Chapter 11

Concurrency

When you come to a fork in the road, take it.

— *Yogi Berra*

11.1 Motivation

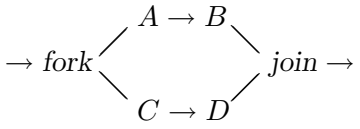
Thus far, all of the languages we have discussed have been characterized by a single locus of control that flows through a program in a deterministic fashion. Assuming all inputs are known in advance, there is only one path that control can take through the program. This single thread of control can be viewed as a time line along which all operations performed by the computation are arranged in a total order. For example, a computation that sequentially performs the operations A, B, C, and D can be depicted as the following total order:

$$\ldots \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow \ldots$$

$X \rightarrow Y$ means that X is performed before Y.

Control can be visualized as a token that resides on the edges of such a diagram and moves according to a set of rules. In a simple diagram like the one above, the only rule is that a token on the input edge to an operation can pass through the operation and end up on its output edge. This step corresponds to performing the operation. The path taken by control can be notated by the sequence of observable actions it performs. If we assume that all operations are observable, the path taken in this case is $ABCD$. In the languages we have considered so far (even those with conditional branching, non-local exits, and exception handling), every program, given a particular input, has exactly one sequential control path.

Purely sequential orderings are too rigid for describing all of the computations we might like to specify. Sometimes it is desirable to specify a relative order between some operations but leave the ordering of other operations unspecified. Here is a sample partial order that declares that A precedes B and C precedes D, but does not otherwise constrain the operation order:



The diagram introduces two new nodes labeled *fork* and *join*. The purpose of these nodes is to split and merge control paths in such a way that a computation has a distinguished starting edge and a distinguished ending edge.

The rules governing how control moves through *fork* and *join* nodes are different from the rules associated with the labeled operations. In one step, a control token on the input edge of a *fork* node splits into two subtokens on the output edges of the node. Each subtoken independently moves forward on its own branch. When tokens are on both input edges of a *join* node, they merge into a single token on the output node. If only one input edge to a *join* has a token, it cannot move forward until the other edge contains a token. Any node like join that forces one control token to wait for another is said to **synchronize** them.

Together, a control token and the sequential subpath along which it moves are called a **thread**. Although *fork* splits a single thread into two, it is common to view the original thread as continuing through one branch of the fork and a new thread as originating on the other branch of the fork.

Programs that may exhibit more than one thread of control are said to be **multi-threaded** or **concurrent**. The interesting feature of concurrency is that multiple threads represent a partially ordered notion of time. Concurrent programs are **non-deterministic** in the sense that control can flow through them in more than one way. Non-determinism at the level of control is often exhibited as non-determinism at the level of program behavior: it is possible for a single program to yield different answers on different executions. While it is possible to add non-determinism to a purely sequential language[1], non-determinism is most commonly associated with concurrent languages.

[1] As an example, consider a `(choose` E_1 E_2`)` form that randomly chooses to return the value of E_1 or E_2.

Suppose that on any step of a multi-threaded computation, only one control token is allowed to move.[2] Then a particular execution of a concurrent program is associated with the sequence of its observable actions, known as its **interleaving**. The **behavior** of the concurrent program is the set of all possible interleavings that can be exhibited by the program. For example, assuming that all operations (except for *fork* and *join*) are observable, then the behavior of the branching diagram above is:

$$\{ABCD,\ ACBD,\ ACDB,\ CABD,\ CADB,\ CDAB\}$$

The behavior of a concurrent program may be the empty set (no interleavings are possible), a singleton set (exactly one interleaving is possible), or a set with more than one element (many interleavings are possible).

We will distinguish concurrency from **parallelism**.[3] In our terminology, concurrency refers to any model of computation supporting partially ordered time. In contrast, we use parallelism to refer to hardware configurations that are capable of simultaneously executing multiple threads on multiple physical processors. Thus, concurrency is a semantic notion and parallelism is a pragmatic one. A concurrent program may be executed by time-slicing its threads on a single processor, or by executing each thread on a separate physical processor.

Concurrent programming languages are important for several reasons:

- *Modularity*: Multiple threads can be simulated in a sequential language by interleaving code fragments or by explicitly managing the transfer of control between program parts. Concurrent languages reap modularity benefits by abstracting over these idioms. They enhance modularity by permitting threads to be specified as separate entities that interact with each other via communication and synchronization. They also separate the specification of the threads from policy issues (such as scheduling threads or allocating threads to processors).

  For example, consider the interaction between the processor(s) of a computer and its numerous input/output devices (keyboards, mice, disks, tape drives, networks, video displays, printers, plotters, etc.). Writing a monolithic program to control all these devices would be a recipe for disaster. Such a program would be hard to understand and modify. In contrast, representing the controller for each device as a separate thread improves readability and facilitates the modification of existing controllers as well as the addition of new ones.

