MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001: Structure and Interpretation of Computer Programs
Fall Semester 1989

Notes on Parallel Computation

November 28, 1989

1 Introduction

These notes are a supplement to the textbook: "Structure and Interpretation of Computer Programs" (Abelson and Sussman, MIT Press, 1985). The textbook presents sequential computing models exclusively. These notes generalize the Scheme language and its interpretation to include parallel models. The material here is based heavily on the textbook— in particular, the "explicit-control evaluator" of Section 5.2.

Why is parallelism important? There are many computational problems in diverse fields that are infeasible on the fastest computers available today because the machines are simply too slow. Examples include: weather prediction, simulating the aerodynamics of a new automobile, seismic prediction, high-speed bank-card transaction processing, etc. In the past, computers have become faster because of improved circuit technology (raw speed and miniaturization) and architectural and compiler-based innovations (pipelining, vector processing, RISC). However, such advances are unlikely to produce improvements in the required orders-of-magnitude range. The most promising solution is parallelism, i.e., instead of speeding up a single machine, replicate it, and get the ensemble to *cooperate* in solving a problem by solving pieces of it.

There is very little consensus about the architectures and languages needed to achieve this goal— it is a rich area of research. Here, we will study the topic at a very general and abstract level. We will study some parallel versions of Scheme and some parallel evaluators that are based on the "Explicit-Control Evaluator" of the textbook (Section 5.2).

2 Sources of parallelism

Parallelism in programs can be viewed at two levels:

- At the *algorithmic* level, i.e., independent of machines, and
- At the *operational* level, i.e., the way an interpreter exploits the parallelism inherent in an algorithm.

Typically, we first try to choose an algorithm that has an intrinsically high degree of parallelism, and then we try to execute it in such a way that none of this parallelism is wasted.

2.1 Algorithmic sources of parallelism in programs

Consider the following two algorithms for computing the factorial of a number n:

Algorithm 1:

- If $n = 1$, the answer is 1.
- If $n > 1$, the answer is $n \times \mathit{factorial}(n-1)$.

Algorithm 2:

The answer is $\mathit{product}(1, n)$, where the algorithm to compute $\mathit{product}(l, m)$ is:

- If $l = m$, the answer is m.
- If $l + 1 = m$, the answer is $l \times m$.
- If $l + 1 < m$, the answer is $\mathit{product}(l, j) \times \mathit{product}(j + 1, m)$, where $j = \lfloor (l + m)/2 \rfloor$.

In Algorithm 1, we set up a linear chain of n multiplications that must be done one after the other, so that it must take $O(n)$ time.

In Algorithm 2, we use a divide-and-conquer strategy. To find the factorial of 8, for example, we independently find the product of 1 through 4 and the product of 5 through 8, and then multiply them together. And, recursively, to find the product of 1 through 4, we independently find the product of 1 through 2 and the product of 3 through 4, and then multiply them together. Thus, we set up a *tree* of multiplications instead of a linear chain, and it is possible to compute the factorial in $O(\log n)$ time.

The lesson: for the same mathematical function, the choice of algorithm can have a dramatic effect on available parallelism.

Of course, the amount of parallelism also depends on the data. In Algorithm 2, for example, the amount of parallelism depends on the value of n. However, the choice of data is not always under our control.

The holy grail of parallel processing is "linear speedup", i.e., double the size of the machine (number of processors and memories), to get double the speed. Unfortunately, this is, in general, unachievable. First, as described above, the structure of the program itself will always place a theoretical limit on how much parallelism is available to be exploited. Second, a larger machine must be physically larger, and it will always take finite time to transport data from one part of a machine to another, and this time spent is an overhead that we can try to minimize, but never eliminate. Finally, there are serious difficulties in marshalling and managing many parallel activities— as anyone who has ever tried to manage a large and unruly organization will appreciate, it is very difficult to keep all processors busy doing useful work all the time!

2.2 Operational sources of parallelism in programs

Once we have chosen a particular algorithm, there are still many sources of parallelism in our choice of execution mechanisms. The particular source that we will examine is the *parallel evaluation of sub-expressions in a combination* .

Consider Algorithm 2 expressed in Scheme:

```
(define (factorial n)
  (define (product l m)
     (cond
      ((= l m) m)
      ((= (+ l 1) m) (* l m))
      (else (let ((j (middle l m)))
              (* (product l j)
                 (product (+ j 1) m))))))
  (product 1 n))
```

When run on a conventional Scheme interpreter, it will still take $O(n)$ time because the implementation picks a particular sequential order in which to compute the arguments of a combination, so that the two recursive products are computed sequentially. However an unconventional, parallel Scheme interpreter may evaluate them in parallel, thus finishing in $O(\log n)$ time. In other words, an *implementation* may or may not exploit the parallelism available in a program. In these notes, we shall study two unconventional, parallel Scheme implementations.

3 Parallelism and side effects: a dangerous mix!

Even in sequential programs, we have seen that side-effects can complicate the clarity of a program. In parallel programs, things get much worse. For example, suppose we wanted to write an "instrumented" version of a function that, in addition to doing its usual thing, also recorded every argument it was called with. Here is an instrumented "identity" function, and some uses of it:

```
==> (define L '())
 ()

==> (define id (lambda (x)
       (set! L (cons x L))
       x))
 F

==> (+ (id 10) (id 20))
 30

==> L
 ?
```

What answer should we get on the last line? In sequential Scheme, since the order of evaluation of arguments in a combination is not specified, we do not know which of the expressions `(id 10)` and `(id 20)` will be evaluated first. Thus, we could get either of these two answers:

(10 20) or *(20 10)*

This program is said to be *indeterminate*, because the same program, when run on two different sequential Scheme implementations, may produce different results.[1]

In parallel Scheme this indeterminacy is possible even on a single implementation because different schedules may be followed on different runs of the same program. In fact, it gets worse, because it is now possible to get results that were impossible in any sequential implementation. Let us focus on the expression:

```
(set! L (cons x L))))
```

Recall the evaluation rules for `set!`. The schedule of events is:

...
Evaluate `(cons x L)`
Update binding for `L`
...

