, , 1–16 ()

Continuation-Based Multiprocessing *

MITCHELL WAND **
Indiana University, Bloomington, IN 47405

Abstract. Any multiprocessing facility must include three features: elementary exclusion, data protection, and process saving. While elementary exclusion must rest on some hardware facility (*e.g.*, a test-and-set instruction), the other two requirements are fulfilled by features already present in applicative languages. Data protection may be obtained through the use of procedures (closures or funargs), and process saving may be obtained through the use of the `catch` operator. The use of `catch`, in particular, allows an elegant treatment of process saving.

We demonstrate these techniques by writing the kernel and some modules for a multiprocessing system. The kernel is very small. Many functions which one would normally expect to find inside the kernel are completely decentralized. We consider the implementation of other schedulers, interrupts, and the implications of these ideas for language design.

1. Introduction

In the past few years, researchers have made progress in understanding the mechanisms needed for a well-structured multi-processing facility. There seems to be universal agreement that the following three features are needed:

1. Elementary exclusion
2. Process saving
3. Data protection

By elementary exclusion, we mean some device to prevent processors from interfering with each other's access to shared resources. Typically, such an elementary exclusion may be programmed using a test and set instruction to create a critical region. Such critical regions, however, are not by themselves adequate to describe the kinds of sharing which one wants for controlling more complex resources such as disks or regions in highly structured data bases. In these cases, one uses an elementary exclusion to control access to a resource manager (*e.g.*, a monitor [11] or serializer [1]), which in turn regulates access to the resource.

Unfortunately, access to the manager may then become a system bottleneck. The standard way to alleviate this is to have the manager save the state of processes which it wishes to delay. The manager then acts by taking a request, considering the state of the resource, and either allowing the requesting program to continue

* Research reported herein was supported in part by the National Science Foundation under grant numbers MCS75-06678A01 and MCS79-04183. This paper originally appeared in J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, CA, 1980. The Lisp Company, Republished by ACM.
** Current address: College of Computer Science, Northeastern University, 360 Huntington Avenue, 161CN Boston, MA 02115, USA.

or delaying it on some queue. In this picture, the manager itself does very little computing, and so becomes less of a bottleneck.

To implement this kind of manager, one needs some kind of mechanism for saving the state of the process making a request.

The basic observation of this paper is that such a mechanism already exists in the literature of applicative languages: the `catch` operator [14], [15], [21], [25]. This operator allows us to write code for process-saving procedures with little or no fuss.

This leaves the third problem: protecting private data. It would do no good to write complex monitors if a user could bypass the manager and blithely get to the resource. The standard solution is to introduce a class mechanism to implement protected data. In an applicative language, data may be protected by making it local to a procedure (closure). This idea was exploited in [19], but has been unjustly neglected. We revive it and show how it gives an elegant solution to this problem.

We will demonstrate our solution by writing the kernel and some modules for a multiprocessing system. The kernel is very small. Many functions which one would normally expect to find inside the kernel, such as semaphore management [2], may be completely decentralized because of the use of `catch`. Our system thus answers one of the questions of [3] by providing a way to drastically decrease the size of the kernel. We have implemented the system presented here (in slightly different form) using the Indiana SCHEME 3.1 system [26].

The remainder of this paper proceeds as follows: In Section 2, we discuss our assumptions about the system under which our code will run. Sections 3 and 4 show how we implement classes and process-saving, respectively. In Section 5 we bring these ideas together to write the kernel of a multiprocessing system. In Sections 6 and 7 we utilize this kernel to write some scheduling modules for our system In Section 8, we show how to treat interrupts. Last, in Section 9, we consider the implications of this work for applicative languages.

2. The Model of Computation

Our fundamental model is that of a multiprocessor, multiprocess system using shared memory. That is, we have many segments of code, called *processes*, which reside in a single shared random-access memory. The extent to which processes actually share memory is to be controlled by software. We have several active units called *processors* which can execute processes. Several processors may be executing the same process simultaneously. We make the usual assumption that memory access marks the finest grain of interleaving; that is, two processors may not access (read or write) the sane word in memory at the same time. This elementary memory exclusion is enforced by the memory hardware.

