Environments as First Class Objects

David Gelernter
Yale University

Suresh Jagannathan
MIT

Thomas London
AT&T Bell Laboratories

We describe a programming language called Symmetric Lisp that treats environments as first-class objects. Symmetric Lisp allows programmers to write expressions that evaluate to environments, and to create and denote variables and constants of type environment as well. One consequence is that the roles filled in other languages by a variety of limited, special purpose environment forms like records, structures, closures, modules, classes and abstract data types are filled instead by a single versatile and powerful structure. In addition to being its fundamental structuring tool, environments also serve as the basic functional object in the language. Because the elements of an environment are evaluated in parallel, Symmetric Lisp is a parallel programming language; because they may be assembled dyamically as well as statically, Symmetric Lisp accomodates an unusually flexible and simple (parallel) interpreter as well as other history-sensitive applications requiring dynamic environments. We show that first-class environments bring about fundamental changes in a language's structure: conventional distinctions between declarations and expressions, data structures and program structures, passive modules and active processes disappear. We argue that the resulting language is clean, simple and powerful.

1 Introduction

All programming languages provide users with some way to create *environments*, where an environment is a dictionary that associates names with their definitions or values. Modern Algol-based languages supply a global and any number of local naming environments; they generally provide record-type objects as well, where a record is another kind of local namespace. Languages that support functional objects usually provide closures, which are naming environments within which a function body may be evaluated. Interpreted languages support environments that users create incrementally.

But few languages allow their environments to be first-class objects. The lack of first-class environments generally forces the inclusion of several different namespace mechanisms, each tailored to a specific sub-domain: modules for separate compilation, abstract data types for information hiding, records for grouping, closures for encapsulating an execution environment, system interfaces to handle persistent data through a file system. Languages that lack first-class environments tend to treat these structures as a set of orthogonal tools, but they are not. They are merely elaborately-disguised variations on a single theme. We will argue that, if we make environments first-class objects and consistently treat them as such, we can throw out these weak variations and substitute the powerful, flexible theme itself. In doing so, we bring about a fundamental change in the structure of our language. Conventional distinctions between declarations and expressions, data structures and program structures, passive modules and active processes disappear.

In this paper we describe a programming language called Symmetric Lisp that treats environments as first-class objects. Symmetric Lisp allows programmers to write expressions that evaluate to environments, and to create and denote variables and constants of type environment as well. One consequence is that higher-order, lexically scoped functions are supported in the presence of a dynamic binding discipline without the introduction of a separate closure type. Environments may be created statically or incrementally; an incrementally-growing environment plays the role of a *stream*[2, 27], but does considerably more besides. Because such environments are time-ordered and have elements that are accessible by name, they are an ideal representation for multiple-version data bases, persistent data structures and the environment that is incrementally constructed by the Symmetric Lisp Interpreter. Environments also form the logical basis of a concurrent execution semantics for Symmetric Lisp. Because an environment's el-

This material is funded in part by the National Science Foundation grant number #DCR-8601920.

This paper was published in the *Proceedings of the 14^{th} Principles of Programming Languages Conference*, January, 1987.

ements are evaluated in parallel, statically-specified environments serve as "cobegin"-type statements; the incremental extension of existing environments plays the role of dynamic process forking. As a parallel programming language, Symmetric Lisp lends itself to implementation on a variety of abstract architectures ranging from conventional shared-memory multiprocessors to Linda machines to graph reduction interpreters.

We explain the *symmetric* in the language's name in Section 6. Symmetric Lisp is one member of the general class of symmetric languages [9, 11]. We suspect that virtually all languages have symmetric variants.

The next section gives a general introduction to the language. Section 3 describes the role of environments as structuring tools. Section 4 discusses the semantics of environments as evaluatable objects; it defines function evaluation as a simple extension of the evaluatation rules for ordinary structures. Section 5 describes the semantics of incrementally-constructed environments. We discuss some of the proposed intermediate target languages and architectures for Symmetric Lisp in Section 7 and present comparisons of Symmetric Lisp with related work along with a summary of our results in Section 8.

2 The Alpha Form

Symmetric Lisp follows conventional Lisp syntax and supports most of the basic arithmetic, logical and control functions provided in Common Lisp.

Environments are constructed of three elements, `ALPHA`, `PRIVATE` and `ALPHA` forms. A `NAME` form binds a name (which must be a symbol) to the result of an expression, and makes this binding visible both within and outside of the environment in which it executes. `PRIVATE` is the same as `NAME`, except that it defines a binding that is invisible outside of the local environment or any nested environments. The `ALPHA` form gives meaning to these bindings by tying them together into an environment. The result of evaluating an `ALPHA` is another `ALPHA` in which every expression in the original has been fully evaluated. Within a given alpha all names, whether bound using `NAME` or `PRIVATE` forms, must be unique.

