Chapter 7

OPERA: Controlling Operational Behavior

As noted in the exposition of Chapter 5, the sliver technique depends crucially on three language features: concurrency, synchronization, and non-strictness. Concurrency permits the processing of the slivers to be interleaved, synchronization guarantees lock step processing, and non-strictness mechanisms allow fine-grained control over evaluation order in an otherwise strict language. In this chapter, we present OPERA, a particular language that supports these features, and show how they can be used to implement slivers and synchronized lazy aggregates. OPERA is a dialect of Scheme that combines classical mechanisms for concurrency, mutual exclusion, laziness, and eagerness with a novel form of barrier synchronization based on the *synchron*, a new first-class synchronization object.

We begin with an informal introduction to the OPERA language (a formal semantics is deferred until Chapter 8). Then we explore how the features of OPERA can be used to implement the SYNAPSE slivers and slags illustrated in Chapter 6.

7.1 An Introduction to OPERA

OPERA is a dialect of Scheme [ASS85, CR+91] that supports various mechanisms for controlling the fine-grained operational details of programs. Figure 7.1 summarizes the syntax of the language and the primitive operations it supports. Both syntactically and semantically,

Kernel Grammar:

P	$\in$	Program
E	$\in$	Expression
I	$\in$	Identifier
L	$\in$	Literal
S	$\in$	Symbolic Expression

P	::=	`(program` E_{body} `(define` I_{name} E_{def}`)*)`	[program]
E	::=	L	[literal expression]
	|	I	[variable reference]
	|	`(lambda (`I_{formal}`*)` E_{body}`)`	[abstraction]
	|	`(pcall` E_{proc} E_{arg}`*)`	[parallel application]
	|	`(pletrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[parallel recursive bindings]
	|	`(if` E_{test} E_{then} E_{else}`)`	[conditional]
	|	`(set!` I_{name} E_{val}`)`	[assignment]
	|	`(quote` S_{text}`)`	[quoted expressions]
	|	`(lazon` E_{body}`)`	[suspension]
	|	`(eagon` E_{body}`)`	[future]
	|	`(exclusive` E_{excl} E_{body}`)`	[mutual exclusion]
	|	`(nex ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical bindings]
	|	`(nexrec ((`I_{name} E_{def}`)*)` E_{body}`)`	[graphical recursive bindings]
I	::=	*usual Scheme identifiers*	
L	::=	*usual Scheme literals*	
S	::=	*usual Scheme quotable expressions*	

Syntactic Sugar Grammar:

E_{sugar} ::= `(cond (`E_{test} E_{action}`*)*)` | `(begin` E_{sequent}`*)` | ... *usual Scheme sugar*
| `(`E_{proc} E_{args}`*)` | `(let ((`I_{name} E_{def}`)*)` E_{body}`)` | `(letrec ((`I_{name} E_{def}`)*)` E_{body}`)`
| `(plet ((`I_{name} E_{def}`)*)` E_{body}`)` | `(seq` E`*)` | `(seq1` E`*)` | `(seqn` E`*)`

Primitives:

OPERA supports the usual Scheme primitive operations on numbers, booleans, procedures, symbols, characters, strings, pairs, vectors, and ports, but it does not support continuations. In addition, it also supports the following objects and operations:

Synchrons: `(synchron)`, `(wait` E_{sync}`)`, `(simul!` E_{sync1} E_{sync2}`)`, `(precede!` E_{before} E_{after}`)`, `(synchron?` E_{obj}`)`

Excludons: `(excludon)`, `(excludon?` E_{obj}`)`

Cells: `(cell` E_{val}`)`, `(cell-ref` E_{cell}`)`, `(cell-set!` E_{cell} E_{new}`)`, `(cell?` E_{obj}`)`

Figure 7.1: OPERA summary

OPERA is a close cousin of Scheme. The key differences are as follows:

- *Concurrency*: OPERA supports two forms of concurrency:

    1. concurrent evaluation of the subexpressions of a procedure call expression.
    2. eager evaluation via the `eagon` special form.

  Here we briefly describe the concurrent evaluation strategy for procedure calls; eagons are described in the bulleted item on non-strictness.

  Both OPERA and Scheme have strict procedure calls, but they use different strategies for evaluating the subexpressions of a procedure application expression: OPERA evaluates the subexpressions in parallel, while Scheme requires that the subexpressions be evaluated in a way that is consistent with some sequential order. The explicit `pcall` (*p*arallel *call*) keyword in OPERA's kernel application form is intended to emphasize this difference. However, since the explicit keyword is cumbersome, we will abbreviate `(pcall` E_{proc} E_{arg}`*)` as the more familiar `(`E_{proc} E_{arg}`*)`. OPERA's concurrent evaluation strategy extends to the bindings of `plet` and `pletrec`, parallel versions of Scheme's `let` and `letrec`. To simplify comparisons between the languages, OPERA treats `let` and `letrec` as synonyms for `plet` and `pletrec`.

  While OPERA programs are designed to look like Scheme programs, OPERA's concurrent evaluation strategy can lead to behaviors that would not be exhibited by a Scheme interpreter. In particular, the concurrency can give rise to new kinds of non-determinism. For this reason, it is important to clearly specify whether a program in Scheme syntax is to be evaluated with respect to OPERA or to Scheme.

  We emphasize that OPERA supports concurrency purely for the purpose of modularity, not for efficiency. Multi-tasking on a single physical processor is a sufficient form of concurrency for our purposes.

- *Synchronization*: OPERA supports two first-class synchronization objects: synchrons and excludons. Synchrons are the key to the lock step processing model on which slivers and synchronized lazy aggregates are based. They provide barrier synchronization in a system where the number of processes participating in a rendezvous is determined dynamically.

Excludons are just standard lock objects [Bir89b] that support mutual exclusion via the `exclusive` special form. Excludons are used to enforce atomic actions by eliminating the undesirable nondeterminism introduced by concurrency.

- *Non-strictness*: OPERA supports lazy and eager evaluation via the `(lazon` E_{body}`)` and `(eagon` E_{body}`)` special forms. Both forms create a placeholder object that stands for the result of evaluating E_{body}. In the case of lazons, evaluation of E_{body} is delayed until the lazon appears in a context that requires its value. Lazons are like the *delayed objects* described in [Hen80] and [ASS85] except that they are implicity forced rather than explicitly forced; we shall see how this difference enhances modularity. In the case of eagons, evaluation of E_{body} proceeds in parallel with the rest of the computation; any context requiring the value of E_{body} must wait until it is available. Eagons are the same as the *futures* used in various parallel Lisp implementations ([Hal85], [Mil87]), except that in OPERA we do not assume the existence of physical parallelism. We use the name "eagon" only for symmetry with "lazon".

- *Other differences*: OPERA supplies a few additional features that add no new power to the language but simplify the expression of some programs. OPERA introduces two new binding forms: `nex` and `nexrec`. These are versions of `let` and `letrec` that do not evaluate the binding expressions before evaluating the body. Instead, they resemble the "graphical" `let`s and `letrec`s of a lazy functional language (e.g., see [Pey87, pages 233-234]). In addition to the usual Scheme syntactic sugar, OPERA provides a few new derived special forms. There are three sequencing forms: `(seq` $E*$`)`, `(seq1` $E*$`)`, and `(seqn` $E*$`)`. All evaluate their subexpressions in left-to-right order, but they return different results: `seq` returns the boolean true value, `seq1` returns the value of the first subexpression, and `seqn` returns the value of the last subexpression (i.e., it is a synonym for `begin`). In addition to the usual Scheme datatypes and operations, OPERA supports mutable one-slot data structures called *cells*. These correspond to the reference structures of ML [MTH90] and FX [GJSO92].

OPERA does not support Scheme's continuations. In principle, OPERA should ultimately be able to support continuations, but the details of how continuations interact with OPERA's fine-grained concurrency have not yet been worked out.

The summary in Figure 7.1 gives the form of an OPERA program as:

$$\texttt{(program } E_{\text{body}} \texttt{ (define } I_{\text{name}} \ E_{\text{def}}\texttt{)*)}$$

In our OPERA examples below, we will not use the `program` form directly. Instead, in the usual Scheme style, we will treat every expression as a program by appropriately packaging it with all the definitions encountered so far.

The following sections explore the features of OPERA in more depth.

7.1.1 Strict Procedure Calls

As in Scheme, the procedure call of OPERA is strict. This means that the evaluation of the procedure body does not begin until the evaluation of the arguments has completed. Because it provides natural barrier between the computation of the arguments and the computation of the body, OPERA's strict procedure call serves as a basic mechanism for controlling operational behavior. Lazons and eagons are mechanisms for overriding the strictness of a procedure call.

7.1.2 Concurrent Evaluation

To permit the concurrent evaluation of slivers, OPERA embraces a concurrent evaluation strategy for the subexpressions of a procedure application. Synchrons and excludons are mechanisms for controlling the concurrency introduced by this strategy.

In this context, "concurrent" does not imply parallel execution of a program on multiple physical processors. It merely indicates a partial ordering on the operations performed in the evaluation of the subexpressions of a procedure call. That is, the evaluation steps of a procedure call's subexpressions may be arbitrarily interleaved. (A precise model of the allowed interleaving is provided in Chapter 8.) Whether the interleaving results from multitasking on a single processor or from evaluating the subexpressions on different processors is irrelevant for our purposes. We use concurrency not to make programs run faster, but to make them more modular. In fact, this work clearly supports the notion that concurrency is an important modularity mechanism. Although orchestrating a program on a parallel machine fits into the general theme of controlling the operational behavior of modular programs, it is beyond the scope of this work.[1]

As indicated in the overview, the concurrent evaluation strategy is a fundamental feature distinguishing OPERA from Scheme. To compare the languages, consider the following definitions:

```
(define (par a b) #t)

(define (test-order)
  (par (par (display 1)
            (display 2))
       (par (display 3)
            (display 4))))
```

The `par` procedure ignores the values of its two arguments and returns the boolean *true* value (`#t`). In both Scheme and OPERA, argument expressions are always evaluated regardless of whether their values are actually used by the procedure. So (`par` E_1 E_2) has the effect of evaluating both E_1 and E_2. `Test-order` is a procedure that tests argument evaluation order by using `par` in conjunction with `display`, a printing procedure.[2]

Because OPERA evaluates all application subexpressions in parallel, there are no constraints on the execution order of the `display`s within `test-order`. Evaluating `(test-order)` in OPERA may print the numerals `1` through `4` in any of the 24 possible orderings. In Scheme, on the other hand, the possible behaviors of this expression are more limited. Scheme evaluates all subexpressions of an application in some (unspecified) sequential order. This means that one of the arguments of `test-order`'s outermost `par` must be completely evaluated before evaluation of the other is begun. For example, `1` may be printed before both the `3` and `4` or after both the `3` and `4` but never between the `3` and `4`. Scheme allows only eight possible orderings for the numbers displayed by `(test-order)`:

[1] See [HS86b] for work along these lines.