At the interface between the processes and the processors is a distinguished process called the *kernel*. The kernel's job is to assign processes which are ready to run to processors which are idle. In a conventional system, *e.g.*, [22], this entails keeping track of many things. We shall see that the kernel need only keep track of ready processes.

It may be worthwhile to discuss the author's SCHEME 3.1 system, which provided the context for this work. SCHEME is an applicative-order, lexically-scoped, full-funarg dialect of LISP [25]. The SCHEME 3.1 system at Indiana University translates input Scheme code into the code for a suitable multistack machine. The machine is implemented in LISP. Thus, we were under the constraint that we could write no LISP code, since such an addition would constitute a modification to the machine. The system simulates a multiprocessor system by means of interrupts, using a protocol to be discussed in Section 8. However, all primitive operations, including the application of LISP functions, are uninterruptible. This allows us to write an uninterruptible test-and-set operation, such as

```
(de test-and-set-car (x)
  (prog2 nil (car x) (rplaca x nil)))
```

which returns the `car` of its argument and sets the `car` to `nil`.

Two other features of SCHEME are worth mentioning. First, SCHEME uses call-by-value to pass parameters. That means that after an actual parameter is evaluated, a new cons-cell is allocated, and in this new cell a pointer to the evaluated actual parameter is planted. (In the usual association list implementation, the pointer is in the cdr field; in the rib-cage implementation [24], it is in the car.) This pointer may be changed by the use of the `asetq` procedure. Thus, if we write

```
(define scheme-demo-1 (x)
  (block                              ; block = PROGN
    (asetq x 3)
    ((lambda (x) (asetq x 4)) x)
    x))
```

any call to `SCHEME-DEMO-1` always returns 3, since the second asetq changes a different cell from the first one. (This feature of applicative languages has always been rather obscure. See [17] Sec. 1.8.5 for an illuminating discussion.) The second property on which we depend is that the "stack" is actually allocated from the LISP heap using `cons` and is reclaimed using the garbage collector. This allows us to be quite free in our coding techniques. We shall have more to say about this assumption in our conclusions.

3. Implementing classes

For us, the primary purpose of the class construct is to provide a locus for the retention of private information. In Simula [6], a class instance is an activation record which can survive its caller. In an applicative language, such a record may be constructed in the environment (association list) of a closure (funarg). This idea is stated clearly in [23]; we discuss it briefly here for completeness.

For example, a simple cons-cell may be modelled by:

```
(define cons-cell (x y)
  (lambda (msg)
```

```
    (cond
      ((eq msg 'car) x)
      ((eq msg 'cdr) y)
      ((eq msg 'rplaca)
       (lambda (val) (asetq x val)))
      ((eq msg 'rplacd)
       (lambda (val) (asetq y val)))
      (t (error 'bad-msg)))))

(define car (x) (x 'car))

(define cdr (x) (x 'cdr))

(define rplaca (x v) ((x 'rplaca) v))

(define rplacd (x v) ((x 'rplacd) v))
```

Here a cons cell is a function which expects a single argument; depending on the type of argument received, the cell returns or changes either of its components. (We have arbitrarily chosen one of the several ways to do this). Such behaviorally defined data structures are discussed in [10], [20], [23]; at least one similar object was known to Church [5], cited in [24].

Another example, important for our purposes, is

```
(define busy-wait ()
  (let ((x (cons t nil)))
    (labels
      ((self (lambda (msg)
               (cond
                 ((eq msg 'P)
                  (if (test-and-set-car x)
                    t
                    (self 'P)))
                 ((eq msg 'V)
                  (car (rplaca x t)))
                 (t (error 'bad-msg))))))
      self)))
```

`busy-wait` is a function of no arguments, which, when called, creates a new locus of busy-waiting. It does this by creating a function with a new private variable `x`. (This `x` is guaranteed new because of the use of call-by-value). This returned function (here denoted `self`) expects a single argument, either `P` or `V`. Calling it with `P` sends it into a test-and-set loop, and calling it with `V` resets the `car` of `x` to `t`, thus releasing the semaphore. There is no way to access the variable `x` except through calls on this function. Note that we are not advocating busy-waiting (except perhaps in certain very special circumstances). Any use of `busy-wait` in the rest of this paper may be safely replaced with any hardware-supported elementary exclusion device which the reader may prefer. Our concern is how to build complex schedulers from these elementary exclusions. In particular, we shall consider better ways to build a semaphore in Section 6.