[2] There are concurrent models in which multiple control tokens can move in a single step, but we shall not consider these.

[3] While there is much disagreement about the definition of these terms in the programming languages community, we will strive to use them consistently here.

- *Specifying Parallelism*: Concurrent languages allow programmers to declare which parts of programs can safely be run in parallel on multiple processors or time-sliced on a single processor. Language implementations can use this information to take advantage of the available hardware.

- *Modelling*: The real world can be viewed as a collection of interacting agents. Concurrent languages are natural for simulating the world, because they allow each agent to be represented as a thread.

- *New programming paradigms*: Concurrent languages support programming paradigms that are cumbersome to express in sequential languages. Consider event-based systems, in which entities generate and respond to events. Such systems are more straightforward to express in concurrent languages than sequential languages.

11.2 Threads

We will explore concurrency by providing a precise semantics for a multi-threaded variant of FL!. It is fairly simple to provide an operational semantics for concurrency, and we will use the SOS framework to do so. The reason that operational semantics approaches to concurrency are the easiest is that they simply incorporate the inherent ambiguity about process interleaving into rules, and an SOS semantics can derive alternative meanings for a program based upon your choice of transition ordering. An operational semantics does not necessarily give you any help to find transition orderings that will yield all of the unique meanings for a given program.

An alternative, more complex approach to modeling concurrency is to treat the meaning of a program as the set of all possible answers the program can compute. A denotational semantics of concurrency takes this approach. A denotational semantics represents the meaning of a concurrent program as a powerdomain. A powerdomain of D is the set of all subsets of D. Thus, if D is the set of integers, the powerdomain of D contains all possible sets of integers. A denotational approach considers all possible interleavings of program operations to produce the meaning of a program. We will not pursue denotational approaches to concurrency further because concurrency is readily described within the SOS framework. Denotational approaches are complex because the functional nature of denotational semantics is hard to adapt to a world where functions are one to many, and interfering concurrent commands produce complex valuation function constructions. The interested reader can explore denotational approaches to concurrency to see how they can be adapted to our discussion [Sch86a]).

11.2.1 MUFL!, a Multi-threaded Language

Our main vehicle for studying concurrency will be MUFL!, a *Mu*ltithreaded version of FL!. MUFL! is FL! extended with the following thread constructs:

E ::= ...| `(fork` E`)` | `(join` E`)` | `(thread?` E`)`

- `(fork` E`)`: Creates a new thread to evaluate E and returns a unique **thread handle** identifying the thread.
- `(join` E`)`: E must evaluate to a thread handle t; otherwise, the expression is an error. `join` waits for the thread t to compute a value and then returns it. While `join` is waiting for t, the thread t' in which the `join` is executing is said to be **blocked**.

  `join` may be called more than once on the same thread. After the first call to `join` returns with a value, that value is effectively memoized at the thread handle for subsequent `join`s.
- `(thread?` E`)`: tests whether the value of E is a thread, returning *true* if it is and *false* otherwise.

`join` and `thread?` can actually be treated as new primitive operators, and we will do so below in describing the semantics of these constructs. However, because `fork` returns before evaluating its operand, it must be a new special form.

We consider a few simple examples of the new constructs. `(join (fork` E`))` is equivalent to E:

```
(join (fork (+ 1 2))) ⟶ 3
                  MUFL!
```

A thread handle can be `join`ed more than once:

```
(let ((c (cell 0)))
  (let ((t (fork (begin (cell-set! c (+ 1 (cell-ref c)))
                        (cell-ref c)))))
    (+ (join t) (join t)))) ⟶ 2
                        MUFL!
```

Here is a procedure for summing the leaves of a binary tree that explores subtrees concurrently:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? tree)
        (leaf-value tree)
        (let ((thread1 (fork (tree-sum (left-branch tree))))
              (thread2 (fork (tree-sum (right-branch tree)))))
          (+ (join thread1) (join thread2))))))
```

Without the `fork`s and `join`s, the left-to-right operand evaluation order inherited from FL! would specify that the sum of a left subtree be computed before the right subtree is visited. The order would be overconstrained even if the `let` expression were replaced by the result of substituting the `fork` expressions for `thread1` and `thread2`:

```
(+ (join (fork (tree-sum (left-branch tree))))
   (join (fork (tree-sum (right-branch tree)))))
```

The reason is that the result of the first `join` would have to be computed before control passed to the second `join`.

MUFL! programs can exhibit non-determinism. For example, suppose that `display` prints (uninterruptably) a symbol, `map` maps a procedure over a list to create a new one, and `for-each` performs a procedure on each element of a list. Then the following `jumble` procedure may print any permutation of the elements of its argument list before returning `#u`.