An `ALPHA` form is evaluated in three steps. In the first step, all names are evaluated simultaneously and recorded as elements of the environment to be defined by this `ALPHA`. In the second, all expressions are simultaneously evaluated; if an expression requires the value of some binding and that value is still being computed, evaluation of the expression *blocks* until the required value becomes available. In the last step, a result `ALPHA` is returned whose k^{th} element is the result yielded by the k^{th} element of the original alpha. Just as an `ALPHA` yields an `ALPHA`, a `NAME` or `PRIVATE` form yields a corresponding `NAME` or `PRIVATE` form. Thus, evaluating

```
(  ALPHA
      (  NAME x ( + 1 1 ))
      (  NAME y ( * x 10 )))
```

yields

```
(  ALPHA
      (  NAME x  2 )
      (  NAME y  20))
```

An expression may be a constant[1] , which evaluates to itself; a name, which yields the value to which it is bound; or an application of a function to arguments which are themselves expressions. Expressions may refer to names that are not defined within their immediately-surrounding alpha. When they do, we search for a binding within the innermost `ALPHA` that encloses the immediately-surrounding one in the evaluation environment; if we still find no binding, we search the next-innermost alpha and so on. This is similar to a *dynamic* binding discipline. But the existence of first-class environments changes the personality of dynamic binding, and makes it possible to support higher-order, lexically-scoped functions without the introduction of any extra linguistic primitives. We defer this topic until Section 4.

A binding may be altered by a `SETQ` command which operates in the same was as its Lisp counterpart, *except* that the name being reassigned must have previously been defined in a `NAME` or `PRIVATE` form. Note that, as in other Lisps, `PROGN` evaluates a list of forms sequentially and returns the value of the last.

Environments, being first-class objects, may be bound to names. If an environment is bound to a name Q – (`NAME` Q (`ALPHA` ...)) – we can evaluate an expression E within Q's namespace by writing (`WITH` Q E). To evaluate this expression we evaluate E, consulting Q first for the values of any free names we encounter. (Free names not bound within Q are looked-up within the immediately-enclosing alpha and so on, as per the normal evaluation rules.) The value yielded by E is returned as the value of the `WITH` form.

Each element of an `ALPHA` may be *any* expression: it is not the case that only `NAME` or `PRIVATE` forms

[1] Lambda's are also constants in Symmetric Lisp – we defer their discussion until Section 4

are acceptable as ALPHA elements. In evaluating an arbitrary expression, we follow the same rules that apply to the evaluation of expressions that appear within bindings. Thus

```
(  ALPHA (  NAME x 10) (+ x 5))
```

yields

```
(  ALPHA (  NAME x 10) 15)
```

The two main selector functions provided for ALPHA's are ACAR and ACDR; ACAR selects the first element of an ALPHA and ACDR returns an ALPHA identical to its argument without its first element. We discuss the role of ACAR and ACDR further in Section 5. The general selector function, ALPHA-nth, given an ALPHA and an integer n, returns the n^{th} element in the ALPHA. We can use ACAR and ACDR to combine and compose new environments from existing ones.

3 Environments as Structuring Tools

3.1 Record Structures

It is easy to see that the environments described thus far can play the role of Pascal-style records: they allow a collection of heterogenous data fields to be encapsulated within a single structure. Thus, evaluating

```
(  NAME leonardo
    (  ALPHA
            (  NAME job  'printer)
            (  NAME city 'florence)
            (  NAME century 16)))
```

creates a record; evaluating (WITH leonardo city) yields the symbol florence. The side-effecting form (WITH! Q E) allows us to alter E. Evaluating

```
  (with! leonardo (setq job
'literary-critic))
```

alters leonardo's "job" field. (All side-effecting operations in Symmetric Lisp are evaluated atomically. If two expressions simultaneously attempt to evaluate WITH! over the same alpha, one proceeds, the other blocks until it is finished and then proceeds; the ordering is non-determined.)

An ALPHA-form is significantly more powerful than a simple record type generator, though; because names can be bound to functions, we can use the ALPHA form to build modules and libraries, too. For example, the expression

```
(  NAME math-library (  ALPHA
        (  NAME pi 3.1415265)
        (  NAME sin  LAMBDA (x)
                < definition of sin >)
        (  NAME cos  LAMBDA (x)
                < definition of cos >)))
```

defines a math library. To compute sin of n, we write

```
(WITH math-library ( sin n ))
```

3.2 Abstract Data Types

The PRIVATE form is useful when we wish to avoid cluttering a public namespace or to conceal information about a structure's internals. One obvious application is in the definition of abstract data types that export operations but hide data representation. The following simple example uses PRIVATE, and demonstrates another important consequence of first-class environments as well: we can define a function that returns an environment as its value; this makes it particularly easy to build parameterized data types. Thus

```
(  NAME queue (  LAMBDA (t)
     ( ALPHA
      (  NAME append ( LAMBDA (item)
           <... code for append ...>))
      (  NAME delete ( LAMBDA
           <... code for delete ...>))
      (  PRIVATE rep ( make-array :type t
)))))
```

The expression (NAME q1 (queue integer)) binds q1 to an environment that represents a queue of integers. If we write (WITH q1 (append 3)), the element 3 is appended to q1. (We can in fact, Smalltalk-style[14], regard integer itself as an environment with an internal representation and suitable syntactic sugaring provided by the system.)