4. Process saving with `catch`

`catch` is an old addition to applicative languages. The oldest version known to the author is Landin's, who called it either "pp" (for "program point") [15] or "J-lambda" [14].[1] Reynolds [21] called it "escape." A somewhat restricted form of `catch` exists in LISP 1.5, as `errset` [16]; another version is found in MACLISP, as the pair `catch` and `throw`. The form we have adopted is Steele and Sussman's [25], which is similar to Reynolds'.

In SCHEME, `catch` is a binding operator. Evaluation of the expression `(catch id expr)` causes the identifier `id` to be bound (using call-by-value) to a "continuation object" which will be described shortly. The expression `expr` is then evaluated in this extended environment.

The continuation object is a function of one argument which, when invoked, returns control to the caller of the catch expression. Control then proceeds as if the `catch` expression had returned with the supplied argument as its value. This corresponds to the notion of an "expression continuation" in denotational semantics.

To understand the use of `catch`, we may consider some examples.

```
(catch m (cons (m 3) 'a))
```

returns 3; when the `(m 3)` is evaluated, it is as if the entire `catch` expression returned 3. The form in which we usually will use `catch` is similar. In

```
(define foo (x) (catch m --body-- ))
```

evaluation of `(m -junk-)` causes the function `foo` to return to its caller with the value of `-junk-`.

The power of `catch` arises when we *store* the value of `m` and invoke it from some other point in the program. In that case, the caller of `foo` is restarted with `m`'s argument. The portion of the program which called `m` is lost, unless it has been preserved with a strategically placed `catch`. A small instance of this phenomenon happened even in our first example—there, `m`'s caller was the `cons` which was abandoned. Calling a continuation function is thus much like jumping into hyperspace—one loses track entirely of one's current context, only to re-emerge in the context that set the continuation.

There will actually be very few occurrences of `catch` in the code we write. For the remainder of this section and the next, we shall consider what things we can do with continuations which have already been created by `catch`. When we get to Section 6, we shall start to use `catch` in our code.

We shall use continuations to represent processes. A process is a self-contained computation. We may represent a process as a pair consisting of a continuation and an argument to be sent to that continuation. This corresponds to the notion of "command continuation" in denotational semantics.

```
(define cons-process (cont arg)
  (lambda (msg)
    (cond
      ((eq msg 'run-it) (cont arg))
      (t (error 'bad-msg)))))
```

Here we have defined a process as a class instance with two components, a continuation and an argument, and a single operation, `run-it`, which causes the continuation to be applied to the argument, thus starting the process. Because `cont` is a continuation, applying it causes control to revert to the place to which it refers, and the caller of `(x 'run-it)` is lost. This is not so terrible, since the caller of `(x 'run-it)` may have been saved as a continuation someplace else.

5. The Kernel

We now have enough machinery to write the kernel of our operating system. The kernel's job is to keep track of those processes which are ready to run, and to assign a process to any processor which asks for one. The kernel is therefore a class instance which keeps a queue of processes and has two operations: one to add a process to the ready queue and one to assign a process to a processor (thereby deleting it from the ready queue).

We shall need to do some queue manipulation. We therefore assume that we have a function `(create-queue)` which creates an empty queue, a function `(addq q x)` which has the side effect of adding the value of `x` to the queue `q`, and the function `(deleteq q)`, which returns the top element of the queue `q`, with the side-effect of deleting it from `q`.

We may now write the code for the kernel:

```
(define gen-kernel ()
  (let ((ready-queue (create-queue))
        (mutex (busy-wait)))
    (lambda (msg)
      (cond
        ((eq msg 'make-ready)
         (lambda (cont arg)
           (block
             (mutex 'P)
             (addq ready-queue (cons-process cont arg))
             (mutex 'V))))
        ((eq msg 'dispatch)
         (mutex 'P)
         (let ((next-process (deleteq ready-queue)))
           (block
             (mutex 'V)
             (next-process 'run-it))))))))

(asetq kernel (gen-kernel))
```

```
(define made-ready (cont arg)
  ((kernel 'make-ready) cont arg))

(define dispatch () (kernel 'dispatch))
```

We have now defined the two basic functions, `make-ready` and `dispatch`. The call `(make-ready cont arg)` puts a process, built from `cont` and `arg`, on the ready queue. To do this, it must get past a short busy-wait. (This busy-wait is always short because the kernel is never tied up for very long. This construction is also in keeping with the idea of building complex exclusion mechanisms from very simple ones.) It then puts the process on the queue, releases the kernel's exclusion, and exits. (Given the code for busy-wait above, it always returns `t`. The value returned must not be a pointer to any private data).