```
(define (jumble lst)
  (for-each join
            (map (lambda (obj) (fork (display obj)))
                 lst)))
```

For instance, `(jumble '(a b c))` may print any of the six sequences `abc`, `acb`, `bac`, `bca`, `cab`, `cba`.

The following `choose` procedure, which may return either one of its two arguments, underscores the non-deterministic behavior of MUFL!:

```
(define (choose a b)
  (let ((c (cell 'ignore)))
    (let ((t (fork (cell-set! c a))))
      (begin (cell-set! c b)
             (join t)
             (cell-ref c)))))
```

The final value returned by `choose` depends on the order in which the two cell assignments are performed. This example exploits what is known as a **race condition**. A race condition exists when two operations vie for the use of some shared resource. In this case, there is contention for the use of the shared cell `c`, and the thread that gets the cell last is the one that determines its value.

Threads introduce a new failure mode for programs: **deadlock**. Deadlock describes a situation in which program execution cannot proceed because threads are waiting for each other (or themselves) to complete. Consider the following expression:

```
(let ((thread-cell (cell 'later)))
  (begin
    (cell-set! thread-cell
               (fork (begin (join (cell-ref thread-cell))
                            1)))
    (join (cell-ref thread-cell))))
```

The forked `begin` can only return `1` after the thread stored in `thread-cell` runs to completion. But the thread stored in `thread-cell` is the one executing the `begin` expression! Since the `begin` expression can never return, the final `join` waits forever in a deadlocked state.

There is also a race condition in this example that shows why debugging multi-threaded code can be very difficult. In this case, the `fork` begins the execution of its `begin` expression while execution also procedes in the `cell-set!`. If the `cell-ref` in the `begin` executes before the `cell-set!` completes, then the `join` in the `begin` expression is an error. If the `cell-set!` expression wins the race, then the example deadlocks. In fact, executions that contain the error may also deadlock, depending on the precise semantics of errors and exceptions in MUFL!.

11.2.2 An Operational Semantics for MUFL!

Given the subtleties introduced by concurrency, it is more important than ever to precisely specify the semantics of programs. Here we introduce a complete SOS for MUFL!. The SOS for MUFL! introduces two domains:

$T \in$ Thread-Handle $=$ Intlit
$A \in$ Agenda $=$ Thread-Handle $\rightarrow$ MixedExp

A thread handle is just an integer literal that serves as a unique identifier for threads. An agenda is a partial function that maps a thread handle to an element of MixedExp, the domain of intermediate expressions. We assume that the grammar defining ValueExp for FLK! is extended to include a new intermediate form, `(*thread*` T`)`, that represents a first class thread handle value.

$V \in$ ValueExp
$V ::= \ldots$ [Value Expressions]
| `(*thread*` T`)` [Thread Values]

Because an agenda is a partial function, it may be defined only on a subset of thread handles. *dom(A)* is the notation for the set of inputs on which A is defined. We will use the notation $A_\emptyset$ to stand for an agenda that is nowhere defined and the notation $A[T = M]$ to stand for an agenda that maps T to M and maps every other T' in *dom(A)* to $(A\ T')$. These notations are useful for any partial function; in fact, we shall use them for stores as well as agendas.

Suppose that T_{top} is a distinguished thread handle for top-level MUFL! expressions. Then an SOS for the MUFLK!, the MUFL! kernel, is

$$\langle CF, \Rightarrow, FC, IF, OF \rangle$$

where

$$\begin{aligned}
CF &= \text{Agenda} \times \text{Store} \\
FC &= \langle A[T_{top} = V], S \rangle \\
IF &= \lambda E \,.\, \langle A_{\emptyset}[T_{top} = E], S_{\emptyset} \rangle \\
OF &= \lambda \langle A[T_{top} = V], S \rangle \,.\, (output\ \ V) \\
(output\ \texttt{(*thread* T)}) &= \texttt{thread} \\
(output\ \ V) &= (output_{FLK!}\ \ V) \text{ where } V \neq \texttt{(*thread*}\ T\texttt{)}
\end{aligned}$$

