Parallelism, Persistence and Meta-Cleanliness in the Symmetric Lisp Interpreter

David Gelernter
Yale University

Suresh Jagannathan
Massachusetts Institute of Technology

Thomas London
AT&T Bell Laboratories

Abstract

Symmetric Lisp is a programming language designed around first-class environments, where an environment is a dictionary that associates names with definitions or values. In this paper we describe the logical structure of the Symmetric Lisp interpreter. In other interpreted languages, the interpreter is a virtual machine that evaluates user input on the basis of its own internal state. The Symmetric Lisp interpreter, on the other hand, is a simple finite-state machine with no internal state. Its role is to attach user input to whatever environment the user has specified; such environments are transparent objects *created by, maintained by and fully accessible to* the user. The interpreter's semantics are secondary to the *semantics of environments* in Symmetric Lisp: it is the environment-object to which an expression is attached, not the interpreter, that controls the evaluation of expressions.

This arrangement has several consequences. Because environments in Symmetric Lisp are governed by a parallel evaluation rule, the Symmetric Lisp interpreter is a *parallel* interpreter. A Symmetric Lisp environment evaluates to another environment; a session with the interpreter therefore yields a well-defined environment object as its result. Users are free to write routines that manage these interpreter-created objects – routines that list the elements of a namespace, coalesce environments, maintain multiple name definitions and so on precisely because environment objects may be freely inspected and manipulated. Because a named environment may contain other named environments as elements, interpreter-created objects may be regarded as hierarchical file systems. Because of the parallel evaluation semantics of environments, the interpreter is well-suited as an interface to a concurrent, language-based computer system that uses Symmetric Lisp as its base language. We argue that – in short – a basic semantic simplification in Symmetric Lisp promises a correspondingly basic increase in power at the user-interpreter interface.

This material is based on work supported in part by the National Science Foundation grant number #DCR-8601920

1 Introduction

In most interpreters, the evaluation of user expressions is controlled by the interpreter's internal state. The user can't see this hidden internal state, and can't easily modify or extend (still less completely rewrite) the programs to access and manipulate the environment image that are provided as part of the interpreter package. Extensibility and modularity of the interpreter-supplied programming environment are controlled by the system's implementors, not its users.

In most interpreters, the explicitly serial READ-EVAL-PRINT loop militates against the idea of a parallel interpreter. Parallel dialects of Lisp have therefore added language constructs to support parallelism without touching the interpreter. Languages like Multilisp [14] and Qlambda[8] are Lisps augmented with parallelism constructs; but their interpreters (evidently) still follow a serial evaluation rule.

In this paper we present a different interpreter-building paradigm, one in which the interpreter maintains *no* internal state information. The Symmetric Lisp interpreter's job is simply to attach user input to a user-specified environment; it is the environment structure, not the interpreter, that controls the evaluation of expressions. Environments are first-class objects – they may be freely examined, accessed and manipulated by the user. Symmetric Lisp's first-class environment serves several purposes:

1. Like conventional namespaces, they bind a collection of names to values.
2. Like parallel control structures, they allow a collection of elements to be evaluated concurrently.
3. *Un*like other control constructs, upon evaluation they yield new environments, in which all constituents of the original appear in evaluated form.
4. They may be treated as extensible structures to which new environment-fields may be attached dynamically.

The introduction of first-class environments has far-reaching ramifications for the construction of a language-based computer system. Because evaluating an environment yields another environment, the result of an interpreter session is a well-defined, transparent structure. Transparent environments make it possible for users to write customized functions that inspect, coalesce, or maintain multiple name spaces. Because the environment objects constructed by the interpreter may be regarded as persistent, hierarchical file systems, these same routines will work for file systems. Because of the underlying parallel evaluation semantics of environments, the Symmetric Lisp interpreter is inherently parallel as well.

In the next section, we discuss the design of the Symmetric Lisp programming language. Section 4 gives an outline of the Symmetric Lisp meta-circular interpreter. We then go on to discuss issues of parallelism, persistency, and general expressivity in the Symmetric Lisp programming environment. The last section describes the context of this work and related projects.

2 Environments in Symmetric Lisp

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.