4 Environments as Functional Objects

Lambda forms are constants in Symmetric Lisp; they evaluate to themselves. This distinguishes Symmetric Lisp both from Classical Lisps and from Scheme-based (or functional) languages. In Classical Lisps, lambdas are "half constants" – they're referred to as constants, but unlike the constant "3", they can't be evaluated. So Classical Lisps provide no notation for expressions that yield functions, and none for full function-valued constants. Lambdas in Scheme are expressions that yield function-valued (or more precisely closure-valued) constants. But the constants they yield are non-denotable. In Scheme, then, integer-valued expressions (say "(+ 1 2)") yield denotable values, but function-valued expressions (say "(lambda () (+ 1 2))") yield opaque closures.

In Symmetric Lisp, lambda forms may be evaluated, and they yield values that are denotable.

Applying a lambda in Symmetric Lisp yields the result of evaluating

```
( ALAST ( ALPHA
      ( PARAMNAME formal-1 actual-1)
      ( PARAMNAME formal-2 actual-2)
               ...
      ( PARAMNAME formal-n actual-n)
      lambda-body))
```

`ALAST` is a selector that returns the last element of an alpha. `PARAMNAME` behaves exectly like name, with one exception: the process of finding definitions for the free names in a `PARAMNAME` expression begins with the first immediately-enclosing alpha, not (as would normally be the case) with the current one. `PARAMNAME` forms are essential to avoid having free names within actual expressions refer, in case of name conflict, to formal parameters of the function. Otherwise, `PARAMNAME` is just like name: specifically, the names bound by `PARAMNAME` forms are visible within the alpha to any non-`PARAMNAME` expression. Thus, names within `lambda-body` are resolved by referring to the `PARAMNAME` forms within this alpha. A `PARAMNAME` form returns an ordinary name form as its value.

Consider, for example, the definition

```
( NAME f ( LAMBDA (x y) (+ x y)))
```

The application (f 2 3) calls for the evaluation of

```
( ALAST ( ALPHA
       ( PARAMNAME x 2)
       ( PARAMNAME y 3)
       (+ x y)))
```

After evaluating the alpha we have

```
( ALAST ( ALPHA
       ( NAME x 2)
       ( NAME y 3)
       5))
```

Evaluating the `ALAST` yields 5.

Symmetric Lisp also supplies a `PLAMBDA` form. Its behaviour is a simpler variant of `LAMBDA`'s. Like `LAMBDA` forms, `PLAMBDA` forms are constants; their evalution rule is the same, except that we omit the `ALAST` in the application rule. If we had written

```
( NAME f ( PLAMBDA (x y) (+ x y)))
```

then the application (f 2 3) would have called for the evaluation of

```
( ALPHA
       ( PARAMNAME x 2)
       ( PARAMNAME y 3)
       (+ x y))
```

and the result would have been an entire alpha, namely

```
( ALPHA
       ( NAME x 2)
       ( NAME y 3)
       5)
```

The forms that make up the body of `LAMBDA`'s and `PLAMBDA`'s constitute an implicit alpha, to be evaluated within the context of the application `ALPHA`. Thus, given

```
(NAME g ( LAMBDA (x)
   ( NAME a-local-variable init)
   ( PROGN ... ))
```

The application `(g 10)` calls for the evaluation of

```
( ALAST ( ALPHA
       ( PARAMNAME x 10)
       ( NAME a-local-variable init)
       ( PROGN ... )))
```

which yields the result returned by the `PROGN`.

4.1 General Structure Specifications

In its most simplest use, `PLAMBDA` serves as a Pascal-style record template generator. Consider the following plambda (its body is null):

```
 ( NAME anyman ( PLAMBDA job wife
popularity))
```

The `NAME` form

```
( NAME philip
     (anyman (``prince-consort''
              ``liz''
              ``middling''))
```

yields

```
( NAME philip ( ALPHA
       (job ``prince-consort'')
       (wife ``liz'')
```

```
          (popularity ‘‘middling’’)))
```

Evaluating (`WITH` philip wife) yields the string “`liz`”.

The `PLAMBDA` form is significantly more powerful than a simple record template generator, though; because the body of the `PLAMBDA` can contain arbitrary expressions, we can use it to define structures whose fields are functions that depend on the `PLAMBDA`'s arguments.