`dispatch` is subtler. A processor will execute `(dispatch)` whenever it decides it has nothing better to do. Normally, a call to `dispatch` would be preceded by a call to `make-ready`, but this need not be the case. After passing through the semaphore, the next waiting process is deleted from the ready queue and assigned to `next-process`. A `(mutex 'V)` is executed, and the `next-process` is started by sending it a `run-it` signal.

The subtlety is in the order of these last two operations. They cannot be reversed, since once `next-process` is started, there would be no way to reset the semaphore. The given order is safe however, because of the use of call-by-value. Every call on `(dispatch)` uses a different memory word for `next-process`. Therefore, the call `(next-process 'run-it)` uses no shared data and may be executed outside the critical region.

(A few explanatory words on the code itself are in order. First, note that `(kernel 'make-ready)` returns a function which takes two arguments and performs the required actions. `(kernel 'dispatch)`, however, performs its actions directly. We could have made `(kernel 'dispatch)` return a function of no arguments, but we judged that to be more confusing than the asymmetry. Second, `block` is SCHEME's sequencing construct, analogous to `progn`. Also, `cond` uses the so-called "generalized cond," with an implicit `block` (or `progn`) on the right-hand-side of each alternative.)

6. Two Better Semaphores

Our function `busy-wait` would be an adequate implementation of a binary semaphore if one was sure that the semaphore was never closed for very long. In this section, we shall write code for two better implementations of semaphores.

For our first implementation, we use the kernel to provide an alternative to the test-and-set loop. If the test-and-set fails, we throw the remainder of the current process on the ready queue, and execute a `DISPATCH`. This is sometimes called a "spin lock."

```
(define spin-lock-semaphore ()
  (let ((x (cons t nil)))
    (labels
```

```
      ((self (lambda (msg)
               (cond
                 ((eq msg 'P)
                  (cond
                    ((test-and-set-car x) t)
                    (t
                      (give-up-and-try-later)
                      (self 'P))))
                 ((eq msg 'V)
                  (car (rplaca x t)))
                 (t (error 'bad-msg))))))
      self)))

(define give-up-and-try-later ()
  (catch caller
    (block
      (make-ready caller t)
      (dispatch) )))
```

Here, the key function is `give-up-and-try-later`. It puts on the ready-queue a process consisting of its caller and the argument `t`. It then calls `dispatch`, which switches the processor executing it to some ready process. When the enqueued process is restarted (by some processor executing a `dispatch`), it will appear that `give-up-and-try-later` has quietly returned `t`. The effect is to execute a delay of unknown duration, depending on the state of the ready queue. Thus a process executing a `P` on this semaphore will knock on the test-and-set cell once; if it is closed, the process will go to sleep for a while and try again later.

While this example illustrates the use of `catch` and `make-ready`, it is probably not a very good implementation of a semaphore. A better implementation (closer to the standard one) would maintain a queue of processes waiting on each semaphore. A process which needs to be delayed when it tries a `P` will be stored on this queue. When a `V` is executed, a waiting process may be restarted, or, more precisely, placed on the ready queue. We code this as follows:

```
(define semaphore ()
  (let ((q (create-queue))              ; a queue for waiting processes
        (count 1)                       ; either 0 or 1
        (mutex (busy-wait)))
    (lambda (msg)
      (cond
        ((eq msg 'P)
         (mutex 'P)
         (catch caller
           (block
             (cond
               ((greaterp count 0)
                (asetq count (sub1 count))
                (mutex 'V)
                t)
```

```
            (t (addq q caller)
               (mutex 'V)
               (dispatch))))))
     ((eq msg 'V)
      (mutex 'P)
      (if (emptyq q)
        (asetq count (add1 count))
        (make-ready (deleteq q) t))
      (mutex 'V)
      t)))))
```

Executing `(semaphore)` creates a class instance with a queue `q`, used to hold processes waiting on this semaphore, an integer `count`, which is the traditional "value" of the semaphore, and a busy-wait locus `mutex`. `mutex` is used to control access to the scheduling code, and is always reopened after a process passes through the semaphore. As was suggested in the introduction, this use of a small busy-wait to control entrance to a more sophisticated scheduler is typical.