[2] For simplicity, we will often illustrate issues of evaluation order with contrived examples that involve I/O or assignment. However, in practice, we are more interested in the relative order of purely functional operations. This order provides a window on the space consumed by pending operations.

```
1234, 1243, 2134, 2143, 3412, 3421, 4312, 4321
```

It is worth noting that Opera is not as concurrent as it could be. Although it exhibits concurrency in argument evaluation, there is no concurrency between the evaluation of a procedure's arguments and the evaluation of the procedure's body. So Opera, like Scheme, is a *strict* language — i.e., it requires all arguments to be evaluated before the body is evaluated. For example, consider the following:

```
(define (ignore a) (display 2))

(define (test-strict) (ignore (display 1)))
```

The call `(test-strict)` is guaranteed to print `1` before `2` in both Opera and Scheme. Strictness effectively defines a barrier between the evaluation of the arguments and the evaluation of the body. This barrier is crucial for controlling the behavior of Opera programs. We'll see shortly how non-strict behavior can be obtained in Opera with lazons, and eagons.

Concurrent evaluation and strictness characterize Opera's `let` (= `plet`) and `letrec` (= `pletrec`) constructs as well. That is, in both cases the binding expressions are evaluated in parallel, but all of these must be completely evaluated before the body is evaluated.

A final subtlety is the extent to which a language implementation must exhibit permitted evaluation orderings in practice. For example, a particular Scheme implementation is *not* required to exhibit all eight of the possible orderings for the `(test-order)` example. In fact, a typical Scheme interpreter follows a fixed ordering that is consistent with the "some sequential order" rule. The rule is more often exploited by Scheme compilers, which can permute the argument evaluation order as convenient. In fact, a Scheme implementation may actually interleave evaluation of arguments as long as it can prove that the resulting behavior is consistent with the "some sequential order" rule (e.g., when the argument expressions have no side effects).

What about Opera? In the presence of synchronization mechanisms like synchrons and excludons, artificially limiting the range of evaluation orders can prevent a program from terminating with an expected result via one of the permitted orderings. So any restricted set of evaluation orders is generally unacceptable. Nevertheless, an implementation may restrict the evaluation order in contexts where it can prove that the restriction will not affect normal termination of a program.

7.1.3 Synchrons

Concurrency alone does not guarantee that slivers networks will exhibit desirable space profiles and operation orderings. After all, a producing sliver might race ahead of one of its consumers, requiring the generated elements to be buffered. To avoid this situation, the sliver technique is based on a lock step processing model. Lock step processing is achieved via synchrons.

Synchronization Model

A *synchron* is a first-class object that represents a point in time. Computational events can be constrained to happen before or after this point in time, or left unordered with respect to it. Synchrons are used to express control in a concurrent system by limiting the allowable orderings between computational events. Intuitively, synchrons express a set of time constraints that are solved by Opera.

The lock step model of the sliver technique implies that slivers are concurrently executing recursive procedures that participate in a *barrier synchronization* at every corresponding call and return. In traditional barrier synchronization [Axe86], the number of participating processes is known in advance, and synchronization can easily be implemented by a counter. However, due to the dynamic configurability of sliver networks, the number of slivers participating in a barrier synchronization cannot generally be predicted from the text of the program. For example, the number of slivers created by the `pascal` procedure on page 243 depends on the values of its `column` argument.

Synchrons solve this problem by tying barrier synchronization to automatic storage management. Pointers to a synchron are classified into two types: *waiting* and *non-waiting*. When a process wishes to rendezvous at a synchron, it enters a waiting state that refers to the synchron with a distinguished waiting pointer. For all other manipulations, a process holds a synchron with a non-waiting pointer. *A rendezvous occurs at a synchron only when all of the pointers to it are waiting pointers.* This means that any non-waiting pointer to a synchron effectively blocks a rendezvous. The automatic storage manager is responsible for determining when a rendezvous has been achieved at a synchron and resuming the waiting processes once the rendezvous has occurred. Because all processes lose access to the synchron upon resumption, there can only be one rendezvous per synchron.

The rendezvous protocol of synchrons sets it apart from other synchronization structures (e.g., semaphores [Dij68], locks [Bir89b], synchronous messages [Hoa85], I-structures [ANP89], and M-structures [Bar92]). Synchronization typically involves some processes *waiting* in a suspended state for a shared synchronization entity to be *released* by the process that currently owns it. Traditional protocols supply explicit *wait* and *release* operations. With synchrons, only the *wait* is explicit; the *release* is implicitly handled by the automatic storage manager when a global rendezvous state is achieved. In this respect, synchrons resemble *weak pairs* [Mil87] and *populations* [RAM83], data structures whose behavior is tied to storage management.

Although developed independently, synchrons are closely related to Hughes's `synch` construct [Hug83, Hug84]. (See Section 3.1.5 for a discussion of `synch`.) Both `synch` and synchrons involve objects that define a rendezvous point. Hughes's objects serve as a rendezvous for exactly two requests for a value. It is possible to build a tree of `synch`s that act as a rendezvous point for a given number of requests, but these trees must be explicitly managed by the programmer. It is unclear how `synch`s could be used to handle a dynamically determined number of requests; I doubt that they can be used for this purpose. In contrast, synchrons serve as a rendezvous point for a dynamically determined number of processes. Because they automatically manage the barrier synchronization of an arbitrary number of processes, synchrons are superior to `synch` for building synchronization abstractions. Furthermore, the additional unification and precedence operations supported by synchrons (see below) are crucial for implementing important aspects of the lock step processing model.

Synchron Operations

Figure 7.2 summarizes the procedural interface to synchrons. The nullary `synchron` procedure creates a new synchron. Since synchrons are first-class objects, they can be named, passed as arguments to procedures, returned as results from procedures, and stored in data strucutures. The `synchron?` predicate tests whether an object is a synchron.

`(synchron)`
    Creates a new synchron.

`(synchron?` *obj*`)`
    Returns a boolean value indicating whether *obj* is a synchron.

`(wait` *sync*`)`
    Waits for a rendezvous at the synchron *sync*. Returns `#t` after the rendezvous has occurred.

`(simul!` *sync1 sync2*`)`
    Unifies synchrons *sync1* and *sync2* to be the same synchron. Returns the unified synchron.

`(precede!` *sync1 sync2*`)`
    Dictates that a rendezvous at the synchron *sync1* must occur before a rendezvous at the synchron *sync2*. Returns `#t`.

Figure 7.2: The procedural interface to synchrons.

The key operation on synchrons is the `wait` procedure. Calling `wait` on a synchron *sync* suspends the current computation until a rendezvous occurs at *sync*. After the rendezvous, all computations waiting on the synchron are resumed with the return of `#t` to each of the `wait` calls.

A `wait` call holds the argument synchron with a distinguished waiting pointer; all other references to the synchron are non-waiting pointers. The rendezvous occurs when the synchron is held only by waiting pointers. Since the return from `wait` drops all of the waiting pointers, a synchron is inaccessible after a rendezvous and its storage may be reclaimed.

The `simul!` procedure declares that the rendezvous of one synchron must occur simultaneously with the rendezvous of another synchron. Conceptually, `simul!` extends the notion of unifying logic variables [Rob65] to unifying the points in time represented by two synchrons. The `simul!` procedure forces its two argument synchrons to be equivalent (as determined by OPERA's `eq?` predicate); it returns the unified synchron as its result.

The `precede!` procedure declares that a rendezvous on one synchron must precede a rendezvous on another synchron. `Precede!` is an explicit method of imposing an ordering on the points in time represented by synchrons. (Synchrons can also be ordered implicitly by the context in which they are used.)

`Wait`, `simul!`, and `precede!` can be viewed as specifying a set of ordering constraints on the events of a computation. OPERA "solves" the set of constraints by dynamically finding an event ordering that satisfies the constraints. Sometimes the constraints are incompatible and no solution exists. For example, no ordering of events can satisfy the declaration that a synchron a precedes itself, or that a precedes b and b precedes a. If OPERA is forced to resolve an incompatible set of constraints, it halts in a *deadlock* state.

Synchron Examples

We illustrate the basics of synchrons through a series of simple examples. As in our discussion of the concurrent evaluation strategy, we will consider only examples involving the ordering of simple I/O operations. The discussion of the SYNAPSE implementation in Section 7.2 provides more realistic examples of synchron use.

We begin with the `test-order` procedure from Section 7.1.2:

```
(define (test-order_initial)
  (par (par (display 1)
            (display 2))
       (par (display 3)
            (display 4))))
```

In OPERA, (`test-order`$_{initial}$) prints out the numerals `1` through `4` in arbitrary order; there are 24 possibile orderings.

Suppose we want to specify the constraint that `3` must be printed before `2`. We can achieve this effect by rewriting the procedure as:

```
(define (test-order_reorganize)
  (par (display 1)
       (par (seq (display 3) (display 2))
            (display 4))))
```

The sequencing of `seq` implements the desired constraint.[3] However, this approach forces us to significantly reorganize the structure of the original procedure.