For example, consider the following, Simula-style[6], object-oriented implementation of a cons cell.

```
( NAME cons ( PLAMBDA car cdr)
        ( NAME replaca
            ( LAMBDA (new-car)
                    ( SETQ car new-car)))
        ( NAME replacd
            ( LAMBDA (new-cdr)
                    (SETQ cdr new-cdr))))
```

The definition

```
( NAME some-list (cons X Y))
```

yields

```
( NAME some-list ( ALPHA
        ( NAME car ‘‘X’’)
        ( NAME cdr ‘‘Y’’)
        ( NAME replaca
                ( LAMBDA ... ))
        ( NAME replacd
                ( LAMBDA ... ))))
```

Evaluating (`WITH` *some-list car*) yield “X”.

Compare the above solution to a comparable solution in Scheme using closures. `PLAMBDA` returns a transparent environment: we can access by name any field in the environment. A closure is also an environment, but an *opaque* one. The named fields within it are *inaccessible* outside the closure. If a closure defines four named fields (as "cons" does above), we can only access the correct one based on our knowledge of the closure's internal structure.

Here is a closure-based implementation of an object-style "cons", from Steele and Sussman[23] (with variable names modified for clarity, and consistency with the example above):

```
( DEFINE (cons car-val cdr-val)
        ( LAMBDA (selector)
            (selector
              car-val
              cdr-val
              ( LAMBDA (new-car)
                      (SETQ car-val
new-car))
              ( LAMBDA (new-cdr)
                      (SETQ cdr-val
new-cdr)))))
```

We can't access "`car-val`" directly; instead we must supply a suitable selector function that picks out the appropriate value based on its knowledge of the form's internals:

```
( DEFINE (car cons-object)
       (cons-object
          ( LAMBDA (first second third
fourth)
                 first)))
```

Similarly for the function-valued fields:

```
  ( DEFINE (replaca cons-obj new-car)
          (cons-obj
            ( LAMBDA (first second third
fourth)
                   ( PROGN
                           (third new-car)
                           cons-obj))))
```

The complete implementation of `cons`, then, consists of (1) the function `cons`, which returns a closure in which four separate, anonymous fields are embedded; (2) four selector functions, two of which (the ones for `car` and `cdr`) are designed to pick out and return the value of one of the four anonymous fields, the other two of which (for `replaca` and `replacd`) are designed to pick out and then apply the value of a function-valued anonoymous field. It's clear, then, that closures are unsuitable as a foundation for a class- and object-based approach to modularity. Even though they can be used to implement classes and objects, one needs to violate the very abstraction facility that closures are supposed to provide to do so.

Thus, consider a language like Common Lisp, a lexically-scoped language, that provides closures. Common Lisp needs provides the `DEFSTRUCT` form layered on top of closures to accomodate records; packages and modules are added to accomodate library units; and, in CommonLoops[4] (an extension of Common Lisp), classes to support object-oriented programming.

Relative to more powerful and general environments, closures exactly one point in their favor. A closure is an applicable function; in cases where we require an environment that contains exactly one function definition, closures are a convenience. Consider a random-number generator. Using closures, we can write

```
  ( DEFINE (rgen seed)
```

```
          ( LAMBDA ()
                ... compute the next
pseudo-random number,
                and update the variable
"seed" ... ))
```

If `f` is the value yielded by (`rgen 15`), the simple expression `(f)` will return the next random number. An equivalent definition in Symmetric Lisp:

```
( NAME rgen ( PLAMBDA seed)
       ( NAME next ( LAMBDA () ... )))
```

The definition (`NAME` f (rgen 15)) yields

```
( NAME f ( ALPHA
       ( NAME seed 15)
       ( NAME next ( LAMBDA () ... ))))
```

To compute the next random number, we must write (`WITH f (next)`). That is, we must name and explicitly reference a field that, in the closure-based solution, was allowed to remain gracefully anonymous.

Even though environments are more expresive than closures, the convenience that closures provide, as seen above, in certain cases, is important enough that Symmetric Lisp provides a facility for getting the same effect. We make it possible to *wrap* an environment around a `LAMBDA` before applying it. The `LAMBDA` will be evaluated as always, except that the evaluation will be carried out in the context of the environment in which it is wrapped. Very simply, a wrapped `LAMBDA` is an `ALPHA` whose last element is an `ALPHA`:

(ALPHA *elt-1 elt-2* ... *elt-n*
(LAMBDA (*args*) *body*))

We can apply such an `ALPHA` just as we apply `LAMBDA`'s with one difference: instead of building a new `ALPHA` which contains the binding of the formals to the actuals of the application and evaluating the body of the `LAMBDA` in this environment, we evaluate the `PARAMNAME` bindings and body within the `ALPHA` being applied. This (as was the case with normal `LAMBDA` application) prevents free names in the actuals from conflicting with names defined in the `ALPHA` enclosing the `LAMBDA`. The result yielded is the result of applying the `LAMBDA` within the context of the alpha. `PLAMBDA`'s (conveniently enough) are a natural way to generate wrapped `LAMBDAS`. Thus consider

```
( NAME rgen2 ( PLAMBDA seed)
       ( LAMBDA () ... ))
```

The definition (`NAME` g (rgen 15)) yields the wrapped lambda

```
( NAME g ( ALPHA
       ( NAME seed 15)
       ( LAMBDA () ... )))
```

Evaluating (g) causes the evaluation of the lambda within the context of the encompassing alpha: thus, the `seed` that appears within the lambda body refers to the `seed` inside the environment named g.