Symmetric Lisp is a programming language designed around first-class environments. First-class environments make it possible to write not only constants of type environment, but expressions that yield environments as results or accept them as arguments. Languages that lack first-class environments rely ordinarily on a smorgasborg of different namespace mechanisms – records or structures for grouping data, closures for encapsulating an execution environment, classes or flavors for building objects, modules or packages for building libraries, system interfaces for building a persistent file system. We argue that the presence of first-class, denotable environments eliminates the need for these weaker mechanisms, and furthermore brings about a fundamental and interesting change in the nature of the programming language: conventional distinctions between data and control structures, modules and processes, expressions and declarations disappear.

A Symmetric Lisp environment is constructed from three elements: NAME, 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. The ALPHA form gives meaning to these bindings by tying them together. 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. First, names are evaluated simultaneously and recorded as elements of the environment to be returned by this ALPHA. This environment is now accessible to any expression that requires it. Its evaluation is only complete, however, when all expressions in the ALPHA have been evaluated. These expressions are evaluated simultaneously; 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. Evaluation of an ALPHA yields another ALPHA 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. (The fact that an ALPHA may be accessed before all of its elements have been evaluated, together with its built-in synchronization rule, makes it similar to functional data structures implemented using early-completion structures[7] or I-structures[2].)

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, 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.

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

Elements of an ALPHA may be selected through the operators ACAR, ACDR, ANTH or ALAST, which are ALPHA counterparts of the traditional Lisp list selectors. We discuss ACAR and ACDR in greater detail in the next section.

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. USE is a simpler variant of WITH: evaluating (USE A) dumps the NAME forms found within ALPHA A into the current environment. It is an error if any name conflicts arise as a result.

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

Environments can be used to implement Pascal-style records; because names can be be bound to functions, we use them to build modules and libraries, too. The PRIVATE form is useful for building abstract data types; an important consequence of first-class environments is that we can define a function that returns an environment as a value, making it easy to build parameterized data types as well.

The implementation of functions in Symmetric Lisp is intimately tied to its treatment of environments. LAMBDAs are functional constants that may be applied; application of a LAMBDA to an argument yields an environment whose last element is the body of the LAMBDA expression. The value of the application is accessed by using the ALAST selector. As described in [10], environments can be used to support curried, higher-order, lexically-scoped functions despite Symmetric Lisp's dynamic binding discipline.

3 Environments as Active Computations

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; each new element is evaluated 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 an-expr)
```

evaluates *an-open-alpha* to yield an open ALPHA and evaluates *an-expr* to yield a Symmetric Lisp constant or form. It then attaches this form to the open ALPHA, where it will be evaluated in the context of the growing environment of which it is now part. Specifically, the ATTACH! expression above causes the "*" within *an-open-alpha* to be replaced by

```
( ALPHA expr-value * )
```

where *expr-value* is the result of evaluating *an-expr* in the environment of the caller of the ATTACH!.

Thus, if we start with our newly-created *environ1* and evaluate

```
( ATTACH! environ1 '( NAME foobar (+ 2 3)))
```

environ1 becomes

```
( ALPHA
   ( ALPHA ( NAME foobar (+ 2 3) * ))
```

which evaluates to

```
( ALPHA
   ( ALPHA ( NAME foobar 5) * ))
```

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.

Thus if, having evaluated the ATTACH! given above, we proceed to evaluate in succession

```
( ATTACH! environ1 '( NAME bazball (* 2 foobar)))
( ATTACH! environ1 'bazball)
( ATTACH! environ1 '( NAME bazball 137))
( ATTACH! environ1 'bazball)
```

we get

```
( ALPHA
   ( ALPHA ( NAME foobar 6) ; already evaluated
     ( ALPHA ( NAME bazball (* 2 foobar))
       ( ALPHA bazball
        ( ALPHA ( NAME bazball 137)
         ( ALPHA bazball)))))
```

which evaluates to

