are given below.

§1 Recognisers

isatom:	tests if argument is an atom
isvalue:	tests if argument is a value
isdtpr:	tests if argument is a list
isbcd:	tests if argument is a compiled function
islambda:	tests if argument is a lambda
isnlambda:	tests if argument is a nlambda
ismacro:	tests if argument is a macro
islexpr:	tests if argument is a lexpr
isarray:	tests if argument is an array
hasfnbnd:	tests if argument has a function binding
hasclb:	tests if argument has a current level binding
isspecial:	tests if argument is a special construct (e.g., "lambda")
isfunarg:	tests if argument is a funarg
isempty:	tests if argument is empty
bottom:	tests if stack is at the bottom marker (not if empty!)

§2 Selectors

clb:	returns the atom's current level binding
l:	returns the object's value
fnbnd:	returns the atom's function binding
funcbind:	as for fnbnd, but also handles clbs, etc.
arglist:	returns the expression's actual operands
bvars:	returns the abstraction's binding variables
body:	returns the abstraction's body
discipline:	returns the bcd's binding type
array:	returns the array's required element
last:	returns the list's last element
func:	returns the expression's operator
next:	returns the list's next element, given started at the begining
top:	returns the top of the stack indicated by the argument
fbody:	returns the body of a funarg
fargs:	returns the arglist of a funarg
env:	returns the environment of a funarg
all-but-last:	returns the argument, except for the last element

§3 Miscellaneous

push:	pushes the first argument onto the stack named as the second argument
pop:	pops the top of the stack named as argument and returns it
map:	applies the first argument to each element of the second argument, returning a list of results
call:	executes bcds
bind:	performs shallow binding of the first argument to the second argument and returns the old binding

unbind:	restores old bindings
savestack:	saves the value of namestack and lbot
restorestack:	restores the old values for namestack and lbot
length:	returns the length of the argument
unwind:	unwinds the bindstack to the point indicated by the argument
wind:	rewinds the bindstack to the normal position
empty:	a constant representing an empty list
error:	indicates an error

§4 Macros

The macros used are as follows:

macro newpopstack()

This macro raises the bottom of the popstack to the present position

macro restorepopstack()

This macro restores the bottom of the popstack

```
macro pusharguments()
    lbot = namestack;
    newpopstack();
    map(pushnamestack,map(eval,argptr));
    restorepopstack();
    push(namestack-lbot,popstack);
    push(namestack,popstack);
```

The above macro first saves the bottom of the popstack. It then evaluates the operands. The results of the evaluations are pushed onto the namestack, the bottom of the popstack is reinstated and finally the number and the address of the evaluated arguments are both pushed onto the popstack. This is so that future calls to **eval** will know where they are.

```
macro pusharguments1()
    lbot = namestack;
    map(pushnamestack,argptr);
    push(namestack-lbot,popstack);
    push(namestack,popstack);
```

This is the same as **pusharguments** except it does not evaluate the operands and, because it does not call **eval**, does not save the bottom of the popstack.

```
macro transferargs()
    lbot = pop(popstack);
    for i = 1 to pop(popstack)
        do
            push(pop(lbot),namestack);
```

This macro copies a set of arguments to the top of the namestack.

```
macro pushallarguments()
    pusharguments();
    pop(popstack);
    nargstack = pop(popstack);
    if partial then
        while numargs > nargstack and not bottom(popstack) do
            transferargs();
            nargstack = nargstack + top(popstack));
    push(nargstack,popstack);
    push(namestack,popstack);
```

The macro **pushallarguments** copies **numargs** arguments to the top of stack in correct order (i.e., as if not partially applied). Using **pusharguments**, it will first evaluate the operands pointed to by **argptr** and push the values onto the namestack. It will then get the number of arguments just pushed using the information in the popstack. If the function is partially applicable and there are not enough arguments for the application (**numargs** $>$ **nargstack**), it will keep transferring saved arguments (using **transferargs**) until there are enough for the application.

```
funcbind(a)
    if not hasfnbnd(a) and hasclb(a) then
       a = clb(a);
       if isatom(a) then
           a = fnbnd(a);
    else
       a = fnbnd(a);
    return(a);
```

The function **funcbind** is simply a function that extracts the function binding of an atom.

References

[1] Allen, J.R. (1978) *Anatomy of LISP,* McGraw-Hill, New York.

[2] Baker, H.G. (1978) "Shallow Binding in LISP 1.5", *Comm. A.C.M.*, 21, pp 565 - 569.

[3] Bobrow, D.G. and Wegbreit, B. (1973) "A Model and Stack Implementation of Multiple Environments", *Comm. A.C.M.*, 16, pp 591 - 603.

[4] Burstall, R.M., Collins, J.S., and Popplestone, R.J. (1971) *Programming in POP-2,* Edinburgh University Press, Edinburgh.

[5] Foderaro, J.K. (1980) *The FRANZ LISP Manual, Opus 36b,* University of California.

[6] Georgeff, M.P. (1982) "A Scheme for Implementing Functional Values on a Stack Machine", *Proceedings of the ACM Symposiom on LISP and Functional Programming,* pp 188 - 195, Pittsburgh, Pa.

[7] Georgeff, M.P. (1984) "Transformation and Reduction Strategies for Typed Lambda Expressions", *Trans. of Prog. Languages and Systems.*

[8] Greenblatt, R. (1974) "The LISP Machine", AI Working Paper 79, AI Lab., Massachusetts Institute of Technology, Cambridge, Mass.

[9] Henderson, P. (1980) *Functional Programming,* Prentice Hall, Englewood Cliffs, N.J.

[10] Kernighan, B.W., Ritchie, D.M. (1978) *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J.

[11] Landin, P. (1964) "The Mechanical Evaluation of Expressions", *Computer Journal,* Vol. 6, pp 308-320.

[12] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. (1965) *LISP 1.5 Programmer's Manual,* M.I.T. Press, Cambridge, Mass.

[13] McDermott, D. (1980) "An Efficient Environment Allocation Scheme in an Interpreter for a Lexically Scoped LISP", Proceedings of the 1980 LISP Conference.

[13] Touretsky, (1979) "A Summary of MAC-LISP", Project MAC, Massachusetts Institute of Technology, Cambridge, Mass.

[14] Steele, G. L. Jr. and Sussman, G.J. (1978) "A Revised Report on SCHEME", AI Memo 452, AI Lab., Massachusetts Institute of Technology, Cambridge, Mass.

[15] Steele, G.L. Jr. (1984) *Common LISP: The Language,* Digital Press, Burlington, Mass.