The resulting semantics strongly resemble the semantics of closures: a closure, too, is a function wrapped in an environment. The crucial difference is that the environment, in Symmetric Lisp's case, is simply an alpha like any other – no separate mechanism is needed to handle environments that are used in function application than is used to handle environments used as data structures.

4.2 Partially Closed Functions

The treatment of `LAMBDA`'s as constants and the use of `PLAMBDA`'s in capturing the define-time values of free variables in a function allows the Symmetric Lisp programmer to selectively choose which variables need to be statically bound and which ones need to be bound dynamically within a function. As an example of when such a facility is useful, consider the problem presented in Steele and Sussman [22]: We wish to write a function, `generate-sqrt-of-given-extra-tolerance`, which is take one argument, which is the factor by which the tolerance is to be increased, and return a function which takes square roots with that much more tolerance than usual, whatever "usual" is defined later to be. The function makes a free reference to a variable `epsilon`, the tolerance it demands of the trial solution. The difficulty of this problem is that both `epsilon` (which is to be dynamically scoped), and `factor` (which is to statically scoped) are used in the same function.

In Symmetric Lisp, we would write this function as

```
( NAME
generate-sqrt-of-given-extra-tolerance
      ( PLAMBDA (factor)
         ( PLAMBDA (x)
             ( LAMBDA (epsilon) (sqrt x))
                (* epsilon factor))))
```

The function when called, rebinds `epsilon` to `factor` times its current value and then calls `sqrt`. When called with a factor of 10 and an integer 100, we get:

```
( ALPHA
      ( NAME factor 10)
      ( ALPHA
```

```
              ( NAME x 100)
              ( LAMBDA (epsilon) ...)))
```

Scheme solves this problem by introducing *fluid* variables whose values are retrieved from the dynamic environment that is passed as an implicit parameter to every function. ZetaLisp [15] requires that the programmer specify what variables are to be closed in the lexical environment, treating all non-closed variables as being dynamically scoped. The Symmetric Lisp solution differs from Scheme insofar as no new data structures *e.g.*, fluid environments need to be introduced to retrieve the proper value of `epsilon`. It's simpler than the ZetaLisp solution in that there is no extra linguistic mechanism *e.g.*, closures, that has to be incorporated to capture the define-time environment of `factor`.

4.3 Curried Functions

As seen above, our treatment of `LAMBDA`'s allows Symmetric Lisp to support higher-order functions. We now consider what happens when a `LAMBDA` or `PLAMBDA` is invoked in a curried fashion *i.e.* is invoked with fewer than the specified number of parameters. Such an invocation returns a wrapped `LAMBDA`, as follows. If the function

```
( LAMBDA ( f1 f2 ...  fn)  lambda-body)
```

is invoked on actual parameters $a_1 \ldots a_k$, where $k < n$, the result is the value yielded by the expression

```
( ALPHA
       ( PARAMNAME formal-1 actual-1)
       ( PARAMNAME formal-2 actual-2)
            ...
       ( PARAMNAME formal-k actual-k)
       ( LAMBDA (f_{k+1} ... f_n)
              lambda-body))
```

which in turn yields

```
( ALPHA
       ( NAME formal-1 actual-value-1)
       ( NAME formal-2 actual-value-2)
         ...
       ( NAME formal-k actual-value-k)
       ( LAMBDA (f_{k+1} ... f_n)
               lambda-body))
```

As a simple example consider

```
( NAME f ( LAMBDA (x y) (+ x y))
```

The definition (`NAME add3 (f 3)`) yields

```
( NAME add3 ( ALPHA
              ( NAME x 3)
              ( LAMBDA (y) (+ x y))))
```

Applying `add3` to 4 yields (by the rules of application for wrapped `LAMBDA`'s)

```
( ALPHA
       ( NAME x 3)
       ( PARAMNAME y 4)
       (+ x y))
```

which evaluates to 7.

5 Environments With Temporal Behaviour

The enviroments we have described thus far are static. The collection of bindings they define cannot be augmented. Sometimes, however, environments can't be defined all at once; they must be constructed incrementally. Consider an environment that represents a file system: each directory is a sub-environment and bindings associate file names with file objects. An environment that represents a general multi-version data base must also be constructed incrementally, and so too must be the environment assembled by the interpreter during an interactive session.

