Using CLOS-like Concepts in a Prototyping System

Richard P. Gabriel
Stanford University

We have been working for about 9 months on the design of a prototyping system, which is a language and an environment for creating mixed Lisp and Ada prototypes. The environment and language use CLOS-like concepts. This precis describes the language concepts we are developing.

Traditional languages like ADA use a model of sequential control with a side-effectable memory. Combination or aggregation mechanisms are used to compose complex data structures from simpler ones. Most programming languages provide extensive mechanisms for defining abstract data types from primitive ones, but very few programming languages provide mechanisms for defining control abstractions (besides procedures) or other non-data abstractions. We believe that prototyping is largely a process of composing or aggregating complex prototypes from simple ones, and sometimes such composition is performed with source code and sometimes not. Therefore, a prototyping environment is one whose primary function is to compose and control existing source and executable mixed language programs and modules.

In particular, we are interested in developing a richer computational model for a prototyping language than the ones used by Ada and Lisp so that prototypes can be manipulated by manipulating this model.

Over the last 6 months we have adopted the following approach:

1. Select a problem for which we wish to develop a prototype.
2. Suppose some existing code that accomplishes some part of the task or a similar task.
3. Determine the program that should constitute the prototype.
4. Infer the partial programming model from that prototype.

The following constraints are also being observed. Note that several are environmental rather than programming concerns:

1. Do not require that the existing code be copied out of an existing configuration. That is, the environment should take care of versions and merging conflicts, up to a point.
2. Where possible, enable the prototyper to alter the context of execution (or compilation, if that's the way you want to think of it—they are the same thing) in order to achieve the behavior needed for the prototype.
3. Use *annotations* to enable the prototyper to keep a record of transformation made by hand or by machine and the reason for the transformation so that the history of

---

The work was supported by DARPA, contract number N00039-84-C-0211.

development is retained. A piece of source code can be annotated by the program that will transform that source to the desired form. The *abbreviation* of the annotation will be the transformed code. The original source code remains intact. When compiling the source code, the compiler will fork off a process to either perform the transformation or retrieve it, and supply the character stream (or tokenized stream) in place of the original text. Annotations and abbreviations will be described in the next section.

4. Do not rule out visual presentation or language, but insist on a textual representation for every prototype.

1. Annotations and Abbreviations

An ***annotation*** is a generalization of hypertext. Annotations are used to integrate the various prototyping tools and to provide an extensible base. An annotation is a relation that links two or more objects in the environment. An annotation is an instance of an extensible class hierarchy. Annotations are active in the sense that methods can be associated with them that are triggered whenever a tool of a certain class operates on an annotation of a certain class that links objects of certain classes. Normally a region of source code is annotated with some information.

The second part of the annotation model is ***abbreviations***. When an object is annotated with some other things, the annotation in general must be displayed when the object is displayed. There are several options. First, the object can be displayed as usual but in a mouse-sensitive manner, so that the annotation can be displayed by interacting with the mouse-sensitive object. Second, the annotation can be displayed as an icon either in place of the object or adjacent to it. Third, the annotation and the object can be displayed in an abbreviated manner. For example, if a comment in a program is turned into an annotation, that comment could be displayed as follows:

```
(defun f (n)
  ;;; Logarithmic Fibonacci function....
  (...))
```

Here the one line comment is an abbreviation of a much larger comment that explains the algorithm. The derivation of the abbreviation could be by taking the first three words of the comment, or it could be some programmer-defined object.

Annotations and abbreviations form the basis for communication, integration, and user interface in the prototyping environment. For example, annotations are used in source level debugging: When a program is halted, the source code is displayed with each variable annotated with its current value. If a function is recursive, further annotations on each variable will provide the values it has in enclosing invocations, so that they may be simultaneously viewed in place in the source.

However, describing the environmental model for prototyping is not a goal of this precis.

2. Samefringe

In following the approach outlined above, one problem in particular has occupied us: The problem is samefringe. This problem was suggested many years ago as the simplest problem requiring multiprocessing to solve satisfactorily. As it turns out, it can be solved quite well and elegantly using serial techniques. The problem is as follows. Assume that T1 and T2 are two binary trees whose leaves are non-NIL atoms. Let L1 be the list of leaves encountered in a preorder tree walk of T1, and let L2 be the corresponding list for T2. Then T1 and T2 have the same fringe if L1=L2. In Lisp, the trees have the same fringe if, when printed, the same atoms appear in the same order. We have restricted leaves to be non-NIL atoms to simplify the problem without loss of generality.

One obvious way to think about how to program this is to envision a tree traversal program like this (in Lisp):

```
(defun traverse (tree)
  (cond ((atom tree) (report tree))
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

Here REPORT simply notes its argument somehow. First we want to set up two processes, one to traverse T1 and the other to traverse T2. Then we set up a third process to look at the atoms coming from each traversal. If any two corresponding atoms are not the same, the processes are terminated and the answer is nil. Otherwise, if both traversal processes end normally, the two trees have the same fringe if both processes terminated after reporting the same number of atoms.

We can assume that the traversal routine is given—it is part of the stock of programs available to the prototyper for re-use. The following comparison routine checks a pair of atoms from each tree:

```
(defun compare (x y)
  (unless (eq x y) (return-from samefringe nil)))
```

We can assume this is given as well.

Later when the visual approach to prototyping is presented we will see that the concept of "process" is possibly unnecessary to solve the problem. This is because the prototyping environment provides a mechanism to make two copies of the source code and to indicate that each will receive one of the trees to work on. Of course, the textual representation of the program will involve processes, and we will be solely concerned with that textual representation except in the section "Illuminated Code."

Given the basic approach outlined earlier, the following seems to be the naive code to solve the problem:

```
(defun samefringe (tree1 tree2)
  (qflet t ((compare (a1 a2)
              (unless (eq a1 a2)
                (return-from samefringe nil))))
    (labels ((traverse (tree)
               (cond ((atom tree) <report-tree-to-compare>)
                     (t (traverse (car tree))
                        (traverse (cdr tree))))))
      (qprogn t
        (traverse tree1)
        (traverse tree2))
      <test-for-same-number-of-atoms-reported-from-each-process>)))
```

The QFLET expression creates a process running COMPARE, and the QPROGN expression spawns two processes, each running TRAVERSE. The T in the QFLET expresssion indicates that a process is always to be created.

Now we must complete the program. The remaining presentation follows the lines of exploration we followed while trying to do that. Often studying such lines can be as illustrative as any of the intermediate points and certainly more than the endpoint alone. At the end of this precis we have the most recently developed version of the samefringe program, and it is not yet entirely satisfactory.

The first step is to solve the communication problem between the expression `<report-tree-to-compare>` and COMPARE.

3. Partially, Multiply Invoked Functions

The basic idea of partially, multiply invoked functions (PMI Functions) is to separate the process of coordinating the arrival of arguments from the process of executing the function on those arguments.

All calls to a function go through an interface to the actual implementation. The implementation of the function receives arguments by position, while the interface accepts only named arguments, provides for all defaulting, and coordinates the arrival of arguments for the function from multiple sources.

The Qlisp work produced the basic idea of PMI functions in late 1987, but the Qlisp formulation has a serious drawback. Namely, arguments need to have names, which often requires those names to be passed about as additional arguments so that the proper values were assigned to the proper parameter names.

Here is a simple example of the Qlisp-style technique:

```
(pmi-defun add-up (x y) (:summand1 :summand2) (+ x y))
```

This function adds up a pair of arguments, called X and Y. The expression

```
(add-up :summand1 1 :summand2 2)
```

simply produces the answer 3. However, one can partially invoke the function as follows:

```
(add-up :summand2 2) -> future
```

In this case the interface remembers the supplied argument and returns a future. A second call will complete the invocation and supply a value to the future:

```
(add-up :summand1 1) -> 3 = <the same future as above>
```

This technique, then, is similar to currying functions, but because all arguments to the interface are named, one does not need to curry in any particular order. One could term it ***dynamic currying***. All calls to a PMI function that supply arguments to the same invocation receive the same future as their value. A future is returned whenever some required arguments to the function have not been supplied to the interface by a function call. If a particular invocation of a function has returned a future, the value returned when all required arguments have been supplied is a realized future. This is to preserve EQ-ness of all values returned for a particular invocation.

Also, when not all arguments are supplied for a particular call, futures are supplied for unsupplied arguments. This way, PMI functions can be truly partially invoked.

For example, we can produce a list of the sums from two streams supplied by two processes as follows:

```
(let ((answer (make-queue)))
  (pmi-flet ((add-stream (x y) (:summand1 :summand2)
               (add-queue (+ x y) answer)))
    (qprogn t
      (loop ...
        (add-stream :summand1 <computation>)...)
      (loop ...
        (add-stream :summand2 <computation>)...))
    answer))
```

The PMI-FLET expression creates a local PMI function. The details of queue management are elided.

It is possible to use streams, pipes, or channels to write SAMEFRINGE and many other programs. It would seem that these other mechanisms are superior because they are simpler, more easily understood, and more intuitive. On the other hand, using these mechanisms requires "wiring up" networks of channels, either statically or dynamically. In either case there would be code to effect the wiring. With PMI functions the items being passed are sent to their destinations, labeled with the argument to which it corresponds, and there is no need to identify, centralize, or explicitly name sources of arguments, only

destinations. Thus it is easier to code up loose collections of processes that communicate variously with one or several central processes.

In this way PMI functions mimic pure message-passing, where the sender does not need to be connected to the receiver. The advantage of PMI functions over classical message-passing is that the recipient can be a function of many arguments, whereas the suppliers of those arguments need not be otherwise co-ordinated to send them.

Furthermore, PMI functions preserve functional style, and there is no restriction that a single invocation of a PMI function must pass exactly one argument. For example, suppose ADD-STREAM takes three arguments, sums them, and adds them to the queue. And suppose that each of the two processes supplying the summands alternate supplying one and two of the summands. We could write this as follows:

```
(let ((answer (make-queue)))
  (pmi-flet ((add-stream (x y z) (:summand1 :summand2 :summand3)
               (add-queue (+ x y z) answer)))
    (qprogn t
      (loop ...
        (add-stream :summand1 <computation>)
        ...
        (add-stream :summand1 <computation>
                    :summand2 <computation>)
        ...)
      (loop ...
        (add-stream :summand2 <computation>
                    :summand3 <computation>)
        ...
        (add-stream :summand3 <computation>)
        ...))
    answer))
```

This is possible with streams and channels, but the cost is that the mechanism for transmitting a variable number of arguments and co-ordinating them must be exposed.

3.1 *PMI Functions and Parallel Processing*

A PMI function can run within a separate process (though the programmer need not know that). Using this combination we can now write the samefringe program as follows:

```
(defun samefringe (t1 t2)
  (pmi-pflet t ((compare (a1 a2) (:leaf1 :leaf2)
                  (unless (eq a1 a2)
                    (return-from samefringe nil))))
    (let ((end-marker (list nil)))
      (labels ((traverse (tree arg-name)
                 (trav tree arg-name)
                 (qwait (compare arg-name end-marker)))
               (trav (tree arg-name)
                 (cond ((atom tree) (compare arg-name tree))
                       (t (trav (car tree) arg-name)
                          (trav (cdr tree) arg-name)))))
        (qprogn t
          (traverse t1 :leaf1)
          (traverse t2 :leaf2))
        t))))
```

We have used Qlisp primitives to fill in the glue portions of the code so that a complete program can be presented. We will try to eliminate those primitives with additional programming model material.

There are some strong points and some weak points of this code. The form PMI-PFLET defines a process that contains a PMI function. The arguments are named :LEAF1 and :LEAF2. The answer NIL is immediately returned if the leaves in any pair sent to COMPARE are not the same. The value of END-MARKER is used as a signal that the traversal of a tree is complete. It is used to stop the traversal of differently fringed trees when the smaller one has been traversed.

The code for TRAV is very similar to the basic traversal function presented earlier, but includes an invocation of COMPARE in which the passed-in name for the argument is used for the leaf. The code for TRAVERSE here includes the call to COMPARE with the end marker.

Finally, QPROGN creates two processes, one to traverse each tree. When both processes have finished the following is true: Each process has sent all of the atoms it found to compare. Furthermore, each has waited (using QWAIT) for the response of the COMPARE process to the end marker signal. This means that all possible pairs that COMPARE could compare have been compared, including the end markers. Therefore, if both processes have completed, the answer must be T.

There are other, interesting aspects of PMI functions that we have not discussed. A similar set of constructs can be found in (Lamping, L&FP 1988).

4. Process Environments

The above treatment of the samefringe problem is not satisfying, however. For example, naming the arguments appears to be adding too much detail to the solution—the reason for naming arguments is to distinguish arguments from different processes. In fact, passing argument names around seems like a hack rather than a clean approach. The function TRAVERSE has been distorted in order to provide a signal that a process has completed its traversal. The QPROGN guarantees that both processes have terminated before T is returned.

What we see sprinkled through this program are ad hoc mechanisms that either deal with controlling the algorithm in the presence of processes whose time-domain behavior is unpredictable or serve to direct information from one process to another using markers (here, the means of identifying processes is analogous to identifying rivers by dropping colored dyes into them).

We have searched for some solutions to this problem using an object-oriented approach. One solution is called ***process environments*** and is similar to Qlisp's heavyweight futures. A process environment is a set of processes, each of which is an instance of the class named PROCESS. The set of these processes is treated as an object that can be manipulated by prototype code. Subsets of the processes can be distinguished by their classes, and control code can be written whose behavior depends on the state of the set of processes or the states of any of the processes in a process environment. When certain important states are achieved or events occur within the environment, specific generic functions are invoked, and these generic functions may be specialized. For example, when a process terminates normally, the generic function TERMINATE is invoked by the system; its default method returns t. A process environment might correspond to an execution of existing code or of prototyping code.

Let's review the relevant parts of the Common Lisp Object System (CLOS). A generic function is a function whose implementation is as a set of methods some of which are invoked when arguments satisfy class constraints. For example, suppose there is a generic function named DISPLAY which takes two arguments, a thing to display and the display itself. Suppose further that there are graphical objects, textual objects, graphical displays, and textual displays. Then one might define display with the following four methods:

```
(defmethod display ((obj graphical-object) (disp graphical-display)) ...)
(defmethod display ((obj graphical-object) (disp textual-display)) ...)
(defmethod display ((obj textual-object) (disp graphical-display)) ...)
(defmethod display ((obj textual-object) (disp textual-display)) ...)
```

Calls to display are all syntactically similar, but which of the above methods is invoked depends on the classes of the two arguments. If OBJ is textual and DISP is graphical, the call

```
(display obj disp)
```

will invoke the third method, which is presumably customized code for that situation.

The first problem with the prototype code we can solve with a CLOS-like idea is the problem of argument naming in PMI functions. The first argument to COMPARE is just the one that came from one of the two processes and the second argument is that one that came from the other process. If values were tagged with the process from which they came, we could use that information to distinguish the arguments to COMPARE. (Note that the ability to "tag" a value with a process implies a powerful underlying mechanism. That mechanism, called ***hybrid inheritance***, will be discussed later.)

So, we define processes to be instances of classes, and we allow values to take on as an additional characteristic the class of the process that produced it. This might look like this:

```
  ...
(class-let ((p1 (process) ...)
            (p2 (process) ...))
  (pmi-flet ((compare (x:p1 y:p2) (unless (eq x y) (return-from ...))))
    ...
    (labels ((traverse (tree)
               (cond ((atom tree)
                      (compare tree))
                     (t
                      (traverse (car tree))
                      (traverse (cdr tree))))))
      (spawn p1 (traverse tree1))
      (spawn p2 (traverse tree2)))))
  ...
```

We use the notation x:c to indicate that the argument to be bound to x must be an instance of the class c. Note that this conflicts with Common Lisp package notation, which we have chosen to abandon.

At this point TRAVERSE looks about the way we want it to look. SPAWN creates an instance of a process and starts it running:

```
(spawn <class> <expression>)
```

This seems to adequately solve the problem of arguments from two different sources being delivered to a single function that requires them. The remaining problem is termination. There are two basic approaches, one using a strategy of class-based descriptive techniques and the other using monotonic variables. The monotonic variables approach will be presented in the section "Monotonic Variables."

A common technique in CLOS is to customize the behavior of a program by allowing the programmer to define methods on generic functions that are called whenever certain conditions hold or events occur. For example, in CLOS itself, if a generic function is

invoked and no methods are applicable, the generic function NO-APPLICABLE-METHOD is invoked. There is a default method for this function that signals an error, but if there is a more specific method, it will be invoked, providing customized behavior in that case.

Some of the problems with the previous samefringe solution can be addressed by providing a vocabulary for talking about processes and their activities. In some ways, this approach is akin to adding a reflective layer in which the system is able to refer to its own activities and to alter or customize them.

A natural way to think about a process is in terms of the essential nature of the state that it is in—for example, it is running, it is terminating, it has been killed, or it has terminated. CLOS provides a vocabulary for discussing an object in terms of its class. That is, an object is distinguished from others primarily by its class-oriented characteristics—the level of abstraction is the class.

As we've already seen, the first step is to define processes to be instances of classes according to the means of their creation. Some classes will be behaviorally indistinguishable from others at various levels because they are instances of the same classes.

The second step is to define processes to be instances of classes according to their state of execution. When a process is running, it is an instance of the class of running processes; when it has terminated, it is an instance of the class of dead processes.

In CLOS, every object is an instance of some class, but the class of an object changes infrequently, usually as a result of programming environment activity. Our application of these ideas is for objects to not only change (possibly) frequently, but for classes to accrete to an object. That is, some particular object may dynamically become an instance of some class for some period and then stop being an instance of that class. The object does not necessarily shed any of its previous classes during this process, but simply adds the class to the set of which it is an instance.

One ingredient of our scheme that is not yet relevant to the samefringe problem is that a process is also an instance of the class named by the function it is currently executing. This provides a mechanism for talking about complex events such as when a process is executing one function inside of another. We will see an example of the use of this later.

4.1 *Hybrid Inheritance*

CLOS multiple inheritance probably does not support this behavior very well, so we use a different sort of inheritance structure, called ***hybrid inheritance***. In hybrid inheritance, a class hierarchy is defined within a ***domain***. Within each domain, inheritance is determined by the mechanism of that domain. A domain that supports instances frequently changing class or temporarily become an instance different classes is called a ***dynamic domain***.

There are two distinct aspects of hybrid inheritance. The first aspect is that if two classes are in two different domains, an object can be a direct instance of both of them. In multiple inheritance a similar effect can be achieved by defining a class that represents

objects that are instances of those two different classes. This might be a satisfactory approach, but it is not a necessary one.

An object can also be a direct instance of two classes within a domain that supports it. The characteristics of inheritance in different domain need not be the same or similar. For example, one domain might be a CLOS-like multiple inheritance system while the other is a Smalltalk-like single inheritance system.

The second aspect is that if an object is a direct instance of two different classes, those parts of the object associated from one class are completely separate from those parts associated with the other, so that such an object is really a simple composite of two parts, each of which is entirely retained. In CLOS multiple inheritance, for example, if a class inherits from two different ones, a process of rationalization takes place that handles any name conflicts that arise. Hybrid inheritance enables a programmer to define a domain within which such rationalization takes place, but also provides a mechanism to avoid such rationalization by enabling separate parts to be pieced together. Names in such a system are meaningful only with respect to a domain.

Using hybrid inheritance it is possible for objects to become instances of a class and then to cease to be an instance of a class. This is the mechanism we use for tagging a value with the process that produced it.

Hybrid inheritance implies that objects can be simultaneously instances of two incommensurable classes. Does that make sense? Take an example from a language with abstract data types. In such a language it is possible to define a queue. The representation of a queue might be a list. Let Q be a queue. To a program that is operating on queues, Q is an instance of the type `queue`. To a debugger being used on this program, Q might be instance of the type `list`.

In this example, it might be said that the *concrete* type of Q is `list` while the *abstract* type is `queue`. We see that different programs can see the same object as different types or classes.

Another category of examples of the need to inherit from more than one object arises when orthogonal properties are being combined. For example, the property of being ***persistent*** is orthogonal to almost all other properties a data structure might have. Persistence is the property of a data structure existing beyond the lifetime of the process or program that created it. A text file in an operating system is an example of a persistent data structure. One can imagine creating a class that not only inherits from a particular family chain, but also has the property of persistence mixed in.

Hybrid inheritance is also useful for other reasons. One is that a prototype might be made up of programs written in several different programming languages. If some prototype code needs to talk about the type or class of an object within its programming language or system, there needs to be a mechanism to distinguish types in different languages. Each language is a separate domain. And an object that is of one type in one language might be of another type when passed into a program in another language.

Moreover, the behavioral or structural characteristics of a type or class system will be different from language to language, and a piece of prototyping code must be able to operate within those domains of characteristics.

When defining a method, each specializer must be a combination of domain and class. A generic function has a signature that includes the set of domains on which it is specialized. If all methods are of this same signature, the generic function behaves exactly as a CLOS generic function. If they aren't, then each signature is equivalent to a method qualifier, and method combination is used to determine how to implement an effective method composed of methods with different signatures.

When the domain is obvious, it will be elided.

4.2 *Method Applicability and Hybrid Inheritance*

In CLOS, the method that is invoked depends on the classes of the arguments to it. We extend this so that the method that is invoked also depends on the class of the process that invokes it. For example, when a process terminates, the generic function TERMINATE is invoked on no arguments. We can specialize TERMINATE by the class of the invoking process to customize the actions of the function according to which sort of process terminated.

In CLOS, a method is applicable if the arguments to the generic function are of the right classes. In the domain of processes—in which the class of a process changes dynamically—we extend the concept of applicability as follows. Suppose a generic function is invoked which is made up of methods, some of which are specialized on classes within dynamic domains. If no method is applicable at invocation time, the generic function will wait until some method is applicable. When a method becomes applicable, the processes that changed the classes of the arguments so as to cause the method to be applicable are paused until the generic function terminates. Method combination provides a mechanism to define complex behavior based on the changing states of processes. For example, one type of method combination would wait for the all arguments to be of the right class at the same time, while a second type would freeze each as it becomes the right class.

4.3 *Samefringe*

Let's look at SAMEFRINGE now:

```
 1 (defun samefringe (t1 t2)
 2   (let ((unique (list nil)))
 3     (class-let ((p1 (process) ...)
 4                 (p2 (process) ...))
 5       (pmi-pflet ((compare (x:p1 y:p2)
 6                     (unless (eq x y) (return-from samefringe nil))))
 7         (gf-flet ((terminate:process () (compare unique))
 8                   (complete (p:dead q:dead) t))
 9           (labels ((traverse (tree)
10                      (cond ((atom tree) (compare tree))
11                            (t (traverse (car tree))
12                               (traverse (cdr tree))))))
13             (complete
14               (spawn p1 (traverse t1))
15               (spawn p2 (traverse t2)))))))))
```

The special form CLASS-LET defines a set of classes with lexical scope and indefinite extent. The special form GF-FLET is similar to Common Lisp's FLET but defines a local generic function by defining its methods.

There is a class named PROCESS of which every process is an instance. Because there are two trees being traversed, there will be two subclasses of PROCESS, one for each tree, called P1 and P2. The function COMPARE will be a PMI function that takes one argument from each of these two processes. When each process terminates, TERMINATE will provide the unique end marker to COMPARE.

The main program will spawn two processes, one an instance of P1 and the other an instance of P2. That main program will then apply COMPLETE to those two processes, where the only method for COMPLETE is applicable when both its arguments are dead processes. Here DEAD is the subclass of PROCESS consisting of processes that have terminated. Therefore, COMPLETE and hence SAMEFRINGE will wait until both processes have terminated.

Line 2 defines the unique object. Lines 3–4 define two distinct subclasses of PROCESS. These subclasses only exist during the dynamic extent of the CLASS-LET. The names P1 and P2 are lexically apparent only within the lexical scope of the CLASS-LET.

Lines 5–6 define the comparison function, which compares an object (X) produced by P1 to an object (Y) produced by P2. Line 7 defines the termination procedure for the processes. When a process of class P1 terminates, the system invokes TERMINATE as if by P1, and then P1 becomes an instance of DEAD. The characteristic of being of class P1 is inherited by this invocation of TERMINATE which will pass on that tag to UNIQUE when it invokes COMPARE.

Line 8 defines a generic function that returns T when invoked on two terminated processes. Furthermore, if this function is applied to two processes, either one of which is not terminated, invocation will be blocked until they both are terminated.

Lines 9–12 are the simple traversal routine. Notice that the call to COMPARE does not manipulate argument names because the arguments are tagged by the classes (processes) that produced them. Also notice that this traversal routine is exactly the one we started with.

Lines 13–15 are the main body of the function. Lines 14–15 spawn two processes, the first an instance of P1 and the second an instance of P2. The function COMPLETE will be invoked when the two processes terminate, and this function will return T in that event. However, recall that COMPARE might terminate the processes early. One fine point is that whenever a CLASS-LET is exited, all instances are killed without invoking TERMINATE on them. (In this case a generic function named KILLED will be invoked, but the default method, which is not shown, simply returns T.)

To be truthful, we still do not like this program. The use of the unique end marker still seems to be a hack, even though one could argue that it is nothing more that an end-of-stream marker. Later we will present another way of programming this function using monotonic variables, which might be a little better. Note also that we could have used some sort of stream abstraction to do the same thing. The prototyping language will probably also have such abstractions, and the above program is not intended to be the only or even best way to program the solution to this problem.

5. Illuminated Code

What we've seen is the textual endpoint of a prototyping activity. This endpoint is necessary to enable tools that operate on source code to have access to prototype code. The environment portion of the prototyping system is largely language-independent and tool-independent, as long as those languages and tools obey a particular set of protocols. We want source-level tools to operate within this framework.

We are working on a partially visual, mostly textual means of creating this prototype from existing code.

Illuminated code is code that is annotated with various environmental and linguistic constructs or notes that cause information to be gathered, moved about, or program structures to be created. For example, the following code is illuminated to count the number of leaves found during traversal:

```
                              (let ((leaf-count 0))
                                ----------------
                               |(incf leaf-count)|)
                                ----------------
(defun traverse (tree)      |
                             -------------
  (cond ((atom tree) |(report tree)|)
                             -------------
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

One interesting side point on this example is that the placement of the initialization of LEAF-COUNT to 0 is not important. It is important that it have been initialized outside the scope of TRAVERSE and the fragment that increases LEAF-COUNT. Note that the structure of the annotation implies these constraints. Let's quickly look at the full code to do something like this in Lisp:

```
(let ((leaf-count 0))
  (let ((increase-leaf-count #'(lambda () (incf leaf-count))))
    (defun traverse (tree)
      (cond ((atom tree)
             (progn (funcall increase-leaf-count)
                    (report tree)))
        (t (traverse (car tree))
           (traverse (cdr tree)))))))
```

We needed to specify several irrelevant things here. First is the name INCREASE-LEAF-COUNT. Second is the placement of the initialization of LEAF-COUNT. Third is the insertion of the call to INCREASE-LEAF-COUNT in TRAVERSE. The fourth is the inclusion of the definition in the nested LET expressions so that the visibility of the irrelevant name INCREASE-LEAF-COUNT would be correct. We could have avoided some of this by using an advise tool to modify the definition of REPORT to call INCREASE-LEAF-COUNT. Of course, we would need to make sure such advice did not conflict with other advice.

Annotations make the description of the essential parts of such additions easier, and a simple transformation system can chose a textual implementation and specify the irrelevant.

In general, the annotation/abbreviation model enables programs to be modified and viewed in different ways without destroying the original programs and without resorting to bulky versioning systems.

Here is the code we assume we're starting with to work on a prototype of SAME-FRINGE.