Now, since there are two concurrent evaluations of `id`, it is possible that their activities get interleaved as follows:

Activity for `(id 10)`	Activity for `(id 20)`
...	
Evaluate `(cons 10 L)`	...
	Evaluate `(cons 20 L)`
Update binding for `L`	...
...	*Update binding for* `L`

Here, each "*evaluate*" step finds the value of `L` to be `'()`. The `cons`'es thus produce `(10)` and `(20)`, respectively. Assuming the *update* activities go in the order shown, the final result is

(20)

Similarly, we could also have got the result: *(10). Neither result is possible in a sequential implementation!*

This situation is called a *race*, i.e., one activity races to perform an update before another activity reads it. For correct execution of the above program, we want the *evaluate* and *update binding* parts of a `set!` to execute *atomically*, i.e., as one, indivisible event. We merely alert the reader to this subtlety at this point; we will return to this issue later in Section **??**. For the moment, we will temporarily banish side-effects from our language— no `set!`, `set-car!` or `set-cdr!`.

4 From individual to multiple processes

4.1 The state of a process

We are going to extend the sequential explicit-control evaluator described in the textbook. That evaluator can be regarded as implementing a single *process*. The state of a process is captured by four things:

[1]Some indeterminacy is in the eye of the beholder. If `L` is regarded as an *unordered set*, then *(10 20)* and *(20 10)* are "the same", and the program is determinate.

- a code sequence, and
- a set of seven registers (`exp`, `env`, `val`, `fun`, `unev`, `argl` and `continue`);
- a stack;
- a memory (also called a *heap*) containing objects such as cons-cells and procedure objects.

The registers may refer to objects in memory. At each step, we execute the first instruction in the code sequence, resulting in a new state for the process.[2]

In order to generalize this to multiple processes, we are going to replicate the first three components of the state for each process, i.e., each process has its own code sequence, registers and stack. All processes share a single heap memory.[3]

In fact, to simplify matters, we are going to represent the code sequence and the stack also using registers. We introduce two new registers: `pc` (for "program-counter") to contain the code sequence, and `stack` to contain the stack. The stack can be regarded merely as a list of items. So that the stack itself can also be stored in the heap memory.

Thus, each *process* is just a set of nine registers.

4.2 Processes *vs.* Processors

The number of processes for a computation can be arbitrarily large, depending on the parallelism available in the computation. However, any real machine will only have a fixed number of *processors*. In practice, therefore, we need some mechanism to multiplex the given processors over the given processes. The details are beyond the scope of these notes—here, we will only look at the process abstraction.

4.3 Creating a process

Every program begins exection as a single, "primordial" process. As it executes, it may create more processes, each of which, in turn, may create still more processes, etc. We say that one process can "fork" a new process.

4.3.1 The primordial process

The primordial process is initialized as follows:

- `Exp` register: the top-level expression e.
- `Env` register: the global environment.
- `Stack` register: the empty stack.
- `Pc` register: the first instruction in the controller code.

The start of the controller code looks like this:

[2]In the literature, one can also find the terms *thread*, *task*, *instruction-stream*, etc. to describe processes.

[3]Thus, our model of the machine is a so-called "shared memory" model. We emphasize that this is only an abstract view. Do not be misled into thinking that this must be implemented as a single physical memory module.

```
    (assign continue RECORD-RESULT)
    (goto EVAL-DISPATCH)

  RECORD-RESULT
    (record-value (fetch val))
    (stop)
```

i.e., it will evaluate the expression, record the result, and stop.

4.3.2 Forking a process

A process P may fork a new process Q by executing the following instruction:

(fork *label new-stack*)

A new set of nine registers for Q are created and initialized thus:

- `Pc` register: code sequence beginning at *label*.
- `Stack` register: *new-stack*

We often refer to P as the "parent" process and Q as its "child".

For example, suppose P wants to fork Q with the following mission: Q should evaluate the expression `(f x)` in the environment which is currently in P's `env` register, print the result and stop. P executes the following code:

```
    (assign val (cons (fetch env) '()))
    (assign val (cons '(f x) (fetch val)))
    (fork EVAL-PRINT-AND-DIE (fetch val))
```

P first builds Q's initial stack in its own `val` register, containing the expression `(f x)` and the environment. It then forks Q which will begin operation at label `EVAL-PRINT-AND-DIE`, with that stack. Then, P can go on to do other things (beyond the `fork` instruction).

Meanwhile, Q begins execution as follows:

```
  EVAL-PRINT-AND-DIE
    (restore exp)
    (restore env)
    (assign continue PRINT-AND-DIE)
    (goto EVAL-DISPATCH)

  PRINT-AND-DIE
    (perform (print (fetch val)))
    (stop)
```

i.e., it sets up the contract for `EVAL-DISPATCH` by setting up the `exp`, `env` and `continue` registers and going there. Thus, when `EVAL-DISPATCH` has done its thing, Q reaches label `PRINT-AND-DIE` with the value in the `val` register. Here, it prints the value and executes the `stop` instruction that kills the process.

4.4 Terminating a process and the `join` instruction

One way for a process to terminate is simply to execute a `stop` instruction. Another way is to use a new instruction called `join`.

Let *cell* be a cons-cell in memory whose CAR is initialized to an integer N. Now suppose a process executes:

```
(join cell)
```

The effect of this instruction is first to decrement the integer in the CAR of the cell, i.e., read it out, decrement it, and store it back using SET-CAR!. Then, if the value is now zero, the process continues at the next instruction— otherwise, it dies.

We call the *cell* containing N a "join counter".

We can use the `join` instruction to bring together N processes into a single process. We create a new cell containing N, and then fork off N processes, passing the cell to each one via its stack. Each process, after doing its work, executes a `join` instruction on this cell. The first N-1 processes to do so will die. Only the last one will continue beyond the `join` instruction.

4.5 Steps, critical paths, and parallelism profiles

When we execute an instruction in a particular process, the process may continue (usual case), die (`stop` and `join` instructions) or continue and create new processes (`fork` instruction).

Our overall view of execution is this. We initialize the machine with the primordial process. Then, we repeatedly conduct a *step*. At step j, let S_j be the set of processes that are alive. We execute *one* instruction in each of these processes. For each such instruction executed, we will put the resulting zero, one or two processes into a new set S_{j+1}.

As we repeatedly perform steps, the number of processes can grow and shrink— some terminate, some do ordinary instructions, some fork new processes. At some time n, all processes will have terminated (unless, of course, the program has an infinite loop). At this time, we say that the entire program has terminated. We call n the *critical path* of the program, i.e., it is the shortest time necessary to execute this program.