Symmetric Lisp accomodates growing environments with the *open alpha* form. An open alpha is an *ongoing computation*, not a passive object. An open alpha is under evaluation from the time at which it is created, and continues under evaluation indefinitely. Using the operation `ATTACH!`, described below, we can drop new elements one-by-one into an open alpha; evaluation of a new element begins as soon as it is attached.

The operation `OPEN-alpha` returns a new open alpha; its value at creation is the empty alpha, an alpha with no elements. Thus

```
( NAME environ1 (open-alpha))
```

yields

```
( NAME environ1 ( ALPHA * ))
```

The symbol "*" means "this alpha is open, and new elements will appear *here*.

The side-effecting function `ATTACH!(an-open-alpha, expr)` attaches `expr` to the ongoing computation represented by `an-open-alpha`, as described below. As soon as it has been attached, `expr` is evaluated in the context of the growing environment of which it is now part.

(ATTACH!an-open-alpha expr) causes the "*" within an-open-alpha to be replaced by (ALPHA expr *). Thus, if we start with our newly-created environ1 and evaluate (ATTACH! environ1 (NAME foobar 1)), ENVIRON1 becomes

```
( ALPHA
    ( ALPHA ( NAME foobar 1) * ))
```

If we then evaluate (ATTACH! environ1 (NAME bazball 2)), environ1 becomes

```
( ALPHA
    ( ALPHA
          ( NAME foobar 1)
          ( ALPHA
                ( NAME bazball 2) * )))
```

After evaluating (ATTACH! environ1 (+ foobar bazball)), environ1 is

```
 ( ALPHA
    ( ALPHA
          ( NAME foobar 1)
          ( ALPHA
                ( NAME bazball 2)
                ( ALPHA 3 * ))))
```

Open alphas evaluate in exactly the same way as closed alphas. All that we need to add is a rule for the evaluation of "*". The rule is as follows: to evaluate the element "*", block until the identity of the name to be defined by this alpha's first element (if there is such a name) has been determined; then block again until "*" is replaced by a new element E, and then evaluate E, using the ordinary alpha scoping rules to resolve the meaning of names within E.

Note that we can always fully evaluate the *first* element of an OPEN-alpha without knowing what form will ultimately replace the asterisk. This is so because "*" must necessarily be replaced by something of the form (ALPHA expr *): that is, the asterisk will always be replaced by an alpha, never by a NAME or a PRIVATE form; this being so, the something that replaces "*" can never alter the naming environment within which the first element evaluates. It follows that, to resolve the meaning of names encountered while evaluating expressions within an open-alpha, the rule is simple: search left; ignore what lies to the right. If a name is multiply-defined within an open alpha, the newest (rightmost) definition supercedes earlier ones – this follows, again, from the normal alpha scoping rules.

If some-env is an open alpha, we can evaluate

```
( WITH  some-env expr)
```

as we can in the case of closed alphas, but WITH works in a slightly different way: it causes expr to be evaluated as if it were substituted for the "*" element of some-env, as the "*" appears at the point where the evaluation of WITH is to begin. (Elements that are attached to some-env while the WITH is evaluating have no effect on its evaluation). Thus if the open alpha referred to above (the one that contains foobar and bazball) is named env2, the form

```
( WITH env2 foobar)
```

yields 1.

The open alpha, like the closed alpha, is a *parallel evaluation form*. If we attach to some open alpha n elements in succession, all n will evaluate simultaneously, subject only to the condition that evaluation of element k can't begin until the name to be defined by element *k-1* has been entered in the symbol table. If, for example, *curr-env* is an open alpha, we can fork three simultaneous matrix multiplications by successively evaluating

```
( ATTACH! *curr-env* (mm mat1 mat2))
( ATTACH! *curr-env* (mm mat3 mat4))
( ATTACH! *curr-env* (mm mat5 mat6))
```

If we give these evaluations names –

```
( ATTACH! *curr-env*
      ( NAME first-product (mm mat1
mat2)))
( ATTACH! *curr-env*
      ( NAME second-product (mm mat3
mat4)))
( ATTACH! *curr-env*
      ( NAME third-product (mm mat5
mat6)))
```

then we can access the value yielded by the first multiplication simply by referring to the name "first-product". If such a reference occurs before the multiplication is complete, it blocks until a value is available and then continues, as per the usual alpha evaluation rule.

It follows from the definition of open alphas that a Symmetric Lisp interpreter is simple (almost trivial) and transparent. In outline,

```
( NAME interp-symlisp
   ( LAMBDA (which-env)
        ( NAME ((next-element
        ( READ-element *input*))))
        ( COND
              (( EQ? next-element
*term*) '())
              (T ( PROGN
```

```
( ATTACH!

which-env

next-element)

(interp-symlisp

which-env))))))
```

This Symmetric Lisp interpreter is a *parallel* interpreter. Up to the seralization imposed by the symbol-table insertion rule, all expressions entered by the user are evaluated in parallel. A reference to a name whose value isn't yet available simply blocks (as always) until a value is computed, then proceeds.