An alternate approach for expressing the constraint is to use a synchron:

```
(define (test-order_synchron:3->2)
  (let ((a (synchron)))
    (par (par (display 1)
              (seq (wait a) (display 2)))
         (par (seq (display 3) (wait a))
              (display 4)))))
```

The `(synchron)` call returns a new synchron object, which is named `a` in this example. The two `(wait a)` expressions inserted into the program implement the desired constraint by requiring `3` to be printed before the rendezvous on `a` and `2` to be printed after the rendezvous.

The advantage of the synchron approach over the approach taken in `test-order`$_{reorganize}$ is that the basic structure of the procedure (a `par` of two `par`s) is unchanged. The creation of the synchron and the insertions of the `seq`s and `wait`s are essentially annotations that have been added to the original structure. This annotation-like character of synchrons is used to preserve modularity while expressing control.

As another example, imagine constraining `test-order`$_{initial}$ to print `4` after both `1` and `3`. This is also straightforward to do with a synchron:

```
(define (test-order_synchron:1+3->4)
  (let ((b (synchron)))
    (par (par (seq (display 1) (wait b))
              (display 2))
         (par (seq (display 3) (wait b))
              (seq (wait b) (display 4))))))
```

Here, two events occur before the rendezvous at the synchron, and one occurs after. In general, an arbitrary number of `wait`s can be used to specify constraints between events. While the given constraints can also be expressed by using the right permutation of `par` and `seq`, the reorganized program would not carry with it any artifact of how it was derived by constraining a simpler program. In contrast, synchrons can be added and removed without altering the basic structure of the program.

[3] Recall that `seq` is an OPERA special form whose subexpressions are evaluated sequentially from left to right. Together, `par` and `seq` constitute a language for the series/parallel combination of expressions.

Another advantage of synchrons is that they facilitate the combination of timing constraints. Here's a single program that embodies the ordering constraints of the previous two examples:

```
(define (test-order_synchron:3->2&1+3->4)
  (let ((a (synchron))
        (b (synchron)))
    (par (par (seq (display 1) (wait b))
              (seq (wait a) (display 2)))
         (par (seq (display 3) (par (wait a) (wait b)))
              (seq (wait b) (display 4))))))
```

Note how the synchron annotations of the previous programs have essentially been superposed in this one. The program permits only five orderings of displayed numerals: `1324`, `1342`, `3214`, `3124`, and `3142`. It is impossible to express this set of orderings with any combination of `par`s and `seq`s. [4]

We will use the following `test-combiner` procedure to illustrate the different means of combining synchrons:

```
(define (test-combiner combiner)
  (let ((c (synchron))
        (d (synchron)))
    (seq (combiner c d)
         (par (seq (display 1) (seq (wait c) (display 2)))
              (seq (display 3) (seq (wait d) (display 4)))))))
```

`Test-combiner` first combines the synchrons `c` and `d` (by side effect), and then executes two processes: one of which prints `1` before `2` and the other of which prints `3` before `4`. Synchron `c` represents the point between the printing of `1` and `2`, while `d` represents the point between the printing of `3` and `4`.

We consider three combiners:

1. `(test-combiner (lambda (x y) 'ignore))` : Here the combiner is a trivial one that leaves `c` and `d` unconstrained. In this case there are six possible outputs:

   `1234, 1324, 1342, 3124, 3142, 3412`

[4] To be fair, neither `par`/`seq` nor synchrons are sufficient to express many orderings. For example, neither is sufficient to express the set $\{1234, 4321\}$, which requires some additional form of nondeterminism. But OPERA with synchrons can express more orderings than OPERA without synchrons.

2. `(test-combiner precede!)` : The `precede!` combiner forces the `c` rendezvous to occur before the `d` rendezvous. This has the effect of adding the constraint that the `1` be printed before the `4`. This filters out the `3412` option, leaving only five possible behaviors:

   `1234`, `1324`, `1342`, `3124`, `3142`

3. `(test-combiner simul!)` : The `simul!` combiner forces the `c` rendezvous and `d` rendezvous to occur simultaneously. This implies two additional constraints over the unconstrained version: `1` must be printed before `4`, and `3` must be printed before `2`. This leaves four possible behaviors:

   `1324`, `1342`, `3124`, `3142`

All of the above examples illustrate how synchrons can be used to constrain the order of operations within a single procedure body. But the real power of synchrons is that they can be used to constrain the order of operations across procedure boundaries. For example, consider the following procedure, which abstracts over the notion of inserting a synchronization point between two `display`s:

```
(define (make-displayer a b)
  (lambda (sync)
    (seq (display a) (seq (wait sync) (display b)))))
```

Using `make-displayer`, the three combination examples from above can be expressed in a modular fashion as follows:

1. No constraints – pass independent synchrons to two displayers:

   ```
   (par ((make-displayer 1 2) (synchron))
        ((make-displayer 3 4) (synchron))
   ```

2. `Precede!` constraint – pass two constrained synchrons to the two displayers:

   ```
   (let ((c (synchron))
         (d (synchron)))
     (seq (precede! c d)
          (par ((make-displayer 1 2) c)
               ((make-displayer 3 4) d))))
   ```

3. `Simul!` constraint – pass the same synchron to both displayers (no need to actually unify two separate synchrons via `simul!`):

```
(let ((c (synchron)))
  (par ((make-displayer 1 2) c)
       ((make-displayer 3 4) c)))
```

As a final example, we will illustrate how synchrons can be stored in data structures. This fact is especially important for implementing synchronized lazy aggregates. We will use the following procedure to test this feature:

```
(define (make-synchronized-displayer a b)
  (let ((sync (synchron)))
    (cons sync
          (lambda ()
            (seq (display a) (seq (wait sync) (display b)))))))
```

`Make-synchronized-displayer` is similar to the `make-displayer` procedure from above. However, rather than returning a synchron-accepting procedure, it returns a pair of a synchron and a thunk that refers to that synchron. The synchron is a hook for controlling some behavioral aspects of the associated thunk. Here is an expression that yields the behavior of the `simul!`-constrained versions of the previous examples (the other two constraints can be expressed similarly):

```
(let ((sync-disp1 (make-synchronized-displayer 1 2))
      (sync-disp2 (make-synchronized-displayer 3 4)))
  (seq (simul! (car sync-disp1) (car sync-disp2))
       (par ((cdr sync-disp1)) ((cdr sync-disp2)))))
```

This example underscores how independently generated synchronization points can be unified together to constrain the behavior of concurrently executing procedures. This is exactly the mechanism used to guarantee the lock step processing of slivers networks exhibiting fan-in.

Some Details

We conclude the overview of synchrons by mopping up a few details:

- *Deadlock*: None of the above examples illustrate deadlock, but it's easy to exhibit examples that do. Here are a number of expressions that give rise to deadlock:

```
;; Example 1
(let ((a (synchron)))
  (seq (wait a) (wait a)))


;; Example 2
(let ((a (synchron))
      (b (synchron)))
  (seq (unify! a b)
       (seq (wait a) (wait b))))


;; Example 3
(let ((a (synchron))
      (b (synchron)))
  (par (seq (wait a) (wait b))
       (seq (wait b) (wait a))))


;; Example 4
(let ((a (synchron))
      (b (synchron)))
  (seq (precede! b a)
       (seq (wait a) (wait b))))


;; Example 5
(let ((a (synchron)))
  (seq (wait a) a))


;; Example 6
(let ((a (sychron)))
  (if (wait a)
      17
      a))
```

Examples 1 and 2 deadlock because they require a rendezvous to happen before itself. Similarly, the deadlocks in examples 3 and 4 are due to cycles in the time ordering constraints between two synchrons. The expression in example 5 deadlocks because the second `a` in `(seq (wait a) a)` is a non-waiting reference to the synchron that prevents the rendezvous from proceeding; since the reference to the second `a` is not dropped until after `wait` successfully returns, there is a constraint cycle here as well. Example 6 is an extension to example 5. It is somewhat disconcerting that this expression deadlocks; because `wait` always returns `#t`, the `a` in the else branch of the `if` can never be reached anyway. A "sufficiently smart" implementation of Opera could determine this fact through static analysis. However, we will rely only on the semantics provided in Chapter 8, which dictate that example 6 must also deadlock.

A careful reader familiar with the Scheme environment model [ASS85] may wonder why a lot *more* examples don't deadlock. Consider the following (non-deadlocking) OPERA expression:

```
(let ((a (synchron))
      (b 17))
  (seq (wait a) (+ b 2)))
```

Under the traditional environment model, the expression `(seq (wait a) (+ b 2))` is evaluated in an environment in which `a` is bound (via a non-waiting pointer) to a synchron and `b` is bound to 17. Since the environment must be present for the evaluation of `(+ b 2)`, and it contains a non-waiting pointer to a synchron, this expression *would* deadlock under the traditional environment model. In the environment model, any expression that names a synchron could deadlock in this way; synchrons are rather useless in such a model.

The reason that such expressions *don't* deadlock is that OPERA is defined in terms of a model of evaluation that aggressively drops spurious references. Since the binding of `a` is not required to evaluate `(+ b 2)`, the non-waiting pointer to the synchron is effectively removed from the environment after the `a` in `(wait a)` is evaluated. The semantics in Chapter 8 spells out how this is accomplished.

- *Blocking behavior of non-waiting synchron references*: The above examples used the idiom

  `(seq` E `(wait` *sync*`))`

  to evaluate E before the rendezvous of *sync*. A similar effect can be achieved by the simpler idiom:

  `(seq` E *sync*`)`

The reason the latter version works is that *sync* is a non-waiting synchron reference that is not dropped until after E is evaluated ; such a reference serves to block a rendezvous on the synchron until it is dropped.