```
( ALPHA
   ( ALPHA ( NAME foobar 6)
     ( ALPHA ( NAME bazball 12)
       ( ALPHA 12
        ( ALPHA ( NAME bazball 137)
         ( ALPHA 137)))))
```

An open ALPHA shares some resemblance with a stream [3,22] with two important differences: (1) elements of an open ALPHA are evaluated not by a stream generating function but by the open ALPHA itself and (2) unlike conventional streams, open ALPHAs may be appended-to by arbitrarily many processes. In this sense, an open ALPHA acts as a multistream or a parallel queue[9], with the ATTACH! operator serving as an atomic append operation. It is from the enriched semantics of multi-streams over ordinary streams that open ALPHAs derive their expressive power. An open ALPHA can act like a stream - it need simply be built by a tail recursive function. It is not possible, however, to use a stream to model the open ALPHA in full generality.

Note that 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 (if there is one) has been entered in the symbol table.

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.

Open ALPHAS blur the distinction between processes and data structures. Since they obey the same scoping rules as normal alphas and can be operated on by the same operators, we can view them as simply a more structured version of the basic ALPHA form. Because open ALPHAS are extensible and their elements are evaluated in parallel, they can also be viewed as dynamic process creators. Their flexibility makes them an ideal tool for a wide variety of applications, among them the Symmetric Lisp Interpreter.

4 The Symmetric Lisp Interpreter

The role of the interpreter is simply to solicit expressions from the user and inject them into an environment of the user's choosing. One way to express such a routine is as follows:

```
( NAME interp
   ( LAMBDA (user-env)
      ( NAME input ( READ))
      ( IF ( EQUAL input 'exit)
         (quit)
         ( ATTACH! user-env input))
      (interp user-env)))
```

This outline interpreter (which neglects matters of printing and formatting) behaves in the following way. Consider what happens when the user types

(*interp env*)

The *interp* function reads expressions from the user, compares each against a termination symbol and, if the user has not called for termination, attaches a new element to the open ALPHA *env*. Each new element is evaluated, once attached, in the context of *env*. ALPHA's parallel evaluation rule doesn't compromise the correctness of the interpreter, because ATTACH! works atomically – user input will be attached to *env* in the correct order even though many input forms may be undergoing evaluation simultaneously. The dataflow-style synchronization rule for name access guarantees that, when expressions require values that are still being computed, they will wait until the values are known and then proceed.

The environment constructed by the Symmetric Lisp interpreter (unlike its counterpart in other interpretative languages) is transparent and denotable. If *curr-env* is the name of an interpreter-constructed open ALPHA, we can evaluate *expr* in this environment by writing (WITH *curr-env expr*). (WITH is sensitive to the ALPHA type of its first argument: (WITH *some-open-alpha expr*) causes *expr* to be evaluated as if it were substituted for *some-open-alpha*'s "*".) We can display *curr-env* just as we can closed ALPHA's: the expression *curr-env* yields the value:

```
( ALPHA ( ALPHA ... * ))
```

Because environments are denotable, it is possible for users to create any number of disjoint (or shared) simultaneous interpreter sessions, or even nest sessions within sessions. Because open ALPHAS are transparent, it is also simple to have disjoint interpreter sessions share information with one another. The invocation (*find some-open-alpha env*) searches down *some-open-alpha* for an environment need *env*:

```
( NAME find
   ( LAMBDA (oa env)
      ( PRIVATE acar-val ( ACAR oa))
      (if ( EQUAL ( NAME-OF acar-val)
                  ( NAME-OF env))
         acar-val
         (if ( END-OF-ALPHA? oa)
            nil
            (find ( ACDR oa) env)))))
```

The function NAME-OF returns the name symbol if its argument is a NAME or PRIVATE form, *nil* otherwise. The predicate END-OF-ALPHA? returns true if ACAR of the open ALPHA that is its argument is "*". Using *find*, an expression in *session1* can ask for the value of name *x* in *session2* thus:

```
( WITH (find global session2) x)
```

where *global* is the global environment or world ALPHA.

5 Parallelism

The function *interp* defines a parallel interpreter. Once an expression has been read, its evaluation can proceed concurrently with expressions input subsequently; the ATTACH! operator acts as the process creator. Expressions that refer to names whose values are not yet known simply block (as usual) until a value is computed, and then proceed.

What use is parallelism in this context? Users often need to perform several computations simultaneously. Conventional machines use multiprogramming to support concurrency; parallel machines can support logical concurrency with genuine parallelism. But how is logical concurrency presented to the user?

Traditional solutions along the lines of a Unix fork aren't very satisfactory, because they place on the file system the burden of keeping track of results yielded by background processes. In Lisp-based machines[12], to take another example, users need to wrap expressions that are to be executed concurrently in a complicated **process** construct (implemented using expensive coroutines); the top-level interpreter remains strictly serial.

Symmetric Lisp, on the other hand, allows concurrent evaluations to coalece naturally into a shared naming environment. Suppose, for example, that a user chooses to run four test cases of some function in parallel. He might type

(NAME *first* (*some-function some-args*))

and then

(NAME *second* (*some-function some-other-args*))

and so on; he might then go off and do some other Symmetric Lisp business. The value returned by the first test case will be accessible, as soon as it is complete, under the name *first*; to inspect this value, the user simply types *first* to the interpreter. (He might jump the gun and type *first* before the associated computation is complete. Evaluation of the expression "*first*" accordingly blocks – but the *interpreter* doesn't block; it stands ready, as always, to accept new input despite the fact that previous expressions, in this case including both "(NAME *first* ...)" and "*first*", are still under evaluation.) If, after creating the four parallel test-case jobs, the user chooses to format the results of the first two and send them to the printer, he might enter (*print* (*format first second*)). *format* blocks until values *first* and *second* are available, and then proceeds.

Functionality of this sort is available in variants of the Unix shell, and again in the context of modern window-based command monitors. But the Symmetric Lisp interpreter operates without the distracting, extraneous influence of a file system or a window monitor. It interfaces cleanly to other pieces of an expanding program structure because the environments that it manipulates are directly accessible to the user. The inherently parallel semantics of environments in Symmetric Lisp leads not only to a simple formulation of a parallel interpreter, but to improved modularity of the user-system interface as well.

An ALPHA form can be also be used to implement **cobegin-coend** style statements. Consider a parallel algorithm that is structured as a controller and a series of identical workers. Entering the expression

(ALPHA (*control*) (*worker*) (*worker*) (*worker*))

creates a four-thread parallel computation (because the ALPHA, like the open ALPHA, has a parallel-evaluation rule).

Like functional language programs that are intended for parallel evaluation, Symmetric Lisp programs can be explosively parallel: every environment, and every function application as well, allows parallel evaluation to take place. One likely Symmetric Lisp host is a parallel graph reduction machine[21,17]. Environments have a natural graph representation, leaves of the graph being ALPHA expressions and internal nodes representing NAME forms. There are opportunties for a compiler to introduce appropriate constraints on degree of parallelism during ALPHA or open ALPHA evaluation by analysing the structure of this graph. The interpreter doesn't have the luxury of performing semantic analysis over the program graph, and must, therefore, rely on heuristics in determining where to bound parallelism. Our current implementation of a concurrent interpreter retains a parallel evaluation semantics for open ALPHAs and for all ALPHA forms typed at the top level of the interpreter loop. Functions and ALPHAs encountered elsewhere are evaluated sequentially.

6 Persistence

Persistency refers to a data structure's outliving the program that created it. Few programming languages support persistency; most require instead the use of an external storage agent such as a file system to manage long-lived data. A notable exception is PS-Algol[4], in which persistence is embedded within the semantics of data types. The vast majority of other languages treat persistency as an extra-linguistic property, one that can be used in conjunction with the facilities provided by the language while remaining external to the language.

Environments in Symmetric Lisp, on the other hand, are persistent objects – an ALPHA's lifetime is independent of the expression that creates it. Any element attached by the interpreter to an open ALPHA exists for as long as the open ALPHA does (unless the user explicitly removes it). Such an interpreter-built environment is in turn one element of a global "world ALPHA", and it too exists indefinitely.

The open ALPHA's persistence allows us to use interpreter-created environments as file systems. They have the attributes that are important in modern file systems: they are named collections of named elements; sub-environments may be freely nested within them. Thus a file system takes the form

```
( NAME file-system
   ( ALPHA
         ( NAME directory-1 ( ALPHA ... ))))
```

Such a file system has the unusual characteristic of having a structure, type and organization that are completely specified by the user. To make a directory *my-dir*, for example, it's sufficient to type