We can draw a graph that plots $|S_j|$ (the number of processes alive at time step j) versus j. This is called the *parallelism profile* of the program, i.e., it tells us, for each time step j, how many things could be done in parallel for that program.

Note: this is an idealization of a parallel machine because it assumes:

- We can execute an instruction in every process at each step
- Each instruction takes the same time to execute (one step)

Neither assumption is realistic, but this idealization is sufficient to start build up our intuitions about parallelism.

5 A Parallel Explicit-Control Evaluator (Strict)

We now have enough mechanisms to specify a parallel evaluator for Scheme. The objective in this section is to design the controller code so that it has the following behavior. It will have the usual `EVAL-DISPATCH` code, performing the usual evaluation of expressions, *except* when it encounters a combination:

```
(e1 ... eN)
```

At this point, it will fork N-1 new processes. The parent process and these child processes are each responsible for evaluating one of the components of the application. When done, they all synchronize using the `join` instruction. The single surviving process goes on to `APPLY-DISPATCH` to apply the procedure value to the argument values.

Thus, the change from the sequential evaluator is in the section of controller code beginning at label `EV-APPLICATION` and ending at `APPLY-DISPATCH`.

When a process arrives at `EV-APPLICATION`, its state is this:

- `exp`: a combination `(e1 e2 ... eN)`
- `env`: an environment
- `continue`: a label L to go to after the value has been computed and stored in the `val` register.

The N processes that we set up will all get the following information in common (in their stacks):

- The label L
- A new structure `(e1 eN ... e3 e2)` to be used for `fun` and `argl` when we get to APPLY-DISPATCH. We will call this an "item list".
- A join counter `(N)`
- The environment from the `env` register.

In addition, each process also gets a reference to one of the cells in the item list. The process is responsible for evaluating the expression in the CAR of the cell and replacing the CAR by its value. Thus, when all N processes `join` together again, the surviving process can be assured that the item list now contains

```
(v1 vN ... v3 v2)
```

It can then place `v1` into `fun` and `(vN ... v2)` into `argl`, and it is now ready for `APPLY-DISPATCH`.

Let us look at the code to achieve this.

```
EV-APPLICATION
  (save continue)

;;; Build an item-list: ARGL:(e1 eN ... e2); and VAL:N
  (assign unev (cdr (fetch exp)))    ; UNEV: (e2 ... eN)
  (assign fun (car (fetch exp)))     ; FUN: e1
  (assign val 1)
  (assign argl '())                  ; ARGL: ()
```

```
BUILD-ITEM-LIST-LOOP
  (branch (null? (fetch unev)) ITEM-LIST-LOOP-DONE)
  (assign val (+ 1 (fetch val)))               ; VAL: j
  (assign exp (car (fetch unev)))              ; EXP: ej
  (assign unev (cdr (fetch unev)))             ; UNEV: (ej+1 ... eN)
  (assign argl (cons (fetch exp) (fetch argl)))  ; ARGL: (ej ... e2)
  (goto BUILD-ITEM-LIST-LOOP)
```

At this point, `fun` contains `e1`, `argl` contains `(eN ... e2)`, and `val` contains N. We can now put together the common information for all the N processes.

```
ITEM-LIST-LOOP-DONE
  (assign argl (cons (fetch fun) (fetch argl))) ; ARGL: (e1 eN ... e2)
  (assign val (cons (fetch val) '()))          ; VAL: (N)
  (save argl)
  (save val)
  (save env)
```

Now, the `stack` contains the environment, the join counter `(N)`, the itemlist `(e1 eN ... e2)` and the continuation label L. We are ready to fork N-1 processes to `EVAL-APP-ITEM`, after which this process itself goes to `EVAL-APP-ITEM` to evaluate the last item.

```
FORK-LOOP
  (save argl)
  (assign argl  (cdr (fetch argl)))
; Now: STACK: (x y ...),E,(N),(e1 eN ... e2),L,...
;      ARGL:    (y ...)
  (branch (null? (fetch argl)) EVAL-APP-ITEM)
  (fork EVAL-APP-ITEM (fetch stack))      ; Applic-item evaluation forked

  (restore exp)                           ; discard (x y ... )
  (goto FORK-LOOP)
```

Each of the N processes evaluating an application item begins here.

```
;;; Assume: STACK: (ei ...),Env,(N),(e1 eN ... e2),L
;;; Effect: VAL:eval(ei, Env); PC: SET-VALUE; STACK: (N),(e1 eN ... e2),L

EVAL-APP-ITEM
  (restore exp)
  (restore env)
  (save exp)                      ; STACK: (ei ...),(N),(e1 ... e2),L
  (assign exp (car (fetch exp)))  ; EXP: ei
  (assign continue SET-VALUE)
  (goto EVAL-DISPATCH)
```

After evaluation, we return to `SET-VALUE`.

```
;;; Assume: VAL: vi  STACK: (ei ...),(N),(e1 eN ... e2),L,...
;;; Effect: Replace ei by vi in argl-cell at top of stack, pop stack.
;;;         Decrement arg-count (N) in frame. If not 0, stop.
;;;         Else go on towards APPLY-DISPATCH
SET-VALUE
  (restore exp)                                ; (ei ... )
  (perform (set-car! (fetch exp) (fetch val)))
  (restore exp)                                ; (N)
  (join (fetch exp))                           ; die if not last
```

Only one of the N processes survives this last `join`.

```
;;; Assume: STACK: (v1 vN ... v2), continuation, ...

   (restore argl)
   (assign fun (car (fetch argl)))
   (assign argl (cdr (fetch argl)))
   (goto APPLY-DISPATCH)
```

5.1 Conclusion

In moving to a parallel evaluator, all we needed were two new instructions— `fork` and `join`. The only part of the code that we had to change was the `EV-APPLICATION` section, upto `APPLY-DISPATCH`.

6 Strictness and Non-strictness

6.1 Evaluating arguments and procedure bodies in parallel

Consider the following program:

```
(define (f x y) (+ (sqrt x) y))

(f 23 (sqrt 49))
```

Our current evaluator (in Section **??**) evaluates `f`, `23` and `(sqrt 49)` in parallel. Then, it invokes the function (value of `f`), which evaluates `+`, `(sqrt 23)` and `7` in parallel, and finally does the application and returns the sum. Thus, the total time for the program will be at least the sum of the times for the two `sqrt` computations.