In a configuration, the agenda keeps track of the evaluation of the expression associated with each thread handle, while the store manages cell states. The input function maps a top-level expression E_{top} to a configuration with an empty store and an agenda whose only thread T_{top} evaluates E_{top}. The transition rules rewrite the configuration until the top-level expression is evaluated or no more rules are applicable. If the top-level expression has been rewritten to a value expression by this point, then a representation of this value is returned. If the top-level expression is not a value and no more progress is possible, the computation is deadlocked.

The transition rules for the MUFLK! SOS are summarized in Figure 11.1.

Each rule allows the one-step progress of the expression associated with a single thread handle. In any configuration, at most one transition is possible at a given thread handle. However, transitions may be possible at several thread handles within one configuration, so the rules are non-deterministic.

The [*fork*] rule allocates a new thread handle for the body of the `fork` and returns it to the thread containing the `fork` expression. `fork` is the only means by which threads are created; there is no mechanism for destroying threads (removing them from the agenda). The [*join*] rule indicates that the `join` of a thread handle T' cannot proceed until the expression associated with T' has progressed to a value. `join` is treated as a primop so that it is not necessary to specify a progress rule for evaluating its operand. Similarly, `thread?` is handled as a primop that determines whether its argument is a thread handle.

Figure 11.1 also presents two **meta-rules** for deriving MUFLK! rewrite rules from the rewrite rules for FLK!. The [*FLK!-axiom*] meta-rule says that any axiom for rewriting FLK! expressions can be applied to the expression of a single MUFLK! thread. The [*FLK!-progress*] meta-rule similarly specifies how

REWRITE RULES

$$\langle A[T = \texttt{(fork } E\texttt{)}], S\rangle \Rightarrow \langle A[T = \texttt{(*thread* } T'\texttt{)}][T' = E], S\rangle, \quad \text{where } T' \notin dom(A) \qquad [fork]$$

$$\langle A[T = \texttt{(primop join (*thread* } T'\texttt{))}][T' = V], S\rangle \Rightarrow \langle A[T = V][T' = V], S\rangle \qquad [join]$$

$$\langle A[T = \texttt{(primop thread? (*thread* } T'\texttt{))}], S\rangle \Rightarrow \langle A[T = \texttt{\#t}], S\rangle \qquad [thread?\text{-}true]$$

$$\langle A[T = \texttt{(primop thread? } V\texttt{)}], S\rangle \Rightarrow \langle A[T = \texttt{\#f}], S\rangle, \quad \text{where } V \text{ is not of the form (*thread* } T'\text{)} \qquad [thread?\text{-}false]$$

META-RULES

For each axiom
$\langle E, S\rangle \Rightarrow \langle E', S'\rangle$
in the FLK! SOS, include the following axiom in the MUFLK! SOS:
$\langle A[T = E], S\rangle \Rightarrow \langle A[T = E'], S'\rangle$
[FLK!-*axiom*]

If X is an FLK expression context and $X\{E\}$ is the result of filling the hole of the context with E, then for each FLK! progress rule of the form

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle X\{E\}, S\rangle \Rightarrow \langle X\{E'\}, S'\rangle} \qquad [FLK!\text{-}progress]$$

include the following progress rule in the MUFLK! SOS:

$$\frac{\langle A[T = E], S\rangle \Rightarrow \langle A'[T = E'], S'\rangle}{\langle A[T = X\{E\}], S\rangle \Rightarrow \langle A'[T = X\{E'\}], S\rangle}$$

Figure 11.1: Rewrite rules and meta-rules for MUFLK!.

to derive MUFLK! progress rules from FLK! progress rules. Note that the [*FLK!-progress*] meta-rule uses two agenda meta-variables, A and A'. This is necessary for handling antecedent transitions in which an inner `fork` extends the agenda with a new thread. If instead A were used on both sides of the antecedent transition, the effect of a `fork` nested within an expression could never be propagated.

For deterministic languages like PostFix and FL!, behavior is defined as the partial function that maps a program to the unique result determined by the operational semantics. But in the presence of non-determinism, there may be more than one result. So it is necessary to extend the notion of behavior to be a *relation* between programs and results. Equivalently, we can define behavior as a total function that maps programs to the powerset of results. In this approach, behavior maps a program to the set of all the results that can be determined for the program via the operational semantics.

As an example of the non-determinism of MUFL! programs, consider the following expression:

```
(let ((c (cell 0)))
  (let ((t (fork (cell-set! c 1))))
    (begin (cell-set! c 2)
           (join t)
           (cell-ref c))))
```

The configuration representing the state of the computation after the allocation of the cell and the thread handle is:

$\langle A_\emptyset[T_{top}=$ `(begin (cell-set! (*cell*` L_1`) 2)`
`(join (*thread*` T_1`))`
`(cell-ref (*cell*` L_1`)))]`
$[T_1=$`(cell-set! (*cell*` L_1`) 1)`$]$,
$S_\emptyset[L_1=0]\rangle$

(For convenience, we are being somewhat loose in our notation, allowing threads to name expressions from the full MUFL! language rather than just kernel expressions.) Figures 11.2 and 11.3 show two possible transition paths that can be taken from this configuration. The first computes a final value of `1`, while the second computes a final value of `2`. Since these are the only two possible results, the behavior of the expression is $\{1, 2\}$.

11.2.3 Other Thread Interfaces

The `fork`/`join` mechanism introduced above is only one interface to threads. Here we discuss some other approaches.

TRANSITION PATH 1:

$\langle A_\emptyset\ [T_{top}=$ `(begin (cell-set! (*cell* `L_1`) 2)`
`(join (*thread* `T_1`))`
`(cell-ref (*cell* `L_1`)))]`
$[T_1=$`(cell-set! (*cell* `L_1`) 1)`$]$,
$S_\emptyset[L_1=0]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset\ [T_{top}=$ `(begin (join (*thread* `T_1`))`
`(cell-ref (*cell* `L_1`)))]`
$[T_1=$`(cell-set! (*cell* `L_1`) 1)`$]$,
$S_\emptyset[L_1=2]$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset\ [T_{top}=$ `(begin (join (*thread* `T_1`)) (cell-ref (*cell* `L_1`)))]
$[T_1=$ `#u`$]$,
$S_\emptyset[L_1=1]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top}=$ `(cell-ref (*cell* `L_1`))`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top}=$ `1`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle$

$(OF\ \langle A_\emptyset[T_{top}=$ `1`$][T_1=$ `#u`$], S_\emptyset[L_1=$ `1`$]\rangle) = 1$

Figure 11.2: Sample transition paths demonstrating the non-determinism of MUFL! programs, Part I.

TRANSITION PATH 2:

$\langle A_\emptyset$ [T_{top}= `(begin (cell-set! (*cell* L1) 2)`
`(join (*thread* T1))`
`(cell-ref (*cell* L1)))`]
[T_1=`(cell-set! (*cell* L1) 1)`],
$S_\emptyset[L_1 = 0]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset$ [T_{top}= `(begin (cell-set! (*cell* L1) 2)`
`(join (*thread* T1))`
`(cell-ref (*cell* L1)))`]
[T_1 = `#u`],
$S_\emptyset[L_1 = 1]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset$ [T_{top}= `(begin (join (*thread* T1)) (cell-ref (*cell* L1)))`]
[T_1 = `#u`],
$S_\emptyset[L_1 = 2]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top} =$ `(cell-ref (*cell* L1))`$][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle$

$\stackrel{*}{\Rightarrow}\langle A_\emptyset[T_{top} = 2][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle$

$(OF\ \langle A_\emptyset[T_{top} = 2][T_1 =$ `#u`$], S_\emptyset[L_1 = 2]\rangle) = 2$

Figure 11.3: Sample transition paths demonstrating the non-determinism of MUFL! programs, Part II.

11.2.3.1 Other thread operations

Many concurrent languages support variants of `fork` and `join`. In some languages, a thread can be `join`ed only once. In such languages, `join` has the effect of destroying the thread from the agenda as well as synchronizing, and joining a thread more than once is an error.

Some languages support other thread operations, such as destroying the thread, suspending it (temporarily removing it from the agenda), and resuming it (adding it back to the agenda). Sometimes there is a parent/child relationship between the thread that contains the `fork` and the resulting thread. In this case, thread operations on a parent can affect the children and vice versa. For example, destroying a parent thread might also destroy all its descendants.

11.2.3.2 Thread Abstractions

There are many useful thread abstractions that can be built on top of `fork` and `join`. For example, it is common to use `fork` and `join` to evaluate a set of expressions concurrently and then manipulate their results when all are finished. This idiom is captured in the `cobegin` construct, which is defined by the following desugaring:

```
(cobegin E1 ... En)
 = (let ((I1 (fork E1))
         ⋮
         (In (fork En)))
     (list (join I1) ... (join In)))
```

This version of `cobegin` returns a list of results, but other options are to return the first value, last value, or the unit value.

A variant on `cobegin` is a `colet` construct that binds names to the results of expressions that are computed concurrently:

```
(colet ((I1 E1) ... (In En)) Ebody)
 = (let ((I1 (fork E1))
         ⋮
         (In (fork En)))
     (let ((I1 (join I1))
           ⋮
           (In (join In)))
       Ebody))
```

Using `colet`, the concurrent tree-sum example can be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (colet ((left-sum (tree-sum (left-branch tree)))
                (right-sum (tree-sum (right-branch tree))))
          (+ left-sum right-sum)))))
```

11.2.3.3 Futures

A **future** is a thread that is implicitly `join`ed in any context that needs to examine its value. (In the context of futures, `join` is usually called `touch`.) Examples of such contexts are operands of strict primitives, the test position of a conditional, and the operator position of a `call`. Futures are supported in a number of LISP dialects [Hal85, Mil87, For91].

Futures are created by the construct `(future` E`)`. Using futures, the concurrent tree summation program can be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (+ (future (tree-sum (left-branch tree)))
           (future (tree-sum (right-branch tree)))))))
```

Futures are more modular than `fork/join` because it is possible to sprinkle `future`s into a program without having to guarantee that `join`s are placed wherever a future might be used. However, in a straightforward implementation, every touching context must check whether or not a value is a future, and, if it is, touch it. This overhead is similar to that incurred by memoization in lazy evaluation.

11.2.3.4 Eager Evaluation

The similarity between futures and lazy evaluation suggests a parameter passing mechanism in which every argument is implicitly wrapped in a `future`. This mechanism is called **eager evaluation**. We will refer to it as **call-by-eager** (CBE) by analogy with call-by-lazy. An example of a language supporting CBE is ID, a mostly functional language designed to be run on parallel computers [AN89].

Under CBE, an argument is evaluated concurrently with other arguments and the body of the procedure. In contrast with CBL, which only evaluates an

argument if it is used in the body, CBE evaluates the argument whether or not it is needed.

In a version of MUFL! with CBE parameter passing, the concurrent tree summation program would be expressed as:

```
(define tree-sum
  (lambda (tree)
    (if (leaf? n)
        (leaf-value n)
        (+ (tree-sum (left-branch tree))
           (tree-sum (right-branch tree))))))
```

Note that this program is indistinguishable from a program in CBV FL. CBE removes inessential constraints governing evaluation order of expressions and emphasizes that the only fundamental constraints are induced by data dependencies.

An interesting feature of CBE is that a procedure may return before its arguments have been evaluated. For example, suppose the following expression is evaluated under CBE:

```
(begin (call (lambda (x) (display 'b))
             (display 'a))
       (display 'c))
```

Assuming `display` does not return until the output operation is successful, the possible displayed outputs of this expression are `abc`, `bac`, and `bca`. In each case, the fact that the procedure body must be fully evaluated before the `call` returns forces `b` to be displayed before `c`. However, because the argument `x` is never used within the body of the `lambda`, the `a` can be displayed at any time, even after the procedure has returned.

11.3 Communication and Synchronization

Concurrent programs are often patterned after interacting entities in the real world. Completely independent threads are inadequate for modelling the interactions between such entities. Here we explore various ways in which threads may interact. We focus on two kinds of interaction:

- **Communication**: Information computed by one thread often needs to be transmitted to another thread.
- **Synchronization**: When concurrently executing threads share state, their accesses and updates must often be carefully choreographed in order to achieve a desired effect.

11.3.1 Shared Mutable Data

Shared mutable data structures (like mutable cells and mutable variables) are the most obvious mechanisms for inter-thread communication, but they are also the ones with the most pitfalls. For example, suppose we want to define a counter that can be incremented by several threads. A straightforward approach is to define a cell and an associated incrementing procedure:

```
(define counter (cell 0))

(define increment!
  (lambda ()
    (begin (cell-set! counter (+ 1 (cell-ref counter)))
           (cell-ref counter))))
```

But then concurrent threads can interact in some nasty ways through the shared cell. For instance, what are the possible values of a call to the following MUFL! procedure?

```
(define test-increment!
  (lambda ()
    (begin
      (cell-set! counter 0)
      (colet ((a (increment!))
              (b (increment!)))
        (+ a b)))))
```

If one call to `increment!` executes to completion before the other begins, the value of this expression is `3`. But other results are possible because the execution of the two calls to `increment!` may be interleaved. The expression can return a `2` if the first `cell-ref` within each `increment!` executes before either `cell-set!`, and can return a `4` if both `cell-set!`s complete before either of the second `cell-ref`s within `increment!` are performed.

The problem here is that concurrency allows the internal operations of two different calls to `increment!` to be interleaved. We often want procedures like `increment!` to be **atomic** in the sense that they appear to execute indivisibly with respect to other computations that manipulate the same shared state. Without some means of guaranteeing atomicity, mutable data is not generally a reliable communication mechanism between threads. Even simple primitives like `cell-set!` and `cell-ref` may not be atomic, further complicating reasoning about threads that interact via shared mutable data.

11.3.2 Locks

The goal of atomicity is to constrain concurrency to avoid undesirable interleaving. Atomicity can be achieved by introducing a **lock** that guards access to a logical collection of shared mutable data elements. Typically, a lock can be "owned" by a single thread in such a way that another thread cannot acquire the lock until the owning thread determines it is safe to release it. A lock is like a lockable door to a room that can be used by only one person at a time. When no one is inside, the door is open to allow the next person to enter. But upon entering the room, an individual locks the door and only unlocks it upon leaving.

There are numerous locking schemes for concurrent languages. Here we extend MUFL! with a simple locking mechanism that has the following interface:

`(lock)`: Return a new lock that is not owned by any thread.

`(acquire!` *E*`)`: If the lock *l* computed by *E* is not owned by a thread, acquire possession of the lock. If *l* is already owned, wait until it is not owned before acquiring it. It is an error if *E* does not evaluate to a lock.

`(release!` *E*`)`: If the lock *l* computed by *E* is owned by the current thread, then release possession of the lock. It is an error if the current thread does not own the lock or if *E* does not evaluate to a lock.

`(lock?` *E*`)`: Determine whether the value of *E* is a lock.

All of these constructs can be regarded as new primitives rather than special forms.

Using a lock, we can implement an atomic version of `increment!`:

```
(define counter (cell 0))
(define counter-lock (lock))

(define increment!
  (lambda ()
    (begin (acquire! counter-lock)
           (cell-set! counter (+ 1 (cell-ref counter)))
           (let ((ans (cell-ref counter)))
             (begin (release! counter-lock)
                    ans)))))
```

Since only one thread can own `counter-lock` at any time, the operations of two calls to `increment!` running in separate threads cannot be interleaved. Thus, the new version of `increment!` guarantees that `(test-increment!)` will return `3`.

If a thread "forgets" to release a lock that it owns, or if it releases it at the wrong time, it is easy to get unintentional deadlocks. For this reason, it is wise to build some abstractions on top of locks. For example, the following `with-lock` procedure takes a lock and a thunk, and executes the thunk while holding the lock. A result is returned only when the lock is released:

```
(define with-lock
  (lambda (lock thunk)
    (begin (acquire! lock)
           (let ((ans (thunk)))
             (begin (release! lock)
                    ans)))))
```

Using `with-lock`, the atomic version of `increment!` can be implemented as:

```
(define increment!
  (lambda ()
    (with-lock counter-lock
      (lambda ()
        (begin
          (cell-set! counter (+ 1 (cell-ref counter)))
          (cell-ref counter))))))
```

If we want multiple incrementers, it is easy to bundle a lock together with each cell:

```
(define make-incrementer
  (lambda ()
    (let ((c (cell 0))
          (l (lock)))
      (lambda (msg)
        (cond ((eq? msg 'increment!)
               (with-lock l
                 (lambda ()
                   (begin
                     (cell-set! c (+ 1 (cell-ref c)))
                     (cell-ref c)))))
              ((eq? msg 'reset!)
               (with-lock l
                 (lambda ()
                   ;; Don't assume CELL-SET! is atomic
                   (cell-set! c 0))))
              (else (error unknown-message)))))))
```

Because each incrementer is equipped with its own lock, the incrementing operations of two separate incrementers can be interleaved, but the incrementing

operations of two `increment!`s to the same incrementer cannot be interleaved.

Figure 11.4 presents the rewrite rules for the lock constructs. We assume that the domain MixedExp of intermediate expressions and the domain ValueExp of value expressions have been extended with the form `(*lock*` L`)`, which represents a first-class lock value. The location in this form contains either `#f`

$$\langle A[T = \texttt{(primop lock)}], S\rangle \Rightarrow \langle A[T = \texttt{(*lock*}\ L\texttt{)}], S[L = \texttt{\#f}]\rangle, \quad \text{where } L \notin dom(S) \qquad [lock]$$

$$\langle A[T = \texttt{(primop acquire! (*lock*}\ L\texttt{))}], S[L = \texttt{\#f}]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = \texttt{(*thread*}\ T\texttt{)}]\rangle \qquad [acquire!]$$

$$\langle A[T = \texttt{(primop release! (*lock*}\ L\texttt{))}], S[L = \texttt{(*thread*}\ T\texttt{)}]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = \texttt{\#f}]\rangle \qquad [release!]$$

[*lock?-true*] and [*lock?-false*] as usual for predicates.

Figure 11.4: The semantics of locks.

(indicating that the lock is currently not owned by a thread) or a thread handle (indicating which thread owns the lock). Since a lock can only be acquired when the associated location contains `#f` and acquiring the lock mutates the location, a lock can be owned by at most one thread.

11.3.3 Channels

It is tedious to use cells and locks for all instances of communication and synchronization. Sometimes higher order abstractions are better suited for the task. One very useful abstraction is the **channel**, a conduit through which values may be communicated. A channel is a First-In/First-Out (FIFO) queue that can be accessed by multiple threads. A thread can **send!** a value to a channel (enqueue it) or **receive!** a value from a channel (dequeue the first value from the channel). Channels have the following synchronization feature built in: an attempt to receive a value from an empty channel blocks a thread until a value is sent.

Here is a procedural interface to channels (all of these can be primitives):

`(channel)`: Create and return a new channel.

`(send!` E_{chan} E_{val}`)`: Send the value computed by E_{val} over the channel computed by E_{chan} and return `#u`. It is an error if the value of E_{chan} is not a channel.

`(receive!` E_{chan}`)`: Receive and return the next value from the channel computed by E_{chan}. It is an error if the value of E_{chan} is not a channel.

`(channel?` E`)`: Determine whether the value of E is a channel.

Note that several threads may be sending to, and several threads may be receiving from, the same channel. It is even possible for a single thread to send and receive on the same channel.

Channels are an alternative to streams for programming in the "signal processing style." Figure 11.5 shows how to encode stream-like operations with channels. For example, we can use such operations to find the sum of the squares of the prime numbers between 0 and 100 as follows:

```
(let ((c1 (channel))
      (c2 (channel))
      (c3 (channel)))
  (colet ((ignore1 (gen-channel 0
                     (lambda (x) (+ x 1))
                     (lambda (x) (> x 100))
                     c1))
          (ignore2 (filter-channel prime? c1 c2))
          (ignore3 (map-channel (lambda (x) (* x x)) c2 c3))
          (ans     (accum-channel 0 + c3)))
    ans))
```

Figure 11.6 presents an operational semantics for channels. We assume that the domain MixedExp of intermediate expressions and the domain ValueExp of value expressions have been extended with the form `(*channel*` L`)`, which represents a first-class channel value. We also assume that storable values are extended to include elements

$Q \in \text{Queue} = \text{ValueExp}^*$,

value sequences that implement the queue underlying a channel.

A few notes:

- The channels described above use unbounded queues. A **bounded buffer** is a channel that has an upper size limit. A sending thread blocks if the queue implementing the buffer is at its maximum size, and will unblock if the size decreases (due to `receive!`s).

- In some concurrent languages, there is no buffer associated with a channel (i.e., the queue size is effectively zero). Instead, a channel defines a **rendezvous** point between a sending thread and a receiving thread. The first thread to reach the rendezvous point must wait for the complementary thread to arrive. When both threads are at the rendezvous point, a

```
(define eoc '*end-of-channel*)

(define (eoc? s) (sym=? s eoc))

(define (gen-channel first next done? out)
  (if (done? first)
      (send! out eoc)
      (begin (send! out first)
             (gen-channel (next first) next done? out))))

(define (map-channel proc in out)
  (let ((val (receive! in)))
    (if (eoc? val)
        (send! out eoc)
        (begin (send! out (proc val))
               (map-channel proc in out)))))

(define (filter-channel pred in out)
  (let ((val (receive! in)))
    (if (eoc? val)
        (send! out eoc)
        (begin (if (pred val) (send! out val) #u)
               (filter-channel pred in out)))))

(define (accum-channel null combine in)
  (letrec ((loop (lambda (ans)
                   (let ((val (receive! in)))
                         (if (eoc? val)
                             ans
                             (loop (combine val ans))))))))
    (loop null)))
```

Figure 11.5: Signal processing style procedures implemented with channels.

$$\langle A[T = \texttt{(primop channel)}], S\rangle \Rightarrow \langle \texttt{agenda}[T = \texttt{(*channel*}\ L\texttt{)}], S[L = [\,]]\rangle \quad [\textit{channel}]$$

$$\langle A[T = \texttt{(primop send! (*channel*}\ L\texttt{)}\ V\texttt{)}], S[L = Q]\rangle \Rightarrow \langle A[T = \texttt{\#u}], S[L = [V]\ @\ Q]\rangle \quad [\textit{send!}]$$

$$\langle A[T = \texttt{(primop receive! (*channel*}\ L\texttt{))}], S[L = Q\ @\ [V]]\rangle \Rightarrow \langle A[T = V], S[L = Q]\rangle \quad [\textit{receive!}]$$

[*channel?-true*] and [*channel?-false*] as usual for predicates.

Figure 11.6: The semantics of channels.

value is communicated from sender to receiver, after which both threads proceed.

- In several concurrent process languages (e.g., Hoare's CSP and Milner's CCS), channels are not first-class values, but are names that must match up between sending and receiving threads. There are renaming operators that permit code written in terms of one channel name to use another.
- Some languages support a kind of write-once channel called a **single-assignment cell**. Single-assignment cells have the following properties:
  - A cell has no initial value.
  - A read of the cell blocks until the cell has a value.
  - An attempt to write to a cell that has already been written is an error.

  Id's I-structures are arrays of such cells. Variables in some logic programming languages are similar to single-assignment cells.

Reading

References:

- Classic: Andrew Birrel on threads [Bir89], Melvin Conway coroutines [Con63], E. W. Dijkstra semaphores [Dij68], C. A. R. Hoare monitors [Hoa74], P. Brinch-Hansen monitors [Bri77].

- Languages: Concurrent ML ([CM90]), MULTI-LISP([Hal85]) , MULTI-SCHEME([Mil87]), ID I-structures ([ANP89]) and M-Structures ([Bar92b]), LINDA([CG89]), Concurrent Objects ([DF96])
- Denotational Semantics: Schmidt on resumption semantics [Sch86a].
- Process Algebras: [Hoa85], [Mil89]. ATP, Pi-Calculus, Pratt's Pomset model, trace semantics, Milner's overview paper on bi-simulation.
- Data Parallel: [HS86], [Ble92], [Ble90], [Sab88], Connection Machine, Connection Machine Lisp.