The Symmetric Lisp interpter has, we believe, far-ranging implications.

1. It is a parallel interpreter.
2. It constructs environments that are transparent and denotable. Let `*curr-env*` be an interpreter-constructed open alpha; we can write

   ```
   ( WITH *curr-env* expr)
   ```

   or display the current value of `*curr-env*`, precisely as we can in the case of closed alphas.
3. The user can create and maintain any number of disjoint simultaneous interpreter sessions, or nest sessions within sessions.
4. The interpreter is well suited to environments in which persistence is desirable. For example: we can create an open alpha called "world-alpha"; we can store each new interpreter session as one element of world-alpha. We can use open-alphas to organize file systems containing old interpreter sessions, or any other kind of file, in the same way we use them within a given session. The interpreter is discussed in greater detail in [13].

5.1 Streams and Open Alphas

We can select elements of an open-alpha using the selector functions over `ALPHA`'s – `ACAR` and `ACDR` – described earlier. If `*env*` is the open alpha

```
( ALPHA
       ALPHA first-elt
           ( ALPHA  other-elts))
```

then (`ACAR` *env*) returns first-elt, *unless* first-elt is "*"; if it is, `ACAR` blocks until "*" is replaced, then proceeds as before. (`ACDR` *env*) returns (`ALPHA other-elts`), unless `other-elts` consists of "*", in which case it blocks until "*" is replaced.

We noted above that side-effecting operations in Symmetric Lisp are evaluated atomically. This goes for `ATTACH!` as well. Several processes may simultaneously `ATTACH!` to the same open alpha; the elements they specify are attached in arbitrary serial order.

We now have everything we need to use open-alpha as *multi-streams*, where a multi-stream is a stream to which arbitrarily-many processes may append. Thus

```
( NAME strm (open-alpha))
```

creates an object which we may think of as a stream;

```
( ATTACH! strm  some-element)
```

appends to the end. Processes wishing to scan the stream and do so using `ACAR` and `ACDR`.

6 Symmetry

What's symmetric about Symmetric Lisp? We claim that, empirically, the basic *structuring rules* in program-building are symmetrical with respect to an interchange between the roles of space and time in their definitions. The model we have discussed supports this basic symmetry with two symmetric structures, the `PROGN` form and the `ALPHA` form.

`PROGN` and `ALPHA` are symmetric transformations of each other in the following sense. A `PROGN`'s elements are evaluated at separate times, in the same space. The evaluations are disjoint in time insofar as one must complete before the next begins. They occupy the same space in the sense that one evaluation's variables and temporary storage locations may always be allocated on top of the last evaluation's locals and temporaries. An `ALPHA`'s elements, on the other hand, are evaluated in separate spaces at the same time. By "separate spaces" we mean that each evaluation's locals and temporaries must be stored separately. Note that the minimal time required to evaluate a `PROGN` is the sum of the evaluation times for each element; minimal space to evaluate an `ALPHA` is the sum of the is the maximum of the spaces required by each element evaluation individually. Minimal time required by an `ALPHA` is the maximum of the required times.

These points are pursued in [13].

7 Implementation

Symmetric Lisp is intended for implementation on a parallel machine. There are a variety of abstract execution models that are well-suited to the operational semantics of the language. One such model that fits well with Symmetric Lisp is parallel graph reduction [25, 19]. We can view an `ALPHA` form as a graph whose leaves are the values of name bindings defined in it. Evaluating an `ALPHA` form causes expressions in the `ALPHA` to be reduced to the value they represent. The parallel semantics of the language makes the reduction process a concurrent one. Symmetric Lisp is an excellent fit to Linda machines as well. [12] discusses the implementation on Linda of a preliminary form of Symmetric Lisp. Linda and Linda machines are described in [10][5].

Symmetric Lisp runs currently as a sequential interpreter written in Common Lisp.

8 Comparison to Other Work

Very few programming languages support environments as first-class, denotable structures and none, to our knowledge, have the semantics of `ALPHA` forms. The reflective languages of Friedman and Wand[26] and Smith[21] treat environments in the usual sense, as bindings of identifiers to values, but are used solely by reifying functions and are not denotable. There has been much interest of late in persistency in database systems. Work by Atkinson [3] on PS-Algol and Napier also treat environments as first-class, denotable objects but the focus of their interests centers around building well-typed, persistent databases and, consequently, bears little similarity to Symmetric Lisp.

Symmetric Lisp also bears superficial similarity with languages based on continuation-passing semantics[7] insofar as environments are treated as meta-linguistic data type and to languages that support closures explicitly such as ZetaLisp[15]. Insofar as the notion of an environment is integral to the implementation of higher-order functions, Symmetric Lisp also has some similarities with functional or applicative languages[17, 24] but differs significantly from all these languages in ways discussed earlier.