Similarly, consider this program:

```
(define (g x y)
    (if (> x 0)
        x
        y))

(g 23 (sqrt 49))
```

Our current evaluator evaluates `g`, `23` and `(sqrt 49)` in parallel. Then, it invokes the function, which will return the value of `x` (i.e., 23), discarding the value of `y`. Thus, the total time includes the time for the `sqrt` computation, *even though it was not needed for the result.*

In both cases, if we could evaluate the function body without waiting for the argument values to be ready, we could return the answer in much less time. In the first example, the two `sqrt` computations would overlap in time. In the second example, the time for the `sqrt` computation is irrelevant— the answer can be produced much sooner.

In our current evaluator, the evaluation of a combination (i.e., application) cannot return a value until *all* the arguments have been evaluated, even if the procedure ignores some or all of them. We call this behavior "*strict*". We implemented this behavior using the `join` instruction.

We will soon study another parallel evaluator in which the evaluation of a combination can return a value even if some of the arguments have not been evaluated yet. We call this behavior "*non-strict*".

The non-strict evaluation of `g` also illustrates some other points. First, the notions of

- "the machine has returned an answer", and
- "the machine has terminated"

are not necessarily synonymous. This is because, even though we may return the value 23 early, the process responsible for `(sqrt 49)` may still be active.

This leads us to the second point, which is that we may have a "resource-management" problem, in that we have wasted some processes on a useless computation. Of course, this is no worse than in a strict evaluator, where, also, we would have done the `sqrt` computation anyway.

The advantages of non-strictness become dramatic when we consider data structures. In particular, to evaluate the expression:

```
(cons e1 e2)
```

we assume that we can allocate and return a cons cell immediately, without knowing the values of `e1` and `e2`. We would have forked off two concurrent processes that are busy evaluating `e1` and `e2` and will store their results in the cons cell when they are done but, in the meanwhile, we can return the pointer to the cons cell immediately. It is only when we try to read the `car` or the `cdr` of the cell that we really need to know the values of `e1` or `e2`. (The astute reader may see a connection here with streams, `FORCE`, `DELAY`, etc. Yes! The connection is deep— we'll explore this issue in more detail later in this Section.)

Let us explore the implication of non-strict conses further. Consider:

$(\texttt{cons}_1\ \texttt{10}\ (\texttt{cons}_2\ \texttt{20}\ (\texttt{cons}_3\ \texttt{30 nil})))$

where we have subscripted the `cons`'es for reference. The process (say, P1) sees the outermost application first, and we soon have three processes (P1, P2 and P3) evaluating $\texttt{cons}_1$, `10` and $(\texttt{cons}_2\ \ldots)$ in parallel. As soon as $\texttt{cons}_1$ has evaluated to the cons procedure, P1 applies it, which allocates and return the first cons cell immediately. Meanwhile, P2 evaluates `10` and stores it into the car field. And, meanwhile, P3 has forked two more processes P3.1 and P3.2, and the three of them are working on $\texttt{cons}_2$, `20` and $(\texttt{cons}_3\ \ldots)$, respectively. Again, P3 can allocate the second cons and store its pointer into the cdr field of the first cons immediately; and so on ...

Thus, the conses for the list are allocated "from the roof down": cell_1, then cell_2, then cell_3. Contrast this with our a strict evaluator, where the conses get "from the ground up": cell_3, then cell_2, and finally cell_1.

However, this non-strictness introduces a new synchronization requirement— since we can return a pointer to a cons cell whose components are still "empty", we must be careful that we do not try to read those component until they are "full". Consider:

```
(car (cons (sqrt 49) nil))
```

The process for `car` (say, P1) may receive a pointer to the cons cell before the process for `(sqrt 49)` (say, P2) has stored 7 in the car field. P1 must somehow *wait* until the car field is no longer empty, i.e., until P2 has stored the value.

Now, let us look at a more significant example. Given the usual definition of `mapcar`:

```
(define (mapcar proc lst)
  (if (null? lst)
      nil
      (cons (proc (car lst))
            (mapcar proc (cdr lst)))))
```

let us assume that the following computation:

```
==> (mapcar square '(1 2 3 4))
```

(1 4 9 16)

takes a certain time T. Now, consider:

```
==> (mapcar₁ square (mapcar₂ square '(1 2 3 4)))
```

(1 16 81 256)

In our strict evaluator, $\texttt{mapcar}_1$ cannot proceed until $\texttt{mapcar}_2$ has returned the entire list `(1 4 9 16)`. It then goes to work, and returns the final result. Thus, the total time taken will be at least $2T$.

In a non-strict evaluator, $\texttt{mapcar}_2$ can return its first cons cell immediately to $\texttt{mapcar}_1$, which, in turn, can then return its first cons cell— the first cons cell of the result— immediately ($\texttt{mapcar}_1$ must wait for this in order to know whether to return a cons cell or `nil`). Similarly, as soon as $\texttt{mapcar}_2$ has produced its second cons cell, $\texttt{mapcar}_1$ can produce its second cons cell, and so on. As soon as `square` inside $\texttt{mapcar}_2$ has placed 9 in its target cons cell, the `square` in $\texttt{mapcar}_1$ can read it and place 81 into its target cons cell. Thus, the two `mapcar` computations can overlap in time, behaving in a “pipelined”, or “producer-consumer” manner. The total computation time, thus, can be much less thatn $2T$; ideally, it will be close to T.

Non-strict computations thus have the property that they compose well for parallelism. If $f(x)$ and $g(x)$ take times T_f and T_g, respectively, the computation $f(g(x))$, instead of taking $T_f + T_g$, may take as little as $\max(T_f, T_g)$.

6.2 Non-strict programs

Non-strict evaluation allows us to express certain computations that we could not have expressed otherwise. Let us start with a simple (if useless, perhaps) example:

```
(let ()
  (define x (cons 23 (1+ (car x))))
  x)
```

In a strict evaluator, we first try to evaluate `(cons 23 (1+ (car x)))`, which soon tries to evaluate `x`. But `x` is not bound yet, so the program is in error.