When a `P` is executed, the calling process first must get past `mutex` into the critical region. In the critical region, the count is checked. If it is greater than 0, it is decremented, the `mutex` exclusion is released, and the semaphore returns a value of `T` to its caller. If the count is zero, the continuation corresponding to the caller of the semaphore is stored on the queue. `mutex` is released, and the processor executes a `(dispatch)` to find some other process to work on.

When a `V` is executed, the calling process first gets past `mutex` into the semaphore's critical region. The queue is checked to see if there are any processes waiting on this semaphore. If there are none, the count is incremented. If there is at least one, it is deleted from the queue by `(deleteq q)`, and put on the kernel's ready queue with argument `t`. When it is restarted by the kernel, it will think it has just completed its call on `P`. (Since a `P` always returns `t`, the second argument to `make-ready` must likewise be a `t`). After this bookkeeping is accomplished, `mutex` is released and the call on `V` returns `t`.

All of this is just what a typical implementation of semaphores (*e.g.*, [2]) does. The difference is that our semaphore is an independent object which lies outside the kernel. It is in no way privileged code.

We have also written code to implement more complex schedulers. The most complex scheduler for which we have actually written code is for Brinch Hansen's "process" [4]. We have written this as a SCHEME syntactic macro. The code is only about a page long.

7. Doing more than one thing at once

We now turn to the important issue of process creation. Although the semaphores in the previous section used `catch` to save the state of the current process, they did not provide any means to increase the number of processes in the system. We may do this with the function `create-process`. `create-process` takes one argument, which is a function of no arguments, and creates a process which will execute this function in "parallel" with the caller of `create-process`.

```
(define create-process (fn)
  (catch caller
    (block
      (catch process
        (block
          (make-ready process t)
          (caller t)))
      (fn)
      (dispatch))))
```

When `create-process` is called with `fn`, it first creates a continuation containing its caller and calls it `caller`. It enters the `block`, and creates a continuation called `process`, which, when started, will continue execution of the `block` with `(fn)`. This continuation `process` is then put on the kernel's ready queue (with argument `t`, which will be ignored when `process` is restarted). Then `(caller t)` is executed, which causes `create-process` to return to its caller with value `t`.

Thus, the process which called `create-process` continues in control of its processor, but `process` is put onto the ready queue. When the kernel decides to run `process`, `(fn)` will be executed. The processor which runs `process` will then do a `(dispatch)` to find something else to do.

(The reader who finds this code tricky may take some comfort in our opinion that this is the trickiest piece of code in this paper. The difficulty lies in the fact that its execution sequence is almost exactly reversed from its lexical sequence [8].)

We can use `create-process` to implement a fork-join. The function `fork` takes two functions of no arguments. Its result is to be the cons of their values. The execution of the two functions is to proceed as two independent processes, and the process which called `fork` is to be delayed until they both return.

```
(define fork (fn1 fn2)
  (catch caller
    (let ((one-done? nil)
          (ans1 nil)
          (ans2 nil)
          (mutex (busy-wait)))
      (let ((check-done
              (lambda (dummy)
                (block
                  (mutex 'P)
                  (if one-done?
                    (make-ready caller
                      (cons ans1 ans2))
                    (asetq one-done? t))
                  (mutex 'V) ))))
        (block
          (create-process
```

```
    (lambda ()
      (check-done (asetq ans1 (fn1)))))
  (create-process
    (lambda ()
      (check-done (asetq ans2 (fn2)))))
  (dispatch))))))
```

`fork` sets up four locals: one for each of the two answers, a flag called `one-done?`, and a semaphore to control access to the flag. It creates the two daughter processes and then dispatches, having saved its caller in the continuation `caller`. Each of the two processes computes its answer, deposits it in the appropriate local variable, and calls `check-done`. `check-done` uses `mutex` to obtain access to the flag `one-done?`, which is initially `nil`. If its value is `nil`, then it is set to `T`. If its value is `T`, signifying that the current call to `check-done` is the second one, then `caller` is moved to the ready queue with argument `(cons ans1 ans2)`.