The `ALPHA` form in Symmetric Lisp serves some of the same roles as the package construct in Ada[1] or the module construct in Modula[28] insofar as all three constructs are used to build local namespaces for information encapsulation.

In most parallel Lisp's such as MultiLisp[16] or Qlambda[8], parallelism is retro-fitted on top of a primarily sequential framework. Concurrency is expressed using control structures such as `PCALL` or `FUTURE` in MultiLisp or `QLET` in Qlambda. Symmetric Lisp, on the other hand, has parallelism organized around data structures, namely, the `ALPHA` form resulting in intrinsic parallelism and a parallel interpreter. Finally, there may be some confusion between the notation used in this paper and that defined in Connection Machine Lisp[20]. Steele's work also involves parallelism in a Lisp context and also uses notation involving the word `ALPHA`, but we feel that despite the similar terminology, the focus of CMLisp (expressing parallelism for fine-grained parallel computers such as the Connection Machine[18]) is quite different from our main goals.

9 Conclusion

We have described a new programming language, Symmetric Lisp, whose distinguishing characteristic is the presence of environments as first class objects. We have examined some of the ramifications of this proposal – the elimination of any distinction between program and data structures, the ability to implement parallel interpreters and persistent data objects, and the unifying treatment of passive modules and active processes.

References

[1] **Reference Manual for the ADA Programming Language**, 1982.

[2] E.A. Ashcroft and W.W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[3] M. Atkinson and R. Morrison. Types, Bindings, and Parameters in a Persistent Environment. In *Persistence and Data Types Papers for the Appin Workshop*. University of St. Andrews, August 1985.

[4] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops:Merging Lisp and Object-Oriented Programming. In *Object Oriented Programming Systems, Languages and Applications*, pages 17–30, September 1986.

[5] Nick Carriero, David Gelernter, and Jerry Leichter. Distributed Data Structures in Linda. In *13^{th} ACM Symposium on Principles of Programming Languages*, Jan. 1986.

[6] O.J. Dahl, B. Myhruhaug, and K. Nygaard. The Simula67 Base Common Base Language. Technical report, Norwegien Computing Center, 1970.

[7] Daniel Friedman, Chris Haynes, and Eugene Kohlbecker. Programming With Continuations. In *Program Transformations and Programming Environments*. Springer-Verlag, 1985.

[8] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 25–44, August 1984.

[9] David Gelernter. Symmetric programming languages. Technical Report TR 353, Yale University Dept. of Computer Science, December 1984.

[10] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, pages 80–112, January 1985.

[11] David Gelernter. Symmetry and the non-evolution of programming languages. Technical Report TR 486, Yale University Dept. of Computer Science, June 1986.

[12] David Gelernter, Nick Carriero, Sarat Chandran, and Silvia Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, August 1985.

[13] David Gelernter, Suresh Jagannathan, and Thomas London. Parallelism, persistence and meta-cleanliness in the symmetric lisp interpreter. Technical report, Yale University Dept. of Computer Science, 1986.

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Press, 1983.

[15] R. Greenblatt, T. Knight, J. Holloway, D. Moon, and D. Weinreb. The LISP Machine. In *Interactive Programming Environments*, pages 326–352. McGraw-Hill, 1984.

[16] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[17] P. Henderson. *Functional Programming: Application and Implementation*. Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

[18] D. Hillis. *The Connection Machine*. MIT Press, 1985.

[19] Robert Keller, Gary Lindstrom, and Suhas Patil. A Loosely-Coupled Applicative Multi-Processing System. In *AFIPS Conference Proceedings*, volume 48, pages 613–622, June 1979.

[20] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-Grained Parallel Symbolic Computing. In *Proceedings of the 1986 Conf. on Lisp and Functional Programming*, pages 279–298, August 1986.

[21] Brian Smith and J. des Rivières. The Implementation of Procedurally Reflective Languages. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 331–347, August 1984.

[22] Guy Steele Jr. and Gerry Sussman. Lambda: The Ultimate Imperative. Technical Report AI-TM 353, MIT Artificial Intelligence Laboratory, 1976.

[23] Guy Steele Jr. and Gerry Sussman. The Art of the Interpreter, or the Modularity Complex. Technical Report AI-TM 453, MIT Artificial Intelligence Laboratory, 1978.

[24] D. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *1985 Proceedings on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, September 1985. Lecture Notes on Computer Science, Number 201.

[25] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software - Practice and Experience*, 9:31–49, 1979.

[26] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 Conf. on Lisp and Functional Programming*, pages 298–307, August 1986.

[27] K.-S. Weng. An Abstract Implementation for a Generalized Data Flow Language. Technical Report TR-228, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.

[28] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1985.