In a non-strict evaluator, the `cons` can immediately return a cons cell to be bound to `x`; 23 is evaluated and stored in the car field; so, `(car x)` can then read this and return 23, and, soon, 24 is stored in the cdr field. We thus get the answer:

```
(23 . 24)
```

This is a simple example of a data structure defined in terms of itself (i.e., recursively). We could not do this in a strict evaluator; there, we could define something recursively only if the recursive reference is protected by a `lambda`. Basically, we rely on the fact that when we evaluate a `lambda` form, we do not look inside it, and so we don't try to evaluate the recursive reference. In a non-strict evaluator, we do not have this arbitrary restriction— anything, including data structures, can be defined recursively.

Previously, we encountered this ability to define data structures recursively when we studied delayed evaluation and streams. In fact, a normal-order evaluator is another way to achieve exactly the same abstract idea of non-strictness.

Suppose we have a program where we say `(DELAY` *e*`)`, producing an object *d* which we later examine by saying `(FORCE` *d*`)`. There is *nothing in the program that indicates exactly when e gets evaluated!* It could have happened at *any* time after the `DELAY` and before the `FORCE`— in particular, it could have happened in parallel with the rest of the program. Thus, non-strictness is an abstract property of programs (i.e., it is a "denotational" property), whereas normal-order and parallel evaluation are properties of implementations (i.e., they are "operational" properties).

The difference between a normal-order evaluator and a parallel evaluator can be viewed as representing different choices as to when the execution of a particular expression is scheduled. In the former, it is scheduled only when demanded, whereas in the latter, it is scheduled to execute in parallel with other activities.

A hallmark of non-strict evaluators is that they will always have some place where they "wait if empty". This is exactly the synchronization requirement we referred to above, when we said that the `car` function must wait until the car of a cons-cell is full. In the (memo-ized) implementation of `FORCE`, we test to see if the expression has already been evaluated or not. In a parallel non-strict evaluator, we have seen that the `car` function must wait until the car field of the cons cell has been filled in by a concurrent process, i.e., we need some way of distinguishing "empty" from "full", or "not yet evaluated" from "evaluated". And, we will see in Section **??**, a parallel non-strict evaluator will also have to wait on empty slots in environment frames.

Given these connections between `FORCE`/`DELAY`/streams and parallel, non-strict evaluation, it is not surprising that side-effects are a major problem in both cases. In fact, it is the same problem! To understand the behavior of side-effects, it is crucial to know the *order* in which they get done. In any non-strict evaluator, it is very difficult to predict the order in which things get done.

7 Waiting until an empty cell becomes full

How do we implement non-strictness? We begin by addressing the central problem: How do we implement the “wait if empty” action?

7.1 I-structure Cells

We introduce a new kind of object called an “I-structure cell”, or I-cell, for short. An I-structure cell can viewed as a storage location for a value, accompanied by a status flag that is either `FULL` or `EMPTY`.

To create an I-cell, we have a new machine operation:

```
(make-I-cell)
```

The new I-cell has `EMPTY` status.

To store a value into an I-cell, the machine executes the instruction:

```
(set-I-cell! <I-cell> <value>)
```

To fetch a value from an I-cell into a particular register, the machine executes the instruction:

```
(get-I-cell <register> <I-cell>)
```

This looks like an ordinary storage cell, but the difference is this. When a process executes the `get-I-cell` instruction on an `EMPTY` I-cell, it simply waits until the I-cell becomes `FULL`, i.e., it does not proceed beyond this instruction until the I-cell becomes `FULL`. When a process executes the `set-I-cell!` instruction, it writes a value there, and sets the status to `FULL`, thus releasing any other processes that may be waiting there.

7.2 Implementation of I-structure Cells

A possible implementation for an I-structure cell is this:

```
(define (make-I-cell) (cons 'I-cell (cons 'empty '())))

(define (is-I-cell? x) (has-type? x 'I-cell))
```

The flag can be tested and set:

```
(define (I-cell-flag-set? I-cell) (eq? (cadr I-cell) 'full))

(define (set-I-cell-flag! I-cell) (set-car! (cdr I-cell) 'full))
```

When the cell is FULL, the CDDR contains the value:

```
(define (I-cell-value I-cell) (cddr I-cell))

(define (set-I-cell-value! I-cell v)
  (set-cdr! (cdr I-cell) v))
```

The question now is: how to implement the notion of a process waiting on a cell while it is empty? Let us assume that a reference to the I-cell is in the `exp` register, and we want the value from the I-cell to be loaded into the `val` register. Then, one possibility is to treat the instruction:

```
(get-I-cell val (fetch exp))
```

as an abbreviation for the following instruction sequence:

```
WAIT-LOOP
  (branch (I-cell-flag-set? (fetch exp)) IS-FULL)
  (goto WAIT-LOOP)
IS-FULL
  (assign val (I-cell-value (fetch exp)))
```

i.e., we spin in a tight loop as long as the I-cell is empty. This is traditionally known as the "busy waiting" solution. It is not very satisfactory, because the process sits in a tight loop, executing instructions that are not doing "useful" work.

Instead, a better solution is the "don't call us, we'll call you" solution. Here, we use the value slot of the empty I-cell to hold a "waiting-list" of processes that are waiting for its value. When the cell is first created, the value slot is initialized to `'()`, i.e., an empty waiting list. Then, when a process executes the `get-I-cell` instruction on an empty I-cell, the process puts itself onto the I-cell's waiting list, and takes itself out of circulation. It uses the primitive:

```
(define (add-to-I-cell-waiting-list I-cell process)
  (set-cdr! (cdr I-cell) (cons process (cddr I-cell))))
```

When some other process executes a `set-I-cell!` instruction, it extracts the waiting list, stores the value in its place, and puts all processes that are on the waiting list back in circulation. It uses the primitive:

```
(define (I-cell-waiting-list I-cell) (cddr I-cell))
```

to extract the waiting list.

What does it mean to "take a process out of circulation" and "put it back in circulation"? Recall that, in our model, each process is simply a set of nine registers. At each step, we have a set of processes that are executing. This set is also called the "ready list", i.e., processes that are ready to execute an instruction. Now, a process can be taken out of circulation by simply removing it from the ready list; it can be put back in circulation by placing it back on the ready list.

Note that the program-counter of any waiting process is still pointing at the `get-i-cell` instruction that put it on a waiting list. When the process becomes active again, it will retry that instruction and, this time, it will successfully read a value and proceed to the next instruction.