8. Interrupts

What we have written so far is quite adequate for a non-preemptive scheduling system [2]. If we wish to use a pre-emptive scheduling system (as we must if we wish to use a single processor), then we must consider the handling of interrupts.

We shall consider only the problem of pre-emption of processes through timing interrupts as non-preempting interrupts can be handled through methods analogous to those in [3], [27].

We model a timing interrupt as follows: When a processor detects a timing interrupt, the next identifier encountered in the course of its computation (say `X`) will be executed as if it had been replaced by `(preempt X)`. `preempt` is the name of the interrupt-handling routine. If we believe, with [23], that a function application is just a `GO-TO` with binding, then this model is quite close to the conventional model, in which an interrupt causes control to pass to a predefined value of the program counter. A very similar treatment of interrupts was developed independently for use in the MIT/Xerox PARC SCHEME chip [12].

The simplest interrupt handler is:

```
(define preempt (x)
  (catch caller
    (block
      (make-ready caller x)
      (dispatch))))
```

With this interrupt handler, the process which the processor is executing is thrown back on the ready queue, and the processor executes a `dispatch` to find something else to do.

A complication that arises with pre-emptive scheduling is that interrupts must be inhibited inside the kernel. This may be accomplished by changing the `busy-wait` in the kernel to `kernel-exclusion`:

```
(define kernel-exclusion ()
  (let ((sem (busy-wait)))
    (lambda (msg)
      (cond
        ((eq msg 'P)
         (sem 'P)
         (disable-preemption))
        ((eq msg 'V)
         (sem 'V)
         (enable-preemption))))))
```

Note the order of the operations for `V`. The reverse order is wrong; an interrupt might occur after the `enable-preemption` but before the `(sem 'V)`, causing instant deadlock. (We discovered this the hard way!)

Now, for the first time, we have introduced some operations which probably should be privileged: `disable-preemption` and `enable-preemption`.[2] We can make those privileged without changing the architecture of the machine by introducing a read-loop like:

```
(define user-read-loop ()
  (let ((disable-preemption
          (lambda () (error 'protection-error)))
        (enable-preemption
          (lambda () (error 'protection-error))))
    (labels
      ((loop
         (lambda (dummy)
           (loop (print (eval (read)))))))
      (loop nil))))
```

This is intended to suggest the user's input is evaluated in an environment in which `disable-preemption` and `enable-preemption` are bound to error-creating functions. This is not actually the way the code is written in SCHEME, but we have written it in this way to avoid dealing with the complications of SCHEME's version of `eval`.

9. Conclusions and Issues

In this paper, we have shown how many of the most troublesome portions of the "back end" of operating systems may be written simply using an applicative language with `catch`. In the course of doing so, we have drawn some conclusions In three categories: operating system kernel design, applicative languages, and language design in general.

For operating systems, this work answers in part Brinch Hansen's call to simplify the kernel [3]. Because all of the scheduling apparatus except the ready queue has been moved out of the kernel, the kernel becomes smaller, is called less often, and therefore becomes less of a bottleneck. By passing messages to class instances (functions) instead of passing them between processes, we avoid the need for individuation of processes, and thereby avoid the need to maintain process tables, etc., further reducing the size of the kernel.

This is not meant to imply that we have solved all the problems associated with system kernels. Problems of storage allocation and performance are not addressed. In the areas of process saving and protection, however, the approach discussed here seems to offer considerable advantages.

In the area of applicative languages, our work seems to address the issue of "state." A module is said to have "state" if different calls on that module with identical arguments may give different results at different times in the computation. Another way of describing this phenomenon is that the model is "history-dependent." (This is not to be confused with issues of non-determinism). If an object does not have state, then it should never matter whether two processes are dealing with the same object or with two copies of it. For processes to communicate, however, they must be talking to the same module, not just to two copies of it. For instance, all modules must communicate with the same kernel, not just with two or more modules produced by calls on `gen-kernel`. Therefore, the kernel and similar modules must have state—they must have uses of `asetq` in their code.