In fact, the `wait`-less version is often preferable to the `wait` version because it implies fewer constraints. Consider the following related test procedures:

```
(define (test-waitful)
  (let ((a (synchron)))
    (par (par (seq (seq (display 1) (wait a)) (display 2))
              (seq (seq (display 3) (wait a)) (display 4)))
         (seq (wait a) (display 5)))))

(define (test-waitless)
  (let ((a (synchron)))
    (par (par (seq (seq (display 1) a) (display 2))
              (seq (seq (display 3) a) (display 4)))
         (seq (wait a) (display 5)))))
```

`Test-waitful` uses the `wait` idiom for expressing a before constraint, while `test-waitless` uses the `wait`-less idiom. Both procedures specify the constraint that both `1` and `3` are printed before `5`. However, `test-waitful` also constrains `1` to be printed before `4` and `3` to be printed before `2`. In contrast, `test-waitless` does not express these extra constraints.

The blocking behavior non-waiting references has other consequences as well. Reconsider the last version of the `simul!`-constrained examples:

```
(let ((sync-disp1 (make-synchronized-displayer 1 2))
      (sync-disp2 (make-synchronized-displayer 3 4)))
  (seq (simul! (car sync-disp1) (car sync-disp2))
       (par ((cdr sync-disp1)) ((cdr sync-disp2)))))
```

Question: does replacing the `seq` in this example by a `par` change the behavior of the code? It tempting to answer "yes", with the rationale that the unification could possibly happen after all the `display`s if it were allowed to proceed in parallel with them. In fact, the answer is "no". The reason is that the `simul!` expression holds non-waiting pointers to both synchrons in question. The `wait`s peformed by the thunks can't possibly return until the non-waiting pointers are dropped, which is only after the unification has succeeded. There is no danger of a race condition here.

That's the good news. The bad news is that the blocking behavior of non-waiting pointers has an nasty side as well. In particular, it can easily lead to deadlock. Consider the following innocuous-looking expression:

```
;; Deadlocking version
(let ((pair (cons (synchron) 17)))
  (seq (wait (car pair))
       (+ 2 (cdr pair))))
```

This expression is guaranteed to deadlock! The reason is that the pair value of `pair` holds a non-waiting pointer to the synchron and that pointer is accessible until `(cdr pair)` is evaluated. But `(cdr pair)` cannot be evaluated until the `wait` on the synchron returns. This subtle dependency cycle leads to deadlock.

In this case the deadlock can be avoided by forcing the `cdr` to be performed before the `wait`:

```
;; Non-deadlocking version that returns 19
(let ((pair (cons (synchron) 17)))
  (let ((num (cdr pair)))
    (seq (wait (car pair))
         (+ 2 num))))
```

In fact, when dealing with a synchron-bearing data structure, it's always a good idea to unbundle the structure into its components before `wait`ing on any of the synchrons. This *aggressive unbundling* strategy is required of every sliver (see Section 5.4.2) precisely to avoid spurious deadlocks.

- *Is* `precede!` *necessary?*: With the introduction of eagons (see below), `precede!` seems superfluous because it could be defined as:

```
(define (precede! sync1 sync2)
  (eagon (seq (wait sync1) (wait sync2))))
```

This definition uses `eagon` to fork off a process that requires the rendezvous on the first argument to occur before the the rendezvous on the second argument.

Nevertheless, it turns out that a primitive `precede!` form allows certain garbage collection subtleties to be resolved in a pleasing way that is not readily available with the user-defined version (see Section 8.2.2 for details). For this reason, we stick with `precede!` as a primitive.

7.1.4 Excludons

The rendezvous style of synchronization supported by synchrons is helpful for getting parts of a computation to work in lock step. However, synchrons don't help to express *mutual exclusion* between parts of a computation. Constraining OPERA's concurrent argument evaluation strategy to be like Scheme's some-sequential-order strategy is an example of mutual exclusion: the order in which the arguments are evaluated does not matter, but each argument evaluation occupies a single interval of time that cannot intersect with the others.

For expressing mutual exclusion, OPERA supports *excludon* objects. An excludon serves as a key that is needed to gain access to various regions of a computation; because its use is restricted to one region at a time, it guarantees that the regions execute in mutually exclusive time intervals. Excludons are similar to the locking mechanisms supplied in many concurrent systems (e.g. semaphores [Dij68], locks [Bir89b], monitors [Hoa74]).

Each call to the nullary `excludon` constructor creates a unique, first-class excludon object. The form `(exclusive` E_{excl} E_{body}`)` first evaluates E_{excl}, which should be an excludon x, and then evaluates E_{body} while having exclusive hold on x. If x is already held by some other `exclusive`, then evaluation of E_{body} blocks until x is released.

Excludons are used to guarantee that sequences of operations behave as atomic actions. This is particularly important when the operations involve side effects. For example, suppose $\{a, b\}$ represents a time interval in which both a and b are printed (in some order). Then the following procedure guarantees that $\{\texttt{1}, \texttt{2}\}$ must precede $\{\texttt{3}, \texttt{4}\}$ or follow it:

```
(define (test-order_exclusive)
  (let ((e (excludon)))
    (par (exclusive e
           (par (display 1)
                (display 2)))
         (exclusive e
           (par (display 3)
                (display 4))))))
```

Excludons can be used to simulate Scheme's some-sequential-order argument evaluation within OPERA. Consider the `scall` special form, defined by the following desugaring:

(`scall` E_{proc} E_{arg_1} ... E_{arg_n})
  desugars to
(`let` ((I_{excl} `(excludon)`))   ; I_{excl} is a fresh variable
  (`pcall` (`exclusive` I_{excl} E_{proc})
    (`exclusive` I_{excl} E_{arg_1})
    ⋮
    (`exclusive` I_{excl} E_{arg_n})
    ))

The excludon named I_{excl} prohibits interleaved evaluation among the subexpressions of the `pcall`. Sequential versions of `let` and `letrec` (call them `slet` and `sletrec`) can also be defined by restricting the corresponding parallel form with excludons.

7.1.5 Non-strictnesss

In OPERA, evaluation of `pcall` arguments is strict: all arguments must evaluate to a value before the procedure can be applied. This is not always the desired semantics. Sometimes we would like an argument not to be evaluated until it is actually used in the body. That way, if the argument is never used, it will never be evaluated. This strategy is called *laziness.* On the other hand, we sometimes want the body and the arguments evaluated in parallel, regardless of whether body actually uses the arguments. This strategy is called *eagerness.* Both laziness and eagerness are examples of *non-strict* evaluation strategies.

OPERA supports these non-strict evaluation strategies because they are helpful for controlling how computations unfold. There are two types of objects used for this purpose: *lazons* and *eagons.* Lazons support lazy evaluation, while eagons support eager evaluation.

Lazons

A *lazon* is a first-class object that serves as a placeholder for the value of an expression. Lazons are created by the special form (`lazon` E_{body}). The evaluation of E_{body} is delayed until its value is demanded by the computation. A context that requires the value of a lazon's body is said to *touch* the lazon. Examples of touching contexts are argument positions of an arithmetic operator, the test position of an `if`, and the operator position of a `pcall`. All of these contexts need to know specific details about the value of E_{body}. Non-touching contexts, such as the operand positions of a `pcall` and the argument positions of a `cons`, do not probe any details of the lazon's body, and simply pass around the lazon instead. Lazons are similar to the delayed values described in [Hen80] and [ASS85], except that delayed values are touched by application of an explicit `force` procedure. [5] In OPERA, the forcing is done implicitly by a touching context.

Consider the following definitions:

```
(define (display&return x) (begin (display x) x))

(define (test-strict2 a b) (begin (display 3) (* b b)))
```

Then in OPERA

```
(test-strict2 (display&return 1) (display&return 2))
```

prints a `1` and `2` in some order before printing a `3` and returning `4`. Note that `(display&return 1)` is evaluated even though its value is not used by the procedure. In contrast,

```
(test-strict2 (lazon (display&return 1)) (lazon (display&return 2)))
```

prints a `3` followed by a `2` before returning `4`. Because lazons are objects, they satisfy the strictness requirement of the parallel call without their bodies being evaluated. The expression `(display&return 2)` is not evaluated until the value of `b` is demanded by the `*`, which happens only after the `3` has been printed. Even though the value of `b` is demanded twice, a `2` is printed only once because the value of a lazon is *memoized* – the lazon body is evaluated once, and its value is cached for subsequent touches. The `1` is never printed because the formal parameter `a` of `test-strict2` is never referenced in the body.

If there is ever a need to explicitly force a lazon, this can be done with the `touch` procedure, defined as follows:

```
(define (touch x) (if x x x))
```

[5] The Scheme report [CR+91] permits, but does not require, implicit forcing of the promises created by Scheme's `delay`. In contrast, OPERA requires implicit touching of lazons.

The test position of the `if` is a touching context, so it forces the evaluation of the lazon body; that value is returned regardless of whether it is false or not.[6] Memoization guarantees that the evaluation happens only once. `Touch` acts as the identity procedure on non-lazons.

OPERA does not support any predicate for testing whether an object is a lazon. No program can ever distinguish whether or not a purely functional expression has been wrapped in a lazon. If an expression performs side effects, however, then wrapping the expression in a lazon may prevent those side effects from occurring in some contexts.

Lazons can be used to define a lazy argument evaluation strategy. Consider the `lcall` special form, defined by the following desugaring:

```
(lcall E_proc E_arg1 ... E_argn)
  desugars to
(pcall E_proc
       (lazon E_arg1)
             ⋮
       (lazon E_argn)
       )
```

The `lazon`s annotating the arguments prevent them from being evaluated before the procedure is called. The arguments will only be evaluated if they are needed within the body of the called procedure. `Lcall` corresponds to the *call-by-need* parameter passing mechanism common in functional programming languages (e.g. Haskell [HJW+92] and Miranda[Tur85]).

Like the delayed objects of [Hen80] and [ASS85], lazons can be used to create conceptually infinite data structures and finesse certain kinds of circular dependencies. For example, the stream datatype of [ASS85] can be defined in OPERA as follows:

```
;;; Syntactic sugar:
;;; (cons-stream E_head E_tail)
;;;    desugars to
;;; (cons E_head (lazon E_tail))

;;; Procedures
(define head car)
(define tail cdr)
(define the-empty-stream '())
(define empty-stream? null?)
```

[6] In OPERA, as in Scheme, tests of a conditional need not be boolean objects. Any non-false object is treated as true.