8 A Parallel Explicit Control Evaluator (non-strict)

With I-structure cells, we are now adequately armed to implement a non-strict, parallel evaluator for Scheme. Despite the fact that non-strictness appears to be a more sophisticated concept, it turns out that the evaluator itself is simpler. Again, starting with the sequential explicit-control evaluator of the textbook as a reference point, the changes we make are in three places in the controller code:

- As in the strict parallel evaluator, the major change is in the `EV-APPLICATION` segment, where, instead of an `EVAL-ARG-LOOP`, we have a `FORK-ARGS-LOOP` to fork off processes to compute the arguments in parallel. Each forked argument-evaluation is given an I-cell in which to store its result. The parent process collects these I-cells in its argument list (`argl` register), and goes on to evaluate the operator and, unlike the strict evaluator, proceeds immediately to `APPLY-DISPATCH` to perform the application. Thus, when `APPLY-DISPATCH` builds a new environment frame, it will bind the formal parameters to the corresponding elements in the `argl` list, i.e., to I-structure cells that will later hold the argument values.
- In `EV-SYMBOL`, we lookup the values of variables in the environment. The lookup procedure returns an I-cell for the argument, and so we use `get-i-cell` to wait for it, if necessary. Obviously, if a procedure body ignores some argument, it will never be looked up, so it will not matter if the argument is not available. This is exactly how we achieve non-strictness.
- In the code for applying strict primitive operators (such as in `APPLY-PLUS`), we use `get-i-cell` to wait for the arguments in `argl`, if necessary.

8.1 Applications (`EV-APPLICATION`)

When a process arrives at `EV-APPLICATION`, the `exp` register holds:

```
(e1 e2 ... eN)
```

First, save the continuation, pick the expression apart, and initialize the argument list to `'()`:

```
EV-APPLICATION
  (save continue)
  (assign unev (cdr (fetch exp)))    ; (e2 ... eN)
  (assign exp  (car (fetch exp)))    ; e1
  (assign argl '())
```

We now loop, creating a new I-cell and forking off a process for each argument, and building up the list of I-cells in `argl`. Each new process needs the environment, an argument expression and the I-cell into which it must store the value of that argument:

```
FORK-ARGS-LOOP
  (branch (null? (fetch unev)) EVAL-OP)

;;; --- Fork an argument; VAL is used as a temporary
  (assign val  (make-I-cell))          ; I-cell for this arg
  (save val)
  (assign argl (cons (fetch val) (fetch argl)))
  (save env)
```

```
    (assign val (car (fetch unev)))      ; eJ
    (assign unev (cdr (fetch unev)))     ; (eJ+1 ... eN)
    (save val)                           ; stack: eJ, env, I-cell, ...
    (fork FORKED-EVAL (fetch stack))     ; fork it

    (restore val)                        ; discard
    (restore val)                        ;    eJ, env, I-cell
    (restore val)                        ; from stack
    (goto FORK-ARGS-LOOP)
```

After all the argument processes are forked, the parent process goes on to evaluate the operator (i.e., `e1`):

```
  EVAL-OP
    (save argl)

    (assign continue EVAL-OP-DONE)
    (goto EVAL-DISPATCH)
```

and then sets up and goes to do the application.

```
  EVAL-OP-DONE
    (assign fun (fetch val))
    (restore argl)
    (goto APPLY-DISPATCH)
```

Note that the process *does not wait* for the arguments to be completely evaluated!

Meanwhile, in parallel, each forked argument evaluation begins here.

```
  ;;; Assume: stack: eJ, env, I-cell
  ;;; Effect: stop after storing eval(eJ, env) into I-cell

  FORKED-EVAL
    (restore exp)
    (restore env)
    (assign continue SET-VALUE)
    (goto EVAL-DISPATCH)
```

And, after the argument is evaluated, it stores the value in the waiting I-cell and the process dies.

```
  SET-VALUE
    (restore exp)                           ; the I-cell
    (set-I-cell! (fetch exp) (fetch val))
    (stop)
```

Note: The `set-I-cell!` instruction will wake up any processes that may be waiting for this value.

Minor note: in the strict evaluator, any of the N processes (parent or N−1 children) could finish last, and that process would go on to `APPLY-DISPATCH`. Here, it is always the parent process that goes to `APPLY-DISPATCH` as soon as the procedure value is ready. Each of the child processes always stops after storing a value into its I-cell.

8.2 Looking up variables (EV-SYMBOL)

When a process arrives at EV-SYMBOL, the exp register has a symbol in it. It fetches the relevant I-cell from the environment, and uses get-I-cell to fetch its value, which will cause it to wait, if necessary:

```
EV-SYMBOL
  (assign val (lookup (fetch exp) (fetch env)))
  (get-I-cell val (fetch val))                  ; wait for it
  (goto (fetch continue))
```

8.3 Non-strict CONS, CAR and CDR

To execute the cons primitive operation, the process simply conses up references to the two I-cells in the frame (for the car and cdr), i.e., it doesn't wait for either I-cell to receive a value:

```
APPLY-CONS
  (assign val (car (fetch argl)))         ;; the I-cell for the cdr
  (assign exp (cdr (fetch argl)))
  (assign exp (car (fetch exp)))          ;; the I-cell for the car

  (assign val (cons (fetch exp) (fetch val)))
  (restore continue)
  (goto (fetch continue))
```

To execute the car primitive operation, the process waits for its argument (a cons cell), extracts the I-cell for the car, and waits for its value:

```
APPLY-CAR
  (assign exp (car (fetch argl)))
  (get-I-cell val (fetch exp))      ; wait for the cons cell

  (assign val (car (fetch val)))
  (get-I-cell val (fetch val))      ; wait for the car

  (restore continue)
  (goto (fetch continue))
```

And, APPLY-CDR is similar.

8.4 Strict primitives (e.g. APPLY-PLUS)

Unlike cons, a strict primitive like + must wait for both its arguments before it can return any value:

```
APPLY-PLUS
  (assign exp (car (fetch argl)))
  (get-I-cell val (fetch exp))        ;; wait for arg 2
  (assign exp (cdr (fetch argl)))
  (assign exp (car (fetch exp)))
  (get-I-cell exp (fetch exp))        ;; wait for arg 1
```

```
    (assign val (+ (fetch exp) (fetch val)))
    (restore continue)
    (goto (fetch continue))
```

8.5 Deadlock

Though non-strict evaluation gives us a more powerful programming language, it also introduces a new source of errors— deadlock. Consider the following (pathological) program:

```
(let ()
  (define x (cons (car y) '()))
  (define y (cons (car x) '()))
  (car x))
```

The two `cons`'es can allocate cons cells immediately, and bind them to `x` and `y`. The process (say P1) responsible for filling in `x`'s car slot gets `y` and attempts to read its car slot. Similarly, the process (say, P2) responsible for filling in `y`'s car slot gets `x` and attempts to read its car slot. The main process (say, P0), meanwhile, gets `x` and tries to read its car slot. Now, both the car slots are empty, so we will end up with `P0` and `P2` on the waiting lists of `x`'s car slot, and `P1` on the waiting list of `y`'s car slot, and *zero* processes ready to execute!

