

## Lazy Evaluation

Lazy evaluation is sometimes called “call by need.” We do an evaluation when a value is used; not when it is defined.

Scheme provides for lazy evaluation:

```
(delay expression)
```

Evaluation of `expression` is delayed. The call returns a “promise” that is essentially a lambda expression.

```
(force promise)
```

A promise, created by a call to `delay`, is evaluated. If the promise has already been evaluated, the value computed by the first call to `force` is reused.

Example:

Though `and` is predefined, writing a correct implementation for it is a bit tricky.

The obvious program

```
(define (and A B)
  (if A
      B
      #f)
)
```

is incorrect since `B` is always evaluated whether it is needed or not. In a call like

```
(and (not (= i 0)) (> (/ j i) 10))
```

unnecessary evaluation might be fatal.

An argument to a function is *strict* if it is always used. Non-strict arguments may cause failure if evaluated unnecessarily.

With lazy evaluation, we can define a more robust `and` function:

```
(define (and A B)
  (if A
      (force B)
      #f)
)
```

This is called as:

```
(and (not (= i 0))
     (delay (> (/ j i) 10)))
```

Note that making the programmer remember to add a call to `delay` is unappealing.

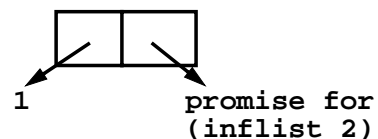
Delayed evaluation also allows us a neat implementation of suspensions. The following definition of an infinite list of integers clearly fails

```
(define (inflist i)
  (cons i (inflist (+ i 1))))
```

But with use of delays we get the desired effect in finite time:

```
(define (inflist i)
  (cons i
        (delay (inflist (+ i 1)))))
```

Now a call like `(inflist 1)` creates



We need to slightly modify how we explore suspended infinite lists. We can't redefine `car` and `cdr` as these are far too fundamental to tamper with.

Instead we'll define `head` and `tail` to do much the same job:

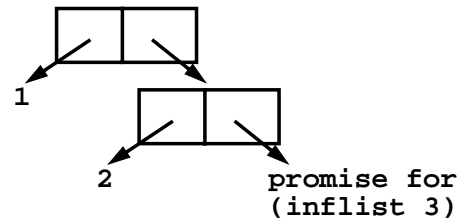
```
(define head car)
(define (tail L)
  (force (cdr L)))
```

`head` looks at `car` values which are fully evaluated.

`tail` forces one level of evaluation of a delayed `cdr` and saves the evaluated value in place of the suspension (promise).

Given

```
(define IL (inflist 1))
(head (tail IL)) returns 2 and
expands IL into
```



## Exploiting Parallelism

Conventional procedural programming languages are difficult to compile for multiprocessors.

Frequent assignments make it difficult to find independent computations.

Consider (in Fortran):

```
do 10 I = 1,1000
  X(I) = 0
  A(I) = A(I+1)+1
  B(I) = B(I-1)-1
  C(I) = (C(I-2) + C(I+2))/2
10 continue
```

This loop defines 1000 values for arrays `x`, `A`, `B` and `C`.

Which computations can be done in parallel, partitioning parts of an array to several processors, each operating independently?

- $X(I) = 0$

Assignments to `x` can be readily parallelized.

- $A(I) = A(I+1)+1$

Note that each computation of  $A(I)$  uses an  $A(I+1)$  value that is yet to be changed. Thus a whole array of new `A` values can be computed from an array of "old" `A` values in parallel.

- $B(I) = B(I-1)-1$

This is less obvious. Each  $B(I)$  uses  $B(I-1)$  which is defined in terms of  $B(I-2)$ , etc. Ultimately all new `B` values depend only on  $B(0)$  and `I`. That is,  $B(I) = B(0) - I$ . So this

computation can be parallelized, but it takes a fair amount of insight to realize it.

- $C(I) = (C(I-2) + C(I+2))/2$   
It is clear that even and odd elements of  $c$  don't interact. Hence two processors could compute even and odd elements of  $c$  in parallel. Beyond this, since both earlier and later  $c$  values are used in each computation of an element, no further means of parallel evaluation is evident. Serial evaluation will probably be needed for even or odd values.

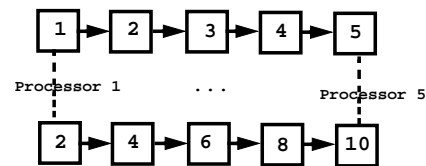
## Exploiting Parallelism in Scheme

Assume we have a shared-memory multiprocessor. We might be able to assign different processors to evaluate various independent subexpressions.

For example, consider

```
(map (lambda(x) (* 2 x))
      '(1 2 3 4 5))
```

We might assign a processor to each list element and compute the lambda function on each concurrently:



## How is Parallelism Found?

There are two approaches:

- We can use a "smart" compiler that is able to find parallelism in existing programs written in standard serial programming languages.
- We can add features to an existing programming language that allows a programmer to show where parallel evaluation is desired.

## Concurrentization

Concurrentization (often called parallelization) is process of automatically finding potential concurrent execution in a serial program.

Automatically finding current execution is complicated by a number of factors:

- Data Dependence

Not all expressions are independent. We may need to delay evaluation of an operator or subprogram until its operands are available.

Thus in

```
(+ (* x y) (* y z))
```

we can't start the addition until both multiplications are done.

- Control Dependence

Not all expressions need be (or should be) evaluated.

In

```
(if (= a 0)
    0
    (/ b a))
```

we don't want to do the division until we know  $a \neq 0$ .

- Side Effects

If one expression can write a value that another expression might read, we probably will need to *serialize* their execution.

Consider

```
(define rand!
  (let ((seed 99))
    (lambda ()
      (set! seed
        (mod (* seed 1001) 101101))
      seed
    )
  )
```

Now in

```
(+ (f (rand!)) (g (rand!)))
```

we can't evaluate `(f (rand!))` and `(g (rand!))` in parallel, because of the side effect of `set!` in `rand!`. In fact if we did, `f` and `g` might see *exactly* the same "random" number! (Why?)

- Granularity

Evaluating an expression concurrently has an overhead (to setup a concurrent computation). Evaluating some very simple expressions (like `(car x)` or `(+ x 1)`) in parallel isn't worth the overhead cost.

Estimating where the "break even" threshold is may be tricky.

## Utility of Concurrentization

Concurrentization has been most successful in engineering and scientific programs that are very regular in structure, evaluating large multidimensional arrays in simple nested loops. Many very complex simulations (weather, fluid dynamics, astrophysics) are run on multiprocessors after extensive concurrentization.

Concurrentization has been far less successful on non-scientific programs that don't use large arrays manipulated in nested for loops. A compiler, for example, is difficult to run (in parallel) on a multiprocessor.