```
( NAME my-dir ( OPEN-ALPHA))
```

to the interpreter. Adding a file to *my-dir* is accomplished by ATTACHing to it. Of course, a workable file system is more than simply a heterogenous collection of elements; there are issues of protection, version management and so on to address. But in Symmetric Lisp, these issues are variants of environment management and access; we consider some of these questions in thex next section.

6.1 Meta-Cleanliness

Symmetric Lisp gives users a high degree of control over their computing and naming environments. More unusual, it makes system-level routines clean and easy to write. Herring and Klint[15] write that two necessary conditions that need to be satisfied in designing a language-based computing system are (1) elimination of any distinction between files and data and (2) absence of any distinction between the naming and typing of files and variables. We agree, and observe that the semantics of ALPHA already contains all the ingredients necessary to produce such a unified framework.

Consider the following simple example. Suppose that we have implemented a file system in which each element of a directory ALPHA is either (1) another ALPHA, in which case it is a nested sub-directory, or (2) not an ALPHA, in which case it is to be treated as a file. Thus

```
( NAME some-directory
   ( ALPHA
         ( NAME file1 ... text ...)
         ( NAME file2 ... text ...)
         ( NAME sub-directory ( ALPHA ...)))
```

and so on.

The user needs a function *ls* to list the contents of a directory, tagging each name with either *file* or *dir*. This function is easy to write:

```
( NAME ls
   ( LAMBDA (directory)
      ( AMAP list-one directory)))
```

AMAP maps a function to the elements of an alpha. *list-one* is:

```
( NAME list-one
   ( LAMBDA (elt)
      ( PROGN
            (print ( NAME-OF elt))
            (if ( ALPHA? elt)
                  (print 'dir)
                  (print 'file)))))
```

Writing a *ls-rec* function to print the contents of all nested directories is just as simple.

In approaching non-trivial examples, the principle is the same: the environment is a simple data structure that is accessible to and manipulatable by the user.

Parallelism and meta-cleanliness together lead to other kinds of expressiveness not available in other interpreter-based programming environments. It's easy, for example, to specify and create daemon processes that watch streams or environments for new developments.

Consider, for example, the following definition of a mail daemon:

```
( NAME monitor-mail
   ( LAMBDA (stream)
      ( NAME mail-message ( ACAR stream)
      ( PROGN
            (display-message mail-message)
            (monitor-mail ( ACDR stream))))))
```

To use the daemon, a user types (*monitor-mail my-mail-stream*). The evaluation of this function runs indefinitely – but of course, once again, the interpreter itself doesn't hang: it's ready to receive new user input despite the fact that the evaluation of a previous expression, here "(*monitor-mail my-mail-stream*)", is still ongoing. Whenever a new message is appended (using ATTACH!) to *my-mail-stream*, the daemon displays the message and then quiesces to await the next one. Users can simultaneously read mail by traversing *my-mail-stream* using ACAR and ACDR.

Parallelism and meta-cleanliness form an interesting and vigorous symbiosis. Because all file-like operations in Symmetric Lisp are actually operations over ALPHA forms, parallelism is inherent in any file operation. The transparency of ALPHAs coupled with their self-synchronizing parallel semantics makes the Symmetric Lisp system interface especially clean and simple.

Consider the following situation: the user wants to run many test cases of a program Q concurrently; he wants to analyze the results of each using an analysis program *analyze*; whenever an analysis turns up a "best result so far", he wants the results entered in a *best-results* directory and a message printed to the terminal screen.

One convenient way to go about this is to create a new environment called *test-runs*, and ATTACH! as many test runs as are needed:

```
( NAME test-runs ( OPEN-ALPHA))
( ATTACH! test-runs (Q first-set-of-args))
( ATTACH! test-runs (Q second-set-of-args))
```

and so on...

All test runs will (of course) evaluate in parallel. We can now set up the analysis stage by typing

```
( AMAP analyze test-runs)
```

and we apply the following function to the result:

```
( NAME inspect
     ( LAMBDA (an-analyzed-stream best-results)
          ( PROGN
               (check-out ( ACAR an-analyzed-stream)
                         best-results))
               (inspect
                    ( ACDR an-analyzed-stream)))))
```

where *check-out* looks at an element, decides whether it represents a "best so far" and, if so, prints a message and attaches its parameter to an open ALPHA called *best-results*.

This is a simple solution to what would be, in most systems, a very complicated problem. (The reader is urged to express the same thing in his own favorite programming environment, and compare.)

7 Related work.

Symmetric Lisp bears an important resemblance to the sort of *monolingual programming environments* described generally by Heering and Klint[15] and, particularly to systems like Smalltalk[11] and Interlisp[20] or Cedar[19]. Unlike InterLisp, Symmetric Lisp treats files and data structures in the same way; it shares many of the goals of Smalltalk, but the two designs differ in dramatic and obvious ways. The work of Dennis[7] in building a general-purpose functional programming system based on dataflow principles addresses many of the issues of modularity we have raised here, but it is based on a radically different programming and execution model. Symmetric Lisp's resemblance to non-language-based program development environments like Gandalf[13] or Unix tools is less pronounced.

Whereas parallelism in other parallel Lisps like MultiLisp[14] or Qlambda[8] is introduced explicitly through control structures (such as PCALL or QLET), parallelism in Symmetric Lisp is intrinsic, revolving around parallel data structures. Insofar as parallelism occurs implicitly through data structures, Symmetric Lisp bears some similarity to functional languages implemented on data flow[7] or parallel graph reduction architectures[17]. Unlike these languages, however, Symmetric Lisp allows side effects and retains a semantics suitable for reasoning about dynamic process invocation. Connection Machine Lisp[18] shares some of the goals of Symmetric Lisp but, while CM-Lisp is best suited to architectures intended to exploit fine-grained parallelism[16], Symmetric Lisp is intended as a high level language for a general language-based parallel workstation, with the Symmetric Lisp interpreter serving as the interface between users and the multi-machine architecture. Symmetric Lisp contains the necessary ingredients to support a parallel object-oriented programming model as described in [1] but without much of the conceptual overhead that these languages introduce. The treatment of bindings as denotable values in Pebble[5] is similar to the semantics of bindings in ALPHA forms but the use of types as values in Pebble as well as the absence of a concurrency semantics distinguishes it from Symmetric Lisp.

8 Conclusions.

Symmetric Lisp is intended to be a host-language for a language-based parallel workstation. The interpreter is to serve as the primary interface between users and the machine. Expressions typed at the terminal are translated into an intermediate form suitable for evaluation by the interpreter implementing the operational semantics of the language. There are a variety of abstract execution models that appear well-suited to Symmetric Lisp; among them are parallel graph reduction [17,21] (described earlier) and the programming model supported by Linda machines [6].

Symmetric Lisp is currently implemented as a concurrent interpreter written Common Lisp and running on the TI Explorer.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* PhD thesis, 1985.

[2] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction*, 1986.

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

[4] 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.

[5] Robert Burstall and Butler Lampson. A Kernel Language for Modules and Abstract Data Types.

[6] Nick Carriero, David Gelernter, and Jerry Leichter. Distributed Data Structures in Linda. In *Proceedings of the ACM Symp. on Principles of Programming Languages*, Jan. 1986.

[7] Jack Dennis, Joseph Stoy, and Bhaskar Guharoy. VIM: An Experimental Multi-User Computer System Supporting Functional Programming. In *1984 Conf. on High-Level Computer Architecture*, 1984.

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

[9] David Gelernter and Suresh Jagannathan. *A Programming Language Supporting Multi-Streams*. Technical Report, Yale University Technical Report, 1987.

[10] David Gelernter, Suresh Jagannathan, and Thomas London. Environments as First-Class Objects. In *14th Principle of Programming Languages Conf.*, 1987.

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

[12] R. Greenblat, T. Knight, J. Holoway, D. Moon, and D. Weinreb. The LISP Machine. In *Interactive Programming Environments*, pages 326–352, McGraw-Hill.

[13] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Trans. Softw. Eng.*, 1117–1128, Decemeber 1986.

[14] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, October 1986.

[15] J. Heering and P. Klint. Towards Monolingual Programming Environments. *ACM Transaction on Programming Languages*, 183–213, July 1985.

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

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

[18] Guy Steele Jr. and Dan 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.

[19] W. Teitelman. A Tour Through Cedar. *IEEE Software*, January 1984.

[20] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *Computer*, 25–34, April 1981.

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

[22] K.-S. Weng. *An Abstract Implementation for a Generalized Data Flow Language*. Technical Report.