This kind of a situation, where we have a cycle of processes waiting for each other, is called a "deadlock" or a "deadly embrace", and is one of the major pitfalls to watch out for in parallel evaluators. Unlike this pathological example, the cycle may be very long and obscure, scattered throughout the program.

9 Orders of growth, revisited

When we first learned about orders of growth, we made certain assertions about the complexity of programs. For example, given the following recursive procedure to count atoms in a tree:

```
(define count-atoms-r (lambda (lst)
    (if (null? lst)
        0
        (if (atom? lst)
            1
            (+ (count-atoms-r (car lst))
               (count-atoms-r (cdr lst)))))))
```

we said that it would take $O(n)$ time and $O(\log n)$ space , where n was the number of nodes in the tree. Now, we must be a little more careful, and realize that these statements are *relative to the underlying execution model*, i.e., the time and space complexities were valid for the sequential execution model.

In our parallel machine models, on the other hand, we can see that the same program will take $O(n)$ space, and can complete in as little as $O(\log n)$ time! This is because all the recursive calls will unfold in parallel so that there is one computation in progress at every node, and the time taken is proportional only to the depth of the tree.

Another example— the sum of the square-roots of the numbers in a list:

```
(define sum-of-sqrts (lambda (lst)
    (sum-of-sqrts-loop lst 0)))

(define sum-of-sqrts-loop (lambda (lst s)
    (if (null? lst)
        s
        (sum-of-sqrts-loop (cdr lst)
                           (+ s (sqrt (car lst)))))))
```

In all three evaluators, this iterative computation should take $O(n)$ time, where n is the length of the list. In both the sequential and the parallel, strict evaluators, it should take constant space (i.e., $O(1)$). However, in the parallel, non-strict evaluator, it will take $O(n)$ space. This is because, in the non-strict evaluator, the recursive call can race ahead, traversing the entire list, selecting all the `car`'s, and initiating all the `sqrt`'s even before the first `sqrt` computation has finished, i.e., all the `sqrt` computations can proceed in parallel. Thus, an iteration is not necessarily "finished" before the next one starts, and so, we cannot reclaim or reuse the space that it occupies.

The last point is an illustration of a general axiom about parallelism: Except when the computation is inherently sequential, you can usually trade parallelism for space.

10 Side-effects, revisited

10.1 More insight into the problem

In Section **??** we pointed out that side-effects may be especially tricky under parallel evaluation, because they lead to "race conditions". We illustrate the problem here again using an example based on the simple bank balance from Section 3.1.1 of the textbook:

```
(define balance 100)

(define withdraw (lambda (amount)
    (set! balance (- balance amount))
    (list amount balance)))
```

(To simplify the presentation, we are not checking for errors such as negative balances, etc.)

Now, suppose we had a parallel evaluator, and two customers were executing these forms at the same time:

```
Joe:   ... (withdraw 10) ...
Moe:   ... (withdraw 15) ...
```

A problem arises because the `set!` form actually consists of two separate activities— first, computing `(- balance amount)` and then updating the binding for `balance`. In a parallel evaluator, the following sequence of events is possible:

Joe	Moe
...	...
`(- 100 10)`	...
...	`(- 100 15)`
`(set!  balance 90)`	...
...	`(set!  balance 85)`
...	...

Thus, the final balance is 85, an outcome that the bank is likely to be very unhappy about, since it has just handed out $25 out of the original $100.

Thus, we need to ensure that the entire `set!` activity is an *atomic* action, i.e., indivisible, so that either Joe's entire transaction precedes Moe's, or vice versa, but they never interleave.

Some more terminology: The `set!` part of the program is also known as a *critical section*, and the requirement that no more than one process can execute it at a time is known as a *mutual exclusion* requirement.

It is not enough for our evaluator somehow to arrange for `set!` alone to be performed atomically. Critical sections can encompass larger program regions. For example, suppose we had two accounts, a money-transfer procedure, and a balance-inquiry procedure:

```
(define balance2 100)
(define balance3 200)

(define transfer-2-to-3 (lambda (amount)
  (set! balance2 (- balance2 amount))
  (set! balance3 (+ balance3 amount))
  (list amount balance2 balance3)))

(define tot-balance (lambda ()
  (+ balance2 balance3)))
```

Now, suppose we are executing the following in parallel:

```
Joe:   ... (transfer-2-to-3 50) ...
Moe:   ... (tot-balance) ...
```

Again, the following sequence of events can occur:

Joe	Moe
...	...
`(set!  balance2 (- 100 50))`	...
...	`(+ 50 200)`
`(set!  balance3 (+ 200 50))`	...
...	...

Moe will find that the total balance is $250 instead of $300. In this example, the two `set!` forms together should be executed atomically, and together they constitute a critical section.

Actually this problem arises at another level in both programs. In the `withdraw` program, it is possible for the two processes to execute the `set!` before the first process can evaluate `(list amount balance)`, so that the first process gets the wrong balance. A similar problem occurs in the second program.

10.2 Locks

To address this issue, we introduce a new kind of an object called a *lock*, and a special form:

```
(HOLDING <lock>
    <expression>
    ...
    <expression>)
```

When a process P evaluates this form, it first tries to "acquire" the lock. Only one process at a time can hold a lock. If some other process Q currently holds the lock, P must wait until Q releases it. When P finally acquires the lock, it proceeds to execute the `<expression>`s in sequence, returning the value of the last one. When the last expression has returned a value, P releases the lock.

In general, there can be many processes waiting for a lock, but only one process may hold it at a time. When a lock is released and there are several waiting processes, only one of them gets it, and the remaining processes continue to wait.

We assume a procedure:

```
(define (make-lock) ...)
```

that creates, and returns a new lock.