Note that OPERA's `tail` operation, unlike Scheme's, does not force the evaluation of the tail of the stream. OPERA relies on touching contexts to do this forcing.

The implicit forcing of lazons supports modularity better than the explicit forcing of delayed objects. We will illustrate this point using a stream integration problem described in [ASS85]. Integration can be defined by the following elegant procedure:

```
(define (integral_original integrand initial-value dt)
  (letrec ((int (cons-stream
                 initial-value
                 (add-streams (scale-stream dt integrand)
                              int))))
    int))
```

`Integral` takes a stream of values, an initial value, and a time step, and returns a stream of the running sum of scaled values (where the sum is initialized to `initial-value`).

Unfortunately, certain feedback problems cannot use this `integral` procedure. For example, consider the following procedure for solving the equation $dy/dt = f(y)$ by integration:

```
(define (solve f y-init dt)
  (letrec ((y (integral dy y-init dt))
           (dy (map-stream f y)))
    y))
```

The `solve` procedure fails because of a circular dependency between `y` and `dy` that is unsolvable (as expressed) in a strict language. The solution suggested in [ASS85] is to delay the first argument to `integral`:

```
(define (solve f y-init dt)
  (letrec ((y (integral (delay dy) y-init dt))
           (dy (map-stream f y)))
    y))
```

But because delayed objects must be explicitly forced, this necessitates a change to `integral`:

```
(define (integral_delayed delayed-integrand initial-value dt)
  (letrec ((int (cons-stream
                 initial-value
                 (add-streams (scale-stream dt (force delayed-integrand)) ; ***
                              int))))
    int))
```

Delaying an argument to a procedure has the undesirable effect of changing its interface. This requires modifying not only the body of the procedure, but all calls to the procedure as well. This is a very non-modular approach to delayed evaluation.

Lazons are much more suitable for such problems. With lazons, the circularity of `solve` can be circumvented as follows:

```
(define (solve f y-init dt)
  (letrec ((y (integral (lazon dy) y-init dt))
           (dy (map f y)))
    y))
```

Because lazons are implicitly touched, there is no need to modify the definition of `integral`$_{original}$, nor any of the existing calls to `integral`. This example illustrates why lazons are superior to delayed objects for preserving modularity.

Eagons

Like lazons, eagons are first-class placeholders for the value of an expression. But whereas a lazon delays the evaluation of its body until it is demanded, an eagon may begin the evaluation of its body as soon as it is created. Eagons are implicitly touched by the same contexts as lazons. Touching an eagon blocks the current computation until the eagon body has been fully evaluated, at which point the computation resumes with this value. Those familiar with *futures* ([Hal85], [Mil87],[For91]) will recognize that "eagon" is just another name for "future".

Eagons are created by the special form `(eagon` E_{body}`)`. Here's a sample use of eagons as arguments to the `test-strict2` procedure from page 316:

```
(test-strict2 (eagon (display&return 1)) (eagon (display&return 2)))
```

The argument expressions immediately evaluate to eagons, allowing the body of `test-strict2` to be evaluated in parallel with the two calls to `display&return`. This means that `1`, `2`, and `3` can be printed in any order. Even though the first argument is never touched, eagons always evaluate their bodies, so the `1` is printed anyway. In fact, the lack of dependences involving `(display&return 1)` means that it is possible for the `1` to be printed *after* `test-strict2` has returned a `4`! On the other hand, data and control dependencies guarantee that both the `2` and `3` are printed before the multiplication in the body of `test-strict2`.

Eagons are useful for controlling fine-grained operational aspects of programs. For example, the `fork2` operator introduced in Section 4.2.5 can be expressed in terms of eagons:

```
(fork2 E1 E2)
  desugars to
(let ((x (eagon E1))
      (y (eagon E2)))
  #t)
```

Eagons are used extensively in the implementation of SYNAPSE to manage the details of demand-driven evaluation.

Eagons can also be used to define an eager argument evaluation strategy. Consider the `ecall` special form, defined by the following desugaring:

```
(ecall Eproc Earg1 ... Eargn)
  desugars to
(pcall (eagon Eproc)
       (eagon Earg1)
              ⋮
       (eagon Eargn)
       )
```

The `pcall` and `eagon`s allow all subexpressions of the `ecall` to be evaluated in parallel. But the `eagon`s on the arguments further allow the arguments to be evaluate concurrently with the body of the procedure. `Ecall` corresponds to the default procedure application strategy for the Id programming language [AN89, Tra88].

7.1.6 Graphical Bindings

OPERA's `nex` and `nexrec` constructs are "graphical" versions of `let` and `letrec`. Whereas `let` and `letrec` associate names with first-class values, `nex` and `nexrec` associate names with syntactic entities so that the general graph-structured syntactic dependencies can be expressed within the tree-structured confines of s-expressions. `Nex` and `nexrec` are essentially lazy forms of `let` and `letrec` embedded within a strict language.

We motivate these constructs and explain their semantics via a simple example. Consider the following OPERA procedure:

```
(lambda (a b x y)
  (cons (if (test a)
            (compute x)
            (compute y))
        (if (test b)
            (compute x)
            (compute y))))
```

Suppose that `test` is a unary predicate and `compute` is an expensive unary procedure with no side effects. In some cases, the procedure performs more work than necessary: `(compute x)` is evaluated twice if both tests are true, and `(compute y)` is evaluated twice if both tests are false.

One way to avoid the extra computation is to employ a finer-grained case analysis:

```
(lambda (a b x y)
  (let ((ta (test a))
        (tb (test b)))
    (if ta
        (if tb
            (let ((cx (compute x))) (cons cx cx))
            (cons (compute x) (compute y)))
        (if tb
            (cons (compute y) (compute x)))
            (let ((cy (compute y))) (cons cy cy))))))
```

Unfortunately, this approach significantly impairs readability.

Another approach is to use the laziness and memoization provided by lazons to achieve the appropriate effect:

```
(lambda (a b x y)
  (let ((cx (lazon (compute x)))
        (cy (lazon (compute y))))
    (cons (if (test a) (touch cx) (touch cy))
          (if (test a) (touch cx) (touch cy)))))
```

Explicit touches are required in this case because the operands of `cons` constructor are not implicitly touched. This approach leads to a more readable program than the finer-grained case analysis, but the extra `lazon`s and `touch`es are distracting management details.

Figure 7.3 uses the graphical notation introduced earlier (and formalized in Chapter 8) to depict the desired structure for the body of the sample procedure. The results of both `compute` applications are shared by both `if`s. According to the demand-driven model detailed in Chapter 8), each call to `compute` is evaluated only once if the result is required, and is not evaluated at all if the result is not required.

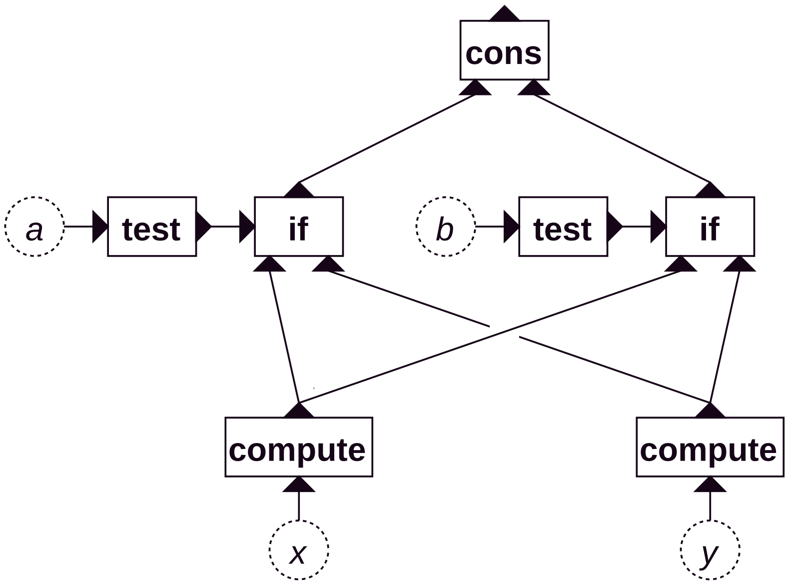

Figure 7.3: Graphical depiction of a program whose sharing properties are difficult to express in textual syntax.

The `nex` binding construct expresses the sharing exhibited by the graphical notation within textual OPERA code. Here is a version of the sample procedure that uses `nex`:

```
(lambda (a b x y)
  (nex ((cx (compute x))
        (cy (compute y)))
    (cons (if (test a) cx cy)
          (if (test a) cx cy))))
```

Intuitively, `nex` associates the names `cx` and `cy` with the outputs of the graph structure corresponding to the `compute` applications, and any references to these names in the body of the `nex` "wire" the appropriate output into the graph structure corresponding to the body. For this reason, we refer to `nex` as a graphical binding. The meaning of the `nex` version is actually defined in terms of the graph model, but it can also be viewed as an more convenient syntax for the explicit `lazon`/`touch` approach (i.e., we could desugar any `nex` expression into an expression using explict `lazon`s and `touch`es).

In general, `nex` behaves differently from `let`. If `nex` were replaced by `let` in the above example, the strictness of OPERA would dictate that both `(compute x)` and `(compute y)` would be evaluated before the body, regardless of whether both their values were used.

`Nexrec` is similar to `nex`, but has the recursive scoping of a `letrec`. For example, the OPERA expression

```
(nexrec ((a (cons 1 b))
         (b (cons 2 a)))
  (cons a b))
```

corresponds to the graphical syntax depicted in Figure 7.4.[7]

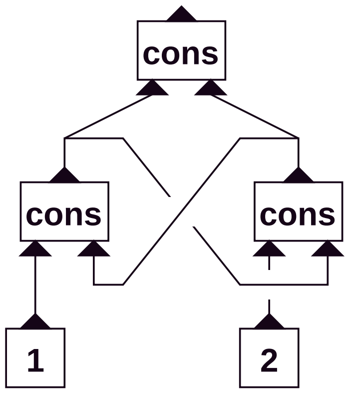

Figure 7.4: Graphical syntax exhibiting mutual dependencies that can be expressed with `nexrec`.