This seems to us to be an important observation. It means that we must come to grips with the concept of the state if we are to deal with the semantics of parallelism. This observation could not have been made in the context of imperative languages, where every module has state. Only in an applicative context, where we can distinguish true state from binding (or internal state), could we make this distinction.[3]

A related issue is the use of call-by-value. A detailed semantics of SCHEME, incorporating the Algol call-by-value mechanism, would give an unambiguous account of when two modules were the "same," and thus also give an account of when two modules share the same state. Such an account is necessary to explain the use of `asetq` in our programs and to determine which data is private and which is shared (as in the last lines of `(kernel 'dispatch)`). In such a description, we would find that restarting a continuation restores the environment (which is a map from identifiers to L-values), but does not undo changes in the global state (the map from L-values to R-values) which is altered by `asetq`. Nonetheless, we find this account unsatisfying, because its systematic introduction of a global state at every procedure call seems quite at odds with the usual state-free picture of an applicative program. We find it unpleasant to say that we pass parameters by worth (*i.e.*, without copying), except when we need to think harder about the program.

In this regard, we commend to applicative meta-programmers a closer study of denotational semantics. Descriptive denotational semantics, as expounded in Chapter 1 of [17] or in [9], provides the tools to give an accurate description of what actually happens when a parameter is passed. There are, however, some measures which would help alleviate the confusion. For example, we could use a primitive `cell` operation in place of the unrestricted use of `asetq`. Then all values could be passed by worth (R-value); L-values would arise only as denotations of cells, and explicit dereferencing would be required. Such an approach is taken, in various degrees, in PLASMA [10], FORTH [13], and BLISS [28].[4] Also, John Reynolds and one of his students are investigating semantics which do not rely on a single global state [personal communication].[5]

Last, we essay some ideas about the language design process. Our choice to work in the area of applicative languages was motivated in part by Minsky's call for the separation of syntax from semantics in programming [18]. We have attempted to home in on the essential *semantic* ideas in multiprogramming. By "semantic" we do not simply mean those ideas which are expressible in denotational semantics, though surely the use of denotational semantics has exposed and simplified the basic ideas in programming in general. We add to these ideas some basic operational knowledge about how one goes from semantics to implementations (*e.g.*, [21]) and some additional operational knowledge not expressed in the "formal semantics" at all, *e.g.*, our treatment of interrupts.

Only after we have a firm grasp on these informal semantic ideas should we begin to consider syntax. Some syntax is for human engineering—replacing parentheses and positional structure with grammars and keywords. Other syntax may be introduced to restrict the class of run-time structures which are needed to support the language. The design of RUSSELL [7] is a good example of this paradigm. One spectacular success which may be claimed for this approach is that of PASCAL, which took the well-understood semantics of ALGOL and introduced syntactic restrictions which considerably simplified the run-time structure.

In our case, we should consider syntactic restrictions which will allow the use of sequential structures to avoid spending all one's time garbage-collecting the stack. Other clever data structures for the run-time stack should also be considered. Another syntactic restriction which might be desirable is one which would prevent a continuation from being restarted more than once.

Any language or language proposal must embody a trade-off between generality (sometimes called "functionality") and efficiency. By considering complete generality first, we may more readily see where the trade-offs may occur, and what is lost thereby. Unfortunately, the more typical approach to language design is to start with a given run-time structure (or, worse yet, a syntactic proposal). When the authors realize that some functionality is lacking, they add it by introducing a patch. By introducing the generality and cleanness first, and then compromising for efficiency, one seems more likely to produce clean, small, understandable, and even efficient languages.

Notes

1. Though `catch` and `call/cc` are clearly interdefinable, J and `call/cc` differ importantly in details; see Hayo Thielecke, "An Introduction to Landin's 'A Generalization of Jumps and Labels'," *Higher-Order and Symbolic Computation*, 11(2), pp. 117–123, December 1998.
2. These were additional primitives that were added to the Scheme 3.1 interpreter.