We can now solve our first problem as follows.

```
(define balance 100)
(define lock (make-lock))

(define (withdraw amount)
    (HOLDING lock
       (set! balance (- balance amount))
       (cons amount balance)))
```

Now, when Joe and Moe try to withdraw money at approximately the same time, one of them acquires `lock`, performs the transaction, and releases the lock, at which time the other can acquire it and perform his entire transaction.

We can solve the second problem as follows:

```
(define balance2 100)
(define balance3 200)
(define lock2 (make-lock))

(define (transfer-2-to-3 amount)
  (HOLDING lock2
    (set! balance2 (- balance2 amount))
    (set! balance3 (+ balance3 amount))
    (list amount balance2 balance3)))

(define (tot-balance)
  (HOLDING lock2
    (+ balance2 balance3)))
```

10.3 Implementation of locks

Locks can be implemented using a mechanism similar to that for I-structure cells. First:

```
(define (make-lock) (cons 'free '()))
```

i.e., a lock is simply a pair with a flag intialized to `FREE` and an empty waiting list of processes.

In the evaluator, we assume two new instructions for dealing with locks. The instruction

```
(acquire-lock  lock)
```

when executed in process P, advances the program counter, and tests if the lock is in the `FREE` state. If it is free, P simply continues (at the next instruction), after setting the lock flag to `BUSY`. If the lock is already `BUSY`, P is added to the waiting list in the lock, and P is taken off the ready list of processes.

The instruction:

```
(release-lock  lock)
```

when executed in process P always succeeds and continues at the next instruction. The lock must be in the `BUSY` state, since P must have previously acquired it. If the waiting list on the lock is empty, then the lock flag is set to `FREE`. Otherwise, a process (say, Q) on the waiting list is put back on the ready list of processes, and the waiting list updated to omit Q. Note that Q will be at the instruction just following the `acquire-lock` instruction that put it on the waiting list.

10.4 Other issues

With locks, we have a first step towards dealing with side-effects in a parallel evaluator. However, we have barely scratched the surface of this issue here.

First, raw locks are too primitive, too unstructured a mechanism. It is still upto the programmer to introduce locks and use them correctly. It is easy to make mistakes: we could have forgotten to use the lock in the `tot-balance` procedure, or we could have used the lock only for the `set!`s and forgotten to enclose the third, `(list ...)` expression, again leading to a consistency problem. In general, we would like more powerful abstractions than raw locks.

Second, we have introduced another *deadlock* problem. Suppose, instead of a single lock guarding both accounts, we had two locks, one for each account:

```
(define balance2 100)
(define lock2 (make-lock))
(define balance3 200)
(define lock3 (make-lock))
```

We might redo our transfer and inquiry procedures as follows:

```
(define transfer-2-to-3 (lambda (amount)
  (HOLDING lock2
    (HOLDING lock3
      (set! balance2 (- balance2 amount))
      (set! balance3 (+ balance3 amount))
```

```
        (list amount balance2 balance3)))))

  (define tot-balance (lambda ()
    (HOLDING lock3
      (HOLDING lock2
        (+ balance2 balance3)))))
```

Note that the two procedures happen to aquire the locks in the opposite order. Now, it is possible that process P1, executing the transfer procedure, acquires `lock2` and then tries to acquire `lock3`. Meanwhile, process P2, executing the inquiry procedure, may have acquired `lock3` and is trying to acquire `lock2`. Again, we will have a situation where both procedures are holding one lock, and neither can make progress because they need a lock that the other one holds.

Or, consider the situation where, after acquiring a lock, a process goes off into an infinite loop, never releasing it, so that other processes wait indefinitely for the lock.

Another problem is that of *fairness*. Suppose process P0 has acquired a lock, and P1 and P2 are waiting for it. When P0 releases it, P1 gets it, and P2 remains waiting. A little later, P0 again joins the waiting list with P2. When P1 releases it P0 gets it. In this way, it is possible for a process like P2 to wait unreasonably long, or even forever (this is called *starvation*).

It should be clear that dealing with side effects in a parallel evaluator is an enterprise not to be taken lightly.

11 Real parallel machines: finite resources

Our parallel computation model, so far, has been one of repeated *steps* across all "ready" processes. We had the concept of a a "ready list" of all processes that were ready to execute an instruction. At each time step, we executed one instruction from each of the ready processes, and constructed a new ready list.

This model is highly idealized, and is only useful in that it gives us some intuition about the time-independent, resource-independent aspects of parallel programs, mechanisms and processes, such as the critical path, parallelism profile, instruction counts, etc. The characteristics and behavior of a real machine are likely to be very different. It is beyond the scope of these notes to explore all these issues in any level of detail; we mention some of them here just to give the reader a flavor of what is involved.

First, it is unlikely that all instructions take the same time to execute, so processes will not advance in lock-step, instruction by instruction.

Second, a real machine will not have an infinite supply of processors. In a real machine, when we have more processes than physical processors, we must somehow arrange to *multiplex* the available physical processors amongst the processes. Apart from the fact that this will lengthen the computation because some things that were done in parallel are now done sequentially, it will also lengthen the computation because it will introduce some management overhead, i.e., extra instructions to do the book-keeping and and scheduling activity of the multiplexing.

Multiplexing m physical processors PP amongst n processes LP, where $m < n$ is almost always a challenging task, and introduces new problems in its own right. For example, at each instant in time, how do we choose which m-subset of the LPs gets to use the PPs? Typically, some LPs are more important, or crucial to the computation, than others— how do we recognize them? A bad choice, for example, can schedule an LP to execute that reads from an empty cell, while the LP that writes into the cell is not yet scheduled. Thus, this kind of multiplexing can introduce deadlock into an otherwise deadlock-free program.

Similarly, we need to take decisions like, "LP45 will run on PP33". How do we avoid the situation where all of the LPs are assigned to run on a few PPs while the other PPs are sitting idle? This issue is called the "load balancing" problem.

Third, we have assumed that forking a new process is instantaneous, i.e., it happens within one instruction, and that access to data is uniformly fast. However, in a real machine, forking a process involves, among other things, transporting the process state to the physical processor that is to execute it, which may be on the other side of the machine. Similarly, how do we ensure that a process runs on a physical processor that is "near" the data that it will access?

These kind of "resource management" issues are major topics for research. In fact, one might say that they are the central issues in parallel computation.

Contents