7.2 Implementing SYNAPSE

The OPERA language is equipped with the concurrency, synchronization, and non-strictness machinery necessary to implement slivers and slags. For the sake of concreteness, we will illustrate OPERA implementations of some the SYNAPSE slivers introduced in Chapter 6.

Implementing the elements of the lock step processing model in OPERA is a challenging activity. The features provided by OPERA for controlling the operational behavior of programs are very powerful but also rather low-level. The dizzying number of ways in which these features can be combined to yield slightly different computations can easily lead to mental overload. Yet, in order to achieve the goal of operational faithfulness, an imple-

[7]This example is somewhat contrived. According to the rules of the EDGAR model introduced in Chapter 8, evaluating the sample `nexrec` expression leads to deadlock. It is necessary to introduce some non-strictness to avoid deadlock.

menter of slivers must meticulously consider the consequences of every design choice. Here are some sample questions that continually confront the sliver implementer:

- How do I propagate demand to this subexpression without demanding these other subexpressions?
- Does this sliver release its hold on storage resources as soon as it can?
- Should I express this timing constraint with sequencing, an explicit `wait`, or a blocking reference to a synchron?
- How can I ensure that this timing constraint doesn't introduce a spurious deadlock?
- Should this expression be left as is, or does it need to be wrapped in a `lazon` or `eagon`?
- Do I need an explicit `touch` here or not?

Because the complexities of sliver implementation are so overwhelming, it is unreasonable to expect that average programmers should ever have to deal with them. Instead, programmers should be provided with easy-to-use abstractions that hide all the messy details. The SYNAPSE procedures of Chapter 6 are in this spirit. The kinds of questions listed above need only be considered by the expert programmers implementing the abstractions.

In order to simplify the presentation, I will discuss the details of sliver implementation in stages. First, I introduce some conventions and abstractions for manipulating slags that simplify the subsequent discussion. Then I describe how to implement a few linear slivers ignoring the issues of filtering. Next I show how to properly handle filtering for the linear slivers. Finally, I conclude with some notes on the implementation of tree-shaped slivers.

7.2.1 Slag Conventions

Representing and manipulating slags is a tricky business for several reasons:

- Slags contain references to synchrons. If these references aren't handled carefully, spurious deadlocks will result.
- The laziness aspect of slags means that components of a slag should not be computed until they are required. Including just the right amount of laziness can be challenging.

- The aggregate nature of slags means that they can potentially consume large amounts of storage. When manipulating slags, it is necessary to exercise great care to ensure that there is no unnecessary consumption of storage resources.

Here we present a set of conventions and some abstractions that simplify the manipulation of slags.

Recall that a slag is an aggregate whose skeletal nodes are wired together with synters (synchronized pointers). We will represent a synter as an entity consisting of two synchrons (one down, one up) and a lazon that computes the skeletal node pointed at by the synter. Figure 7.5 presents a stylized schematic of a synter. The nodes labelled **S** are synchrons and the node labelled **L** is a lazon.

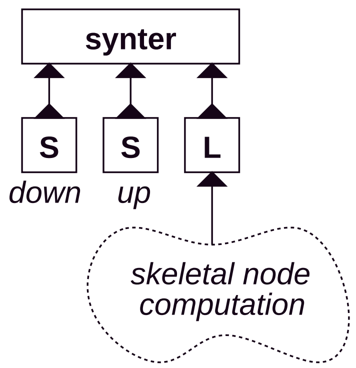

Figure 7.5: Graphical depiction of the structure of a synter.

A synquence (synchronized lazy list) is a slag whose skeletal nodes are either a distinguished end-of-list marker or a pair of a lazily computed element and child synquence (see Figure 7.6). A syndrite (synchronized lazy tree) is a slag whose skeletal nodes are pairs of a lazily computed element and a list of children syndrites (see Figure 7.7).

The basic structure illustrated in the figures is augmented with timing constraints that define when the computations suspended by the lazons actually take place. For a synter, the computation of the skeletal node takes place after the rendezvous at the down synchron. Attempting to manipulate the skeletal node before this point in time will result in deadlock.

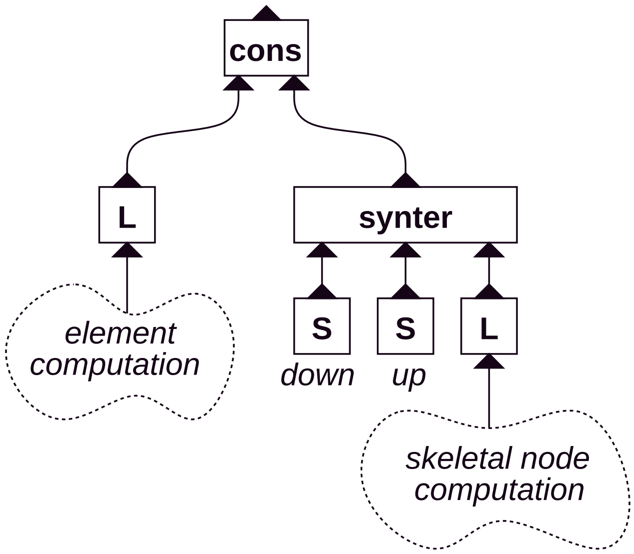

Figure 7.6: Structure of the skeletal node of a synquence.

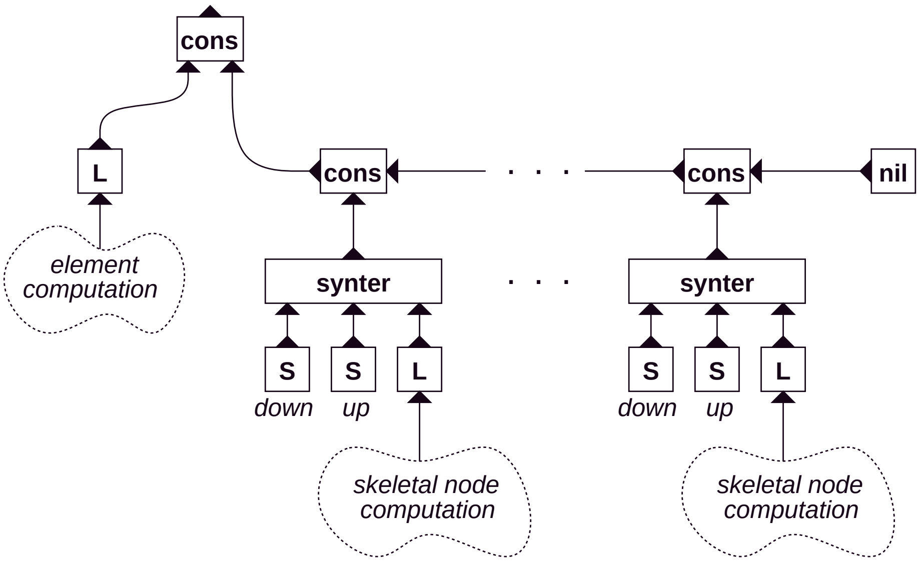

Figure 7.7: Structure of the skeletal node of a syndrite.

The timing of the element computation of a synquence depends on the shapes of the slivers producing and consuming it. If the producing sliver has *down* shape, then the element must be computed before the time of the down synchron of the child synquence. If the producing sliver has *up* shape, then the element must be computed after the rendezvous of the up synchron of the child synquence but before the rendezvous of the up synchron of the synquence itself. If the producing synquence has *across* shape, then the timing constraints are determined by context (see Section 5.2.3).

The timing constraints on the computation of a syndrite element are even more varied. They express when the element is computed relative to the down and up synchrons of the given syndrite and its children syndrites. It is a dynamic error for a program to require the value of a synquence or syndrite element at a time that precedes the time at which it is actually computed.

Taken together with the sliver requirements of Section 5.4.2, the structure depicted in Figures 7.5 – 7.7 and the timing constraints sketched above comprise a set of conventions for implementing slivers. On the one hand, the conventions determine the properties of an input slag that a sliver implementer can depend on. For example, an implementer knows that that all the synters for the children of a syndrite will be available before a rendezvous on any child's down synchron. On the other hand, the conventions limit the ways in which the implementer may produce output slags. As the flip side of the previous syndrite example, an implementer must guarantee that all the synters for the children of a produced syndrite must be available before the computation of any of the skeletal nodes of those children.

7.2.2 Slag Abstractions

Carefully following all the conventions outlined above can be difficult because there are so many details to monitor. Here we present a few OPERA procedures that facilitate slag manipulation by packaging up some of the conventions into handy abstractions.

Synters will be represented as three-element lists assembled via `synter` and disassembled via `unsynter`:

```
(define (synter down up $snode)
  (list down up $snode))

(define (unsynter slag receiver)
  (receiver (car slag) (cadr slag) (caddr slag)))
```

The `down` and `up` arguments of `synter` should be synchrons, and the `$snode` argument should be a lazon that computes the skeletal node. The `$` in `$snode` is a naming convention indicating that the variable names a lazon.

The following abstractions capture the notion of a rendezvous occuring at a procedure call boundary:[8]

```
(define (call/down down proc . args)
  (seqn (wait down)
        (apply proc args)))

(define (call/down-up down up proc . args)
  (seqn (wait down)
        (seq1 (apply proc args)
              (wait up))))
```

`Call/down` uses the synchron `wait` procedure to rendezvous with other slivers at a `down` synchron before a tail call of `proc` applied to `args`. `Call/down-up` is similar, but also engages in a rendezvous at the `up` synchron upon return from `proc` to model a non-tail call.