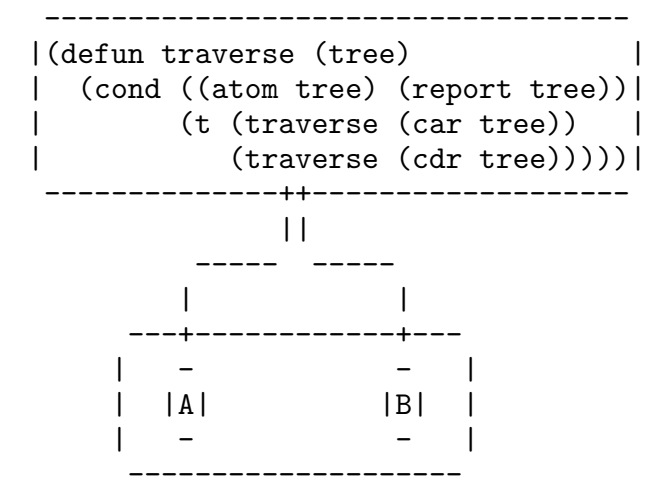
```
(defun traverse (tree)
  (cond ((atom tree) (report tree))
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

Using the annotation/abbreviation model environment, we would create a ***palette*** with two objects on it: Each is an annotation whose abbreviation is just the above code, and the annotation simply points to the configuration that holds that original code. The class of the annotation indicates that each of the annotations on the palette is to be regarded as a copy if it is modified. A palette is nothing more than a very large window that contains code for a particular prototype written in the prototyping language. As such it must be able to manipulate source and graphics, and its contents are to rendered into textual form. Every palette implicitly represents a process environment.

Here is what the palette looks like:

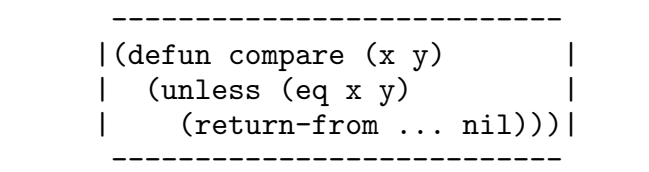
```
 ----------------------------------  ----------------------------------
|(defun traverse (tree)             ||(defun traverse (tree)            |
|  (cond ((atom tree) (report tree))||  (cond ((atom tree) (report tree))|
|         (t (traverse (car tree))  ||         (t (traverse (car tree))  |
|            (traverse (cdr tree))))||            (traverse (cdr tree))))|
 ----------------------------------  ----------------------------------
```

Below is a schematic form of the above situation. The definition of TRAVERSE is someplace in a configuration of other source code. There are two annotations on it, one linking to the object labeled A and the other linking to the object labeled B. A and B are on the palette, which is indicated by the box surrounding the boxes labeled A and B. A and B are displayed using the abbreviation that is simply the original source code. That is, the two boxes of code just above are the abbreviated display of the boxes labeled A and B below:



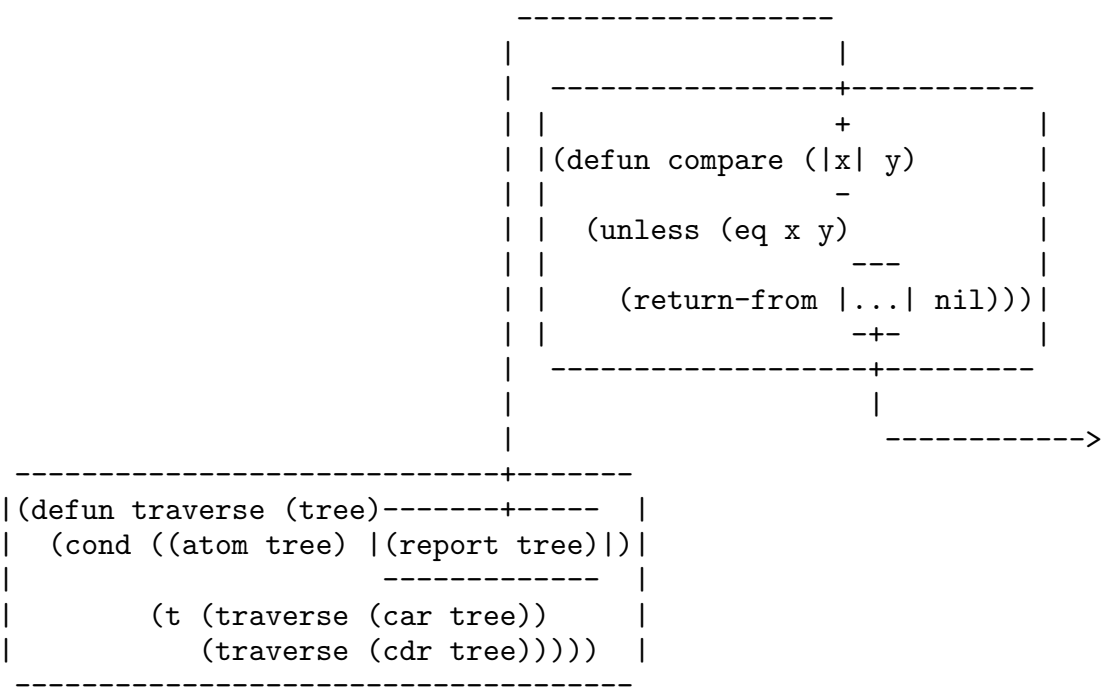
Looking back at the palette, we annotate each box with a note stating it is an instance of a class. This accomplishes stating that there are two subclasses of PROCESS, P1 and P2. We need not name them.

The palette represents the SAMEFRINGE prototype, and we now define COMPARE on it:

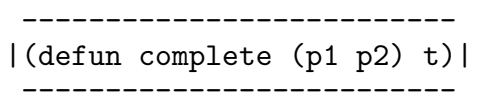


We annotate the argument X in the argument list in COMPARE with the form (REPORT TREE) in the left-hand instance of TRAVERSE, and the argument Y with the form (REPORT TREE) in the right-hand instance of TRAVERSE. Because the two instances are distinguished by the classes of the processes they are instances of, this is enough to generate the PMI definition of COMPARE. We annotate the ellipsis with the palette, which implies that if X and Y are not EQ, NIL is returned from the computation defined by the code on the palette.

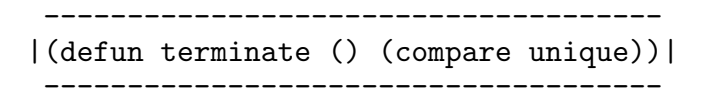
It's hard to show this without fancy graphics, but it would be something like this, using only the annotation on X as an example:



Now we write down the definition of COMPLETE:



The arguments are similarly annotated with the processes. Finally, TERMINATE:



This code is annotated with the two instances of TRAVERSE. Because TERMINATE is invoked by the system when the processes on the palette terminate, it is necessary only to place it on the palette with the annotation. Probably the remainder of the program can be completed automatically. The annotation is an instance of the class DEAD, which indicates that TERMINATE is invoked when the corresponding process dies.

The finished code seen in the last section could be generated from the palette and annotations. The palette is annotated with that finished code. The various parts of the code and palette are cross-annotated.

The traversal code we started with is exactly what we needed to incorporate into the prototype with minimal changes. Suppose we were not so well off and that the code we wanted to start with looked like this:

```
(defun traverse (tree)
  (operate tree)
  (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))
```

This code traverses trees exactly the right way, but it invokes an inappropriate operation at every node whether internal or leaf. To use this code we have several options:

1. Annotate the expression (OPERATE TREE) with this code:

   ```
   (when (atom tree) (compare tree))
   ```

   The effect of this is to alter the original code to be as follows:

   ```
   (defun traverse (tree)
     (when (atom tree) (compare tree))
     (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))
   ```

   The use of an annotation has the following effect. The actual code that is executed is as it appears immediately above. That code is derived from the original source code by substituting the destination of the annotation (the WHEN expression) for the source of the annotation (the OPERATE expression). However, the original source for TRAVERSE remains in the configuration as it always has, and only in the "mind of an execution engine" does the modified code exist. When the prototype is complete, the final code can be automatically derived by looking through the layers of annotations. In this case, we could further annotate this line and the following with a transformation that would collapse the WHEN followed by the UNLESS into a more compact conditional expression.
2. Annotate TRAVERSE with this;

   ```
   (class-let ((atomic-tree (predicated-class) #'atom))
     (gf-labels ((operate (tree:atomic-tree) (compare tree))
                 (operate (tree:t) nil))
       (defun traverse (tree)
         (operate tree)
         (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))))
   ```

   Here the idea is to slightly alter the context of execution (or compilation) so that when the original program called OPERATE, the prototype conditionally invokes COMPARE. A predicated class is one whose instances satisfy some predicate. If the behavior of OPERATE must be preserved, there might be more we need to do with this approach.
3. Introduce and invoke a generic function that continually waits for the original program to achieve a complex state:

```
(defgeneric watch-visit :method-combination :continuous)
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit (p:operate<tree:atomic-tree>) (compare tree))
```

Here the method combination type states that the generic function is invoked continually. The function WATCH-VISIT must be invoked on the process that traverses the trees:

```
(defun samefringe (t1 t2)
  ...
        (labels ((traverse (tree)
                   (operate tree)
                   (unless (atom tree)
                     (traverse (car tree))
                     (traverse (cdr tree)))))
          (complete
            (watch-visit (spawn p1 (traverse t1)))
            (watch-visit (spawn p2 (traverse t2))))))
```

By "continually" we mean that when WATCH-VISIT is applicable, the process that caused it to be applicable is paused and WATCH-VISIT is completed. Once WATCH-VISIT is completed, the previously paused process is continued until it would no longer be applicable. At that point WATCH-VISIT is re-invoked. That is, the order of events are as if a process, say P1, that is traversing a tree reaches OPERATE. The process P1 is paused and WATCH-VISIT runs. When WATCH-VISIT ends, P1 is continued until it exits OPERATE. At that point WATCH-VISIT is re-invoked (and P1 continues as well).
The odd syntax means this: WATCH-VISIT will be called when the process P is within OPERATE with a first argument of class ATOMIC-TREE. Note that the argument to OPERATE is available in the WATCH-VISIT method.
This could also have been written as follows, where applicability and pausing are as described above.

```
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit (p:operate<tree:atomic-tree>)
  (compare tree) (watch-visit p))
```

4. Introduce a generic function that will be triggered when OPERATE is invoked with a first argument of class ATOMIC-TREE:

```
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit:<tree:atomic-tree> :triggered () (compare tree))
```

where SAMEFRINGE looks like this:

```
(defun samefringe (t1 t2)
  ...
        (labels ((traverse (tree)
                   (operate tree)
                   (unless (atom tree)
                     (traverse (car tree))
                     (traverse (cdr tree)))))
          (watch-visit)
          (complete
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2)))))
```

We will discuss triggered functions more later.

6. Naming Variables

When dealing with the internal parts of one program from another, we need to be able to refer to the variables of one program from another. This requires a means to name variables to distinguish them from variables of the same name in other programs. We can implement program-specific variables by considering their names to be qualified by the process in which the program defining them is being executed. Similarly if the need arose to introduce new variables, the same mechanism could be used. Here's an example:

```
(class-let ((p1 (process)) (p2 (process)))
  (let ((n:p1 0)(n:p2 0))
    (flet ((count (l) (dolist (i l) (incf n))))
      (spawn p1 (count ...))
      (spawn p2 (count ...)) ...)))
```

7. Monotonic Variables

Another useful concept is called "monotonic variables." A monotonic variable is one whose value always increases or always decreases. Subclasses of monotonic variables are MONOTONIC:INCREASING and MONOTONIC:DECREASING.

Here is an interesting generic function:

```
(defmethod lesser (x:dead y:dead)
  (cond ((< x y) t)
        (t nil)))

(defmethod lesser (x:dead: y:monotonic:increasing)
  (cond ((< x y) t)
        (t (lesser x y))))

(defmethod lesser (x:monotonic:increasing y:dead)
  (cond ((<= y x) nil)
        (t (lesser x y))))
```

The idea is that if one of the numbers is dead and the other monotonic increasing, we can sometimes know which is smaller before both objects are dead. This can then be used to terminate processes at appropriate points. We can define similar methods for other combinations of DEAD, MONOTONIC:INCREASING, and MONOTONIC:DECREASING.

For example, here is a simple function to find the shorter of two lists:

```
(defun shorter (l1 l2)
  (class-let ((p1 (process)) (p2 (process)))
    (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
      (flet ((len (l)(dolist (x l) (incf n))))
        (spawn p1 (len l1))
        (spawn p2 (len l2))
        (cond ((lesser n:p1 n:p2) l1)
              (t l2))))))
```

For example, if L1 is shorter than L2, N:P1 will possibly be computed and rendered dead before P2 completes. If this happens, LESSER will terminate when N:P2 exceeds N:P1, which will exit the CLASS-LET defining P1 and P2, which will kill P2. The use of monotonic variables prevents SHORTER from running for a long time when the answer can be more quickly determined.

We can write a different version of SAMEFRINGE using monotonic variables:

```
(defun samefringe (t1 t2)
  (class-let ((p1 (process) ...)
              (p2 (process) ...))
    (pmi-pflet ((compare (x:p1 y:p2)
                  (unless (eq x y) (return-from samefringe nil))))
      (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
        (gf-flet ((count:compare:process :trigger () (incf n)))
          (labels ((traverse (tree)
                     (cond ((atom tree) (compare tree))
                           (t (traverse (car tree))
                              (traverse (cdr tree))))))
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2))
            (= n:p1 n:p2)))))))
```

Here, the function COUNT is a triggered function: When some process becomes an instance of an invocation of COMPARE, that process is paused while the body of COUNT is executed. That invocation of COUNT accretes the classes of the process that triggered it, so that INCF increases the right variable.

In the main body, SAMEFRINGE spawns the two processes and then returns the result of the equality test, which is implemented similarly to LESSER above.

8. Temporal or Historical Abstraction

Some of the problems in prototyping have to do with modifying a program to keep track of the history of a data structure or to answer questions about what things happened at what time. We believe there is a new type of abstraction, called ***temporal*** or ***historial abstraction*** that can make such changes simpler.

Consider a program that is simulating marriage and employment for the purposes of understanding how government policy decisions should be made about affirmative action. In such a simulation one could imagine a class being defined whose instances represent people. These instances might have some structure (a data abstraction) and a protocol.

One slot in the instance might store the gender of the individual and another might record whether the person is married. Part of the protocol is an initialization of an instance to store the correct gender, and another part is to assign the correct setting for the marriage slot during the abstract operations of marriage, divorce, and spousal death.

From the point of view of a reader of this simulation program, these two slots are identical except insofar as what is stored in them. Yet, from a real-world semantics point of view, they represent incommensurable things: The gender of an individual is a characteristic of the individual, and so it makes sense for the gender slot to be part of the representation of the individual. The marriage slot is a characteristic about the history of the individual, yet the marriage slot is not part of the representation of the history of the

individual. From a program semantics point of view the value of the marriage slot depends exactly on the history of the representation of the individual.

Therefore, the use of a data abstraction to represent whether an individual is married is incongruous regardless of which of the two most plausible ways you look at it.

If we later decide that the simulation needs to depend on whether a person was previously divorced, a new slot might be added to the class so that each individual records whether they have been previously married. There are many such historical questions we wish to ask of objects in the simulation: how many times was someone married, was this child born before or after the second divorce, was the salary of a person ever larger than the salary of the spouse.

The proliferation of data abstractions to implement other abstractions in this example leads one to wonder how many data abstractions are really implementations of different types of abstraction.

Even if the need to define a data abstraction to store historical information were removed, the problem of temporal abstractions would not be solved if the programmer still were required to insert protocol invocations at the points at which the history needed to be updated. This would defeat locality. Furthermore, it would reveal implementation.

Because a prototyper wishes to modify a program as little as possible, a temporal or historical abstraction mechanism would be welcome, if it were then possible to isolate them from the original code.

We believe that our class-based description mechanisms are a step in this direction. For example, a triggered function defines a temporal or historical abstraction, though in a primitive way. We wish to define a common pattern that captures the essence of counting the number of times that a process enters a particular state. Usually one would have to define some storage and then modify occurrences of the lexical manifestation of the target state to invoke code to update that storage to accomplish this, but just as one hopes to isolate the implementation of data abstractions from their use, so one hopes to isolate the implementation of the historical abstraction from its use.

In this case, its use is its definition, and its implementation is how the definition is turned into the proper bookkeeping code.

In some sense, this is just a highfalutin way of saying that with historical abstractions, we do not need to modify TRAVERSE to get SAMEFRINGE (in this case such modification is easy, but with more complex prototypes it might not be so easy):

```
(defun samefringe (t1 t2)
  (class-let ((p1 (process) ...)
              (p2 (process) ...))
    (pmi-pflet ((compare (x:p1 y:p2)
                  (unless (eq x y) (return-from samefringe nil))))
      (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
        (labels ((traverse (tree)
                   (cond ((atom tree) (incf n) (compare tree))
                         (t (traverse (car tree))
                            (traverse (cdr tree))))))
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2))
            (= n:p1 n:p2))))))
```

These two versions have several nice properties. First, each is pretty concise. Second, each matches well the description of the solution to the problem, repeated here:

> First we want to set up two processes, one to traverse T1 and the other to traverse T2. Then we set up a third process to look at the atoms coming from each traversal. If any two corresponding atoms are not the same, the processes are terminated and the answer is nil. Otherwise, if both traversal processes end normally, the two trees have the same fringe if both processes terminated after reporting the same number of atoms.

Some bad properties are the ugly, visible definitional forms: CLASS-LET, PMI-PFLET, LET, and GF-FLET, which can be hidden with annotations and abbreviations. The use of a visual layer on top of simpler text would eliminate some of this—remember that this is simply the textual representation of the program.

9. Conclusions

It's probably hard to tell what is really going on from this precis, but that's how research goes.