3. This paragraph grew out of conversations I had had with Carl Hewitt over the nature of object identity. I had objected that Hewitt's notion of object identity in a distributed system required some notion of global state (C. Hewitt and H. G. Baker, Actors and Continuous Functionals, in E. J. Neuhold (ed.) *Formal Descriptions of Programming Concepts*, pages 367–390. North Holland, Amsterdam, 1978; at page 388). This is an issue that remains of interest in the generation of globally-unique identifiers for use in large distributed systems such as IP, DCOM or the World-Wide Web.
4. This approach was of course adopted in ML. At the time, changing Scheme in this way was at least conceivable, and we seriously considered it for the Indiana Scheme 84 implementation. After the Revised^3 Report in 1984, such a radical change became impossible. Sussman and Steele now list this as among the mistakes in the design of Scheme (G.J. Sussman and G.L. Steele Jr., The First Report on Scheme Revisited, *Higher-Order and Symbolic Computation* 11(2), pp. 399–404, December, 1998).
5. I am not sure to what this refers. My best guess is that it refers to his work with Oles on stack semantics (J. C. Reynolds, "The Essence of Algol," in J. W. deBakker and J. C. van Vliet, eds., *Algorithmic Languages*, pages 345–372. North Holland, Amsterdam, 1981).

References

1. R. Atkinson and C. Hewitt. Synchronization in actor systems. In *Conf. Rec. 4th ACM Symp. on Principles of Programming Languages*, pages 267–280, 1977.
2. P. Brinch Hansen. *Operating Systems Principles.* Prentice-Hall (Englewood Cliffs, NJ), 1973.
3. P. Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall (Englewood Cliffs, NJ), 1977.
4. P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Comm. ACM*, 21:934–941, 1978.
5. A. Church. *The Calculi of Lambda-Conversion.* Annals of Mathematics Studies. Princeton University Press (Princeton, NJ), 1941.
6. O-J. Dahl and C.A.R. Hoare. Hierarchical program structures. In O-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, pages 175–220. Academic Press (London), 1972.
7. A.J. Demers and J.E. Donahue. Data types, parameters, and type checking. In *Conf. Rec. 7th Ann. ACM Symp. on Principles of Programming Languages*, pages 12–23, 1980.
8. E.W. Dijkstra. Go to statement considered harmful. *Comm. ACM*, 11:147–148, 1968.
9. M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer, 1979.
10. C.E. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.
11. C.A.R. Hoare. Monitors: An operating system structuring concept. *Comm. ACM*, 17:549–557, 1974.
12. J. Holloway, G. L. Steele, G.J. Sussman, and A. Bell. The SCHEME-79 chip. AI Memo 559, MIT Artificial Intelligence Laboratory, Dec. 1979.
13. J.S. James. FORTH for microcomputers. *SIGPLAN Notices*, 13(10):33–39, October 1978.
14. P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Comm. ACM*, 8:89–101, 1965.
15. P.J. Landin. The next 700 programming languages. *Comm. ACM*, 9:157–166, 1966.
16. J. McCarthy et al. *LISP 1.5 Programmer's Manual.* MIT Press (Cambridge, MA), 1965.
17. R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall (London) and Wiley (New York), 1976.
18. M. Minsky. Form and content in computer science. *J. ACM*, 17:197–215, 1970.
19. J.H. Morris. Protection in programming languages. *Comm. ACM*, 16:15–21, 1973.
20. J.C. Reynolds. GEDANKEN – a simple typeless language based on the principle of completeness and the reference concept. *Comm. ACM*, 13:308–319, 1970.
21. J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM National Conf.*, pages 717–740, 1972.
22. A.C. Shaw. *The Logical Design of Operating Systems.* Prentice-Hall (Englewood Cliffs, NJ), 1974.

23. G.L. Steele. LAMBDA: The ultimate declarative. AI Memo 379, MIT Artificial Intelligence Laboratory, October 1976.
24. G.L. Steele and G.J. Sussman. The art of the interpreter, or the modularity complex. AI Memo 453, MIT Artificial Intelligence Laboratory, May 1978.
25. G.L. Steele and G.J. Sussman. The revised report on SCHEME. AI Memo 452, MIT Artificial Intelligence Laboratory, January 1978.
26. M. Wand. SCHEME version 3.1 reference manual. Technical Report Technical Report No. 93, Indiana University Computer Science Department, June 1980.
27. N. Wirth. Modula: a language for modular multiprogramming. *Software-Practice and Experience*, 7:3–35, 1977.
28. W.A. Wulf, D.B. Russell, and A.N. Habermann. BLISS: A language for systems programming. *Comm. ACM*, 14:780–790, 1971.