Figure 7.8 presents some abstractions that are helpful for satisfying the rendezvous requirements for procedure calls that manipulate slags. `Unslag/down` and `unslag/down-up` are variants of `call/down` and `call/down-up` that specially handle the case where the first argument is a slag. To allow demand to propagate through a chain of nested sliver applications, it is necessary in this case to eagerly evaluate the (otherwise delayed) computation that produces the skeletal node of the slag. Every skeletal node computation must perform a `wait` on the down synchron before doing any real work. If this computation were not eagerly evaluated, then the `wait` performed by `call/down` or `call/down-up` would certainly deadlock. The `wait`s within the skeletal node computations guarantee that the eager evaluation of one skeletal node can't race too far ahead of another.

`Reslag/down`, `reslag2/down` and `reslag/down-up` are useful for implementing slivers that map one or more input slags to an output slag. `Reslag/down` maps an input slag to

[8] Since the current implementation of OPERA does not support rest arguments, these procedures and other procedures using rest arguments are actually implemented as macros.

```
(define (unslag/down proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down down
        proc (eagon (touch $snode))
             args))))

(define (unslag/down-up proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down-up down up
        proc (eagon (touch $snode))
             args))))

(define (reslag/down proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (synter down up
        (lazon (apply call/down down
                 proc (eagon (touch $snode))
                      args))))))

(define (reslag2/down proc slag1 slag2 . args)
  (unsynter slag1
    (lambda (down1 up1 $snode1)
      (unsynter slag2
        (lambda (down2 up2 $snode2)
          (let ((down (simul! down1 down2))
                (up (simul! up1 up2)))
            (synter down up
              (lazon (apply call/down down
                       proc (eagon (touch $snode1))
                            (eagon (touch $snode2))
                            args)))))))))

(define (reslag/down-up proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                               proc (eagon (touch $snode))
                                    args)))
          (nex ((new-snode (car new-snode&other))
                (other (eagon (seq1 (touch (cdr new-snode&other))
                                    (wait up)))))
            (let (($other (lazon other)))
              (receiver (synter down up (lazon (seq1 new-snode other)))
                        $other)))))))
```

Figure 7.8: Abstractions for procedure application that facilitate the implementation of rendezvous requirements.

an output slag in which every skeletal node of the output is determined by applying `proc` to the corresponding skeletal node of the input. It is essentially a slag-returning version of `unslag/down`. `Reslag2/down` is a similar procedure that maps two input slags to an output slag. In order to handle fan-in, `reslag2/down` uses `simul!` to unify the corresponding synchrons of its two inputs.

`Reslag/down-up` is a slag-returning version of `unslag/down-up`. Its interface is complicated by the fact that it returns not only a new slag, but also an accumulator (`other`) that is computed in the *up* phase. The `receiver` argument to `reslag/down-up` is a procedure that accepts these two results. The `proc` argument, which is applied to the input slag and the other arguments, is expected to return a pair of the new slag and up accumulator. The intricacies in the definition of `reslag/down-up` are due the complex collection of timing constraints that must be satisfied. In particular, the new slag must be available in the *down* phase of the recursion, but the up accumulator will not be computed until the *up* phase of the recursion. To get this effect, the up accumulators returned by `proc` and returned to `receiver` must be appropriately delayed. The definition uses `nex`, `eagon`, and `seq1` to implement intricate mechanisms for ensuring that operations occur in an order that avoids spurious deadlocks and space build-up.

While the procedures presented in this section are helpful in many cases, they aren't appropriate for all situations. Implementing slivers sometimes requires manipulating low-level details in a way that is incompatible with the abstractions developed here.

7.2.3 Unfiltered Synquences

As an introduction to sliver implementation, we first consider *unfiltered synquences* — i.e., synquences whose elements are all present. This permits us to study the essence of the synchronization issues without the considerable complications associated with filtering. We will describe the implementations of a few representative slivers: the `genQ` generator, the `map2Q` mapper, the `downQ` and `upQ` accumulators, and the `down-scanQ` and `up-scanQ` scanners.

GenQ

`GenQ` creates a synquence from scratch:

```
(define (genQ init next done?)
  (define (gen-synq $arg up-parent)
    (let ((down (synchron))
          (up (synchron)))
      (precede! up up-parent)
      (synter down up
        (lazon (call/down down gen-list (touch $arg) up)))))
  (define (gen-list arg up)
    (if (done? arg)
        '()
        (cons (lazon arg) (gen-synq (lazon (next arg)) up))))
  (gen-synq (lazon init) (synchron)))
```

The internal `gen-synq` procedure creates two fresh synchrons and packages them up into a synter, while the `gen-list` procedure returns a skeletal pair node for the new synquence. The `up-parent` argument to `gen-synq` is the up synchron of previous call; it is passed in to permit the declaration that a rendezvous at the current up synchron must precede a rendezvous at its parent. The `(synchron)` in the initial call to `gen-synq` is an arbitrary synchron that serves as the parent to the first up synchron in the resuling synquence. The `$arg` argument to `gen-synq` is delayed so that it will only be computed if `done?` tests false.

In the above code, the `lazon`s in the `(lazon arg)` and `(lazon init)` expressions are actually superfluous. Since `arg` and `init` are already guaranteed to be values, there is no need to delay their computation. Henceforth, we will eliminate superfluous lazons without comment.

Map2Q

`Map2Q` uses `reslag2/down` to map two input synquences into one output synquence:

```
(define (map2Q fun synq1 synq2)
  (reslag2/down synq1 synq2
    (lambda (lst1 lst2)
      (if (or (null? lst1) (null? lst2))
          '()
          (let (($elt1 (car lst1))  ; Aggressive unbundling
                ($elt2 (car lst2))) ;
            (cons (lazon (fun $elt1 $elt2))
                  (map2Q fun (cdr lst1) (cdr lst2))))))))
```

The handling of the `null?` tests guarantees that the length of the output synquence is the shorter of the two input synquences. Truncation slivers can be handled in a similar fashion.

Using `let` to name the `car`s of `lst1` and `lst2` is a crucial application of the aggressive unbundling principle set forth in Section 5.4.2. If the expression

```
(let (($elt1 (car lst1))
      ($elt2 (car lst2)))
  (cons (lazon (fun $elt1 $elt2))
        (map2Q fun (cdr lst1) (cdr lst2))))
```

were instead replaced by

```
(cons (lazon (fun (car lst1) (car lst2)))
      (map2Q fun (cdr lst1) (cdr lst2)))
```

then the `map2Q` sliver could lead to deadlock in certain contexts. The reason is that `lst1` and `lst2` refer to synters in their `cdr` slots, and those synters contain references to down synchrons. Even though the element computation would not require these synters, the references to them would not be dropped until the element computation was demanded. If the element computation were not demanded until the *up* phase, then the rendezvous on the down synchrons would be blocked and deadlock would ensue. This example is typical of the kind of careful reasoning that must be performed in the presence of synchrons to avoid deadlock.

DownQ and UpQ

Except for the use of the synchronized calling abstractions, the down and up accumulators look like classical list accumulators:

```
(define (downQ init op synq)
  (define (accum lst acc)
    (if (null? lst)
        acc
        (unslag/down accum (cdr lst) (op (car lst) acc))))
  (unslag/down accum synq init))

(define (upQ init op synq)
  ;; OP's first arg is lazy and second arg is eager
  (define (accum lst)
    (if (null? lst)
        init
        (op (car lst)
            (eagon (unslag/down-up accum (cdr lst))))))
  (unslag/down-up accum synq))
```

`DownQ` uses `unslag/down` to preserve tail recursion, while `upQ` uses `unslag/down-up` to return from every call.[9]

The careful reader may be curious about the presence of an `eagon` within `upQ`. The justification for this `eagon` is rather subtle. We shall motivate its purpose in depth as an example of the kinds of details and design choices that makes sliver implementation so challenging.

The `eagon` within the definition `upQ` is intended to enhance the operational faithfulness of that sliver. Without the `eagon`, `upQ` can unduly restrict the timing of of element computations. Consider the SYNAPSE expression:

```
(upQ (lambda ($elt acc) (+ $elt acc))
     0
     (mapQ square (to-1 3)))
```

Since the square mapper is arranged between a down and an up sliver, we expect that the relative order of `square`s should be unconstrained (see Section 5.2.3). But an `eagon`-less implementation of `upQ` will force all the `squares` to happen in the *up* phase of the computation! Why? Since synquence elements are lazy, they are not computed until their value is required. By inspecting the body of the `lambda` expression, *we* can tell that all of the results of the `square` applications *will* be needed — as arguments to `+`. But the OPERA interpreter doesn't look inside the body of the `lambda` until it applies the procedure, and it only does that when both arguments are available. Since the second argument does not become available until the *up* phase, none of the `square` applications will be demanded until the *up* phase.

Inserting an `eagon` around the second argument allows (but does not require) the application of `op` to occur in the *down* phase. If the body of `op` requires the element denoted by its first argument (as it does in the above example), the element computation may be performed in the *down* phase. If the body of `op` does not require the value of this element,

[9]There really should be two versions of `upQ`: one in which the initial call is made via `unslag/down`, and another in which it is made via `unslag/down-up`. The former would be used in situations where `upQ` appears in a tail call position, and the latter would be used for calls in non-tail positions. Similarly, there should be two versions of `downQ`. We have simplified matters here by considering only the most common case. The need for different versions underscores the fact that the slivers are explicitly *simulating* tail calls. It would be preferable to have a system that automatically inserted an up rendevous only when it was necessary. However, preliminary attempts at such a system have been unsuccessful.

it will not be computed at all. This behavior better matches the expected behavior of `upQ`.

Unfortunately, the fix changes the interface to `op` in a subtle way. The fact that the second argument to `op` is now an eagon can lead to unexpected behavior for an `op` that does not require the value of its second argument. Consider the following SYNAPSE expression:

```
(upQ (lambda ($elt acc) (display $elt))
     'ignore
     (to-1 3))
```

Naively, we might expect this expression to print out numbers during the *up* phase in the order `1`, `2`, `3`. And this *would* be the behavior in the `eagon`-less version of `upQ`. But in the `eagon` version of `upQ`, nothing prevents the numbers to print out during the *down* phase in the order `3`, `2`, `1`. In such cases, getting the desired behavior may require extending `op` with an explicit control dependency:

```
(upQ (lambda ($elt acc) (seqn (touch acc) (display $elt))
     'ignore
     (to-1 3))
```

In this case, requiring the `touch` of `acc` to occur before the `display` expresses the constraint that all the `display`s should occur in reverse order during the *up* phase.

Down-scanQ and Up-scanQ

`Down-scanQ` and `up-scanQ` are scanning transducers patterned after the `downQ` and `upQ` accumulators. The definition of `down-scanQ` is essentially a version of `downQ` that uses `reslag/down` in place of `unslag/down`:

```
(define (down-scanQ init op synq)
  (define (scan lst acc)
    (if (null? lst)
        '()
        (let ((new-acc (op (car lst) acc)))
          (cons new-acc
                (reslag/down scan (cdr lst) new-acc)))))
  (reslag/down scan synq init))
```

The internal `scan` procedure maps a given skeletal node (`lst`) into one that holds the new value of a down accumulator.

Although the definition of `up-scanQ` is also closely related to that of `upQ`, the resemblance is clouded by the more complex interface of `reslag/down-up`:

```
(define (up-scanQ init op synq)
  ;; OP's first arg is lazy and second arg is eager
  (define (scan lst)
    (if (null? lst)
        (cons '() init)
        (let (($elt (car lst)))
          (reslag/down-up scan (cdr lst)
            (lambda (new-list $sub-acc)
              (let (($up-acc (lazon (op $elt $sub-acc))))
                (cons (cons $up-acc new-list)
                      $up-acc)))))))
  (reslag/down-up scan synq (lambda (final-synq $final-acc) final-synq)))
```

In this case, `scan` maps a given skeletal node (`lst`) into a pair of (1) a new skeletal node and (2) a lazon of the calculation of the up accumulator. The `lazon` is necessary because the pair must be returned in the *down* phase, long before the application of `op` will take place in the *up* phase. Removing the `lazon` would lead to a deadlock because then the pair could not returned until the *up* phase even though its `car` would be required by the *down* phase.

The delayed up accumulator, `$up-acc`, is used both by the new skeletal node and the `cdr` of the returned pair. Since the accumulator is already present in the returned skeletal node, why is it also necessary to return it as a separate element of the pair? The answer is that it helps to make the base case work smoothly. Since the empty list does not contain the initial accumulator `init`, an alternate mechanism must be found for associating `init` with the empty list. Returning the accumulator in the second slot of the pair avoids having to test `new-list` with `null?` upon every return. If `up-scanQ` were modified so that it did not take an initial accumulator value but instead started accumulation with the last element of the input, this workaround would be unnecessary.

7.2.4 Filtered Synquences

As described in Section 5.5.3, it is tricky to design reusable slivers that deal with filtering in a reasonable way. Here, we present an approach to synquence filtering that addressed the issues raised in that section. Fortunately, we will be able to use many of the same synchronization abstractions that proved handy for the unfiltered case. Indeed, by hiding unnecessary detail, those abstractions help to highlight what is different about the filtered case.

New Abstractions

We begin by defining some new abstractions designed specifically for filtering. In the unfiltered case, the element held by a skeletal node is represented as a lazy computation. In the filtered case, each element is annotated with two flags:

1. The *down-element?* flag indicates whether the associated element is computed in the down or up phase.

2. The *down-gap?* flag indicates whether the element slot holds a gap in the *down* phase of a computation. If *down-gap?* is true, the associated element must be a gap. If *down-gap?* is false, then the associated element might be a non-gap value determined in the *down* phase, or it might be any value (including a gap) determined in the *up* phase.

All four combinations of flags are possible. The most "interesting" combination is when the *down-element?* is false but *down-gap?* is true. This means that a slot produced by a *up* sliver is known to contain a gap during the *down* phase even though the ungapped slots produced by the sliver are all computed in the *up* phase. This explicit shape information makes general filtering mechanisms possible by informing slivers when it's safe to test a slot for a gap.

An element and its two flags will be bundled together into a structure called a *filton*. These structures are built by `filton` and taken apart by `unfilton`:

```
(define (filton down-elt? down-gap? $elt)
  (list down-elt? down-gap? $elt))

(define (unfilton fil receiver)
  (receiver (car fil) (cadr fil) (caddr fil)))
```

It is also helpful to have functions that map between filtons. Given a filton, `refilton` returns a new filton with the same flags in which a given function is lazily applied to the original element:

```
(define (refilton fil f)
  (unfilton fil
    (lambda (down-elt? down-gap? $elt)
      (filton down-elt? down-gap?
        (lazon (let ((e (eagon (touch $elt))))
                 (if (gap? e)
                     #g       ; The gap literal
                     (f e))))))))
```

We assume that `f` never returns a gap as a result, because that might violate the validity of the `down-gap?` flag. `Refilton2` is similar, but takes two inputs:

```
(define (refilton2 fil1 fil2 f)
  (unfilton fil1
    (lambda (down-elt1? down-gap1? $elt1)
      (unfilton fil2
        (lambda (down-elt2? down-gap2? $elt2)
          (filton (and down-elt1? down-elt2?)
                  (or down-gap1? down-gap2?)
                  (lazon (let ((e1 (eagon (touch $elt1)))
                               (e2 (eagon (touch $elt2))))
                           (if (or (gap? e1) (gap? e2))
                               #g   ; The gap literal
                               (f e1 e2))))))))))
```

The resulting filton holds a gap if either input holds a gap. The *down-gap?* flag of the result is true if it is true of either input, but the resulting element is produced in the down phase only if both inputs are produced in the *down* phase.

The key constraint to be obeyed in filton manipulation is that the two flags of a filton must be accessible in the *down* phase of a computation. The element itself may be computed at any time, but its enclosing filton must contain a lazon of this computation in the *down* phase.

Implementing Filtered Slivers

We study filtering by examining some sliver implementations. First we consider the `filterQ` sliver, and then we reconsider the same six slivers studied for the unfiltered case.

`FilterQ` is the only primitive sliver capable of introducing gaps where they did not exist before. An implementation of the synquence filtering sliver, `filterQ`, is shown in Figure 7.9. `FilterQ` is a special kind of mapping sliver that maps an element either to itself or to a gap. As in most slivers on filtered synquences, the action taken by `filterQ` depends on the values of the filton flags. If the element is known to be a gap in the *down* phase the filton is passed untouched. Otherwise the action depends on when the element will be available. If the element is available in the *down* phase (i.e., *down-element?* is true), then it can be immediately tested with `pred`. But if the element won't be available until later (i.e., *down-element?* is false), then a `gap?` test and `pred` test must be delayed until the *up* phase. In all cases, the sliver must produce a valid filton object in the *down* phase so that other slivers can examine it during the *down* phase.

```
(define (filterQ pred synq)
  (define (filter-list lst)
    (if (null? lst)
        '()
        (let ((fil (car lst)))
          (cons (unfilton fil
                  (lambda (down-elt? down-gap? $elt)
                    (if down-gap?
                        fil
                        (if down-elt?
                            (if (pred (touch $elt))
                                fil
                                (filton #t #t #g))
                            (filton #f #f
                              (lazon (let ((elt (touch $elt)))
                                       (if (gap? elt)
                                           #g
                                           (if (pred elt) elt #g)))))))))
                (reslag/down filter-list (cdr lst))))))
  (reslag/down filter-list synq))
```

Figure 7.9: An implementation of the `filterQ` sliver for filtering synquences.

Filtered versions of `genQ` and `map2Q` appear in Figure 7.10. These are minor tweaks to the unfiltered versions; only the lines marked with `;***` have been changed. Every filton created by `genQ` contains ungapped elements produced in *down* phase (i.e., *down-element?* is true, *down-gap?* is false). `Map2Q` calls `refilton2` to do the actual mapping work.

Filtered versions of `downQ` and `down-scanQ` (Figure 7.11) differ from their unfiltered counterparts by treating gaps specially. Each avoids performing an accumulation step in the case where *down-gap?* is true.

Extending the unfiltered versions of `UpQ` and `up-scanQ` to filtered versions requires more substantial changes (Figure 7.12). When *down-gap?* is true, `upQ` can effectively make a

```
(define (genQ init next done?)
  (define (gen-synq $arg prev-up)
    (let ((down (synchron))
          (up (synchron)))
      (precede! up prev-up)
      (synter down up
        (lazon (call/down down gen-list (touch $arg) up)))))
  (define (gen-list arg up)
    (if (done? arg)
        '()
        (cons (filton #t #f arg)                  ; ***
              (gen-synq (lazon (next arg)) up))))
  (gen-synq init (synchron)))


(define (map2Q fun synq1 synq2)
  (define (map2-list lst1 lst2)
    (if (or (null? lst1) (null? lst2))
        '()
        (let ((fil1 (car lst1))
              (fil2 (car lst2)))
          (cons (refilton2 fil1 fil2 fun)         ; ***
                (reslag2/down map2-list (cdr lst1) (cdr lst2))))))
  (reslag2/down map2-list synq1 synq2))
```

Figure 7.10: Filtered implementations of `genQ` and `map2Q`.

```
(define (downQ init op synq)
  (define (accum lst acc)
    (if (null? lst)
        acc
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-elt?
                (if down-gap?
                    (unslag/down accum (cdr lst) acc)
                    (unslag/down accum (cdr lst) (op $elt acc)))
                (error "DOWNQ: Shape error"))))))
  (unslag/down accum synq init))

(define (down-scanQ init op synq)
  (define (down-list lst acc)
    (if (null? lst)
        '()
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-elt?
                (if down-gap?
                    (cons (filton #t #f acc)
                          (reslag/down down-list (cdr lst) acc))
                    (let ((new-acc (op $elt acc)))
                      (cons (filton #t #f new-acc)
                            (reslag/down down-list (cdr lst) new-acc))))
                (error "DOWN-SCANQ: Shape error"))))))
  (reslag/down down-list synq init))
```

Figure 7.11: Filtered implementations of `downQ` and `down-scanQ`.

tail call; but if *down-gap?* is false `upQ` must defer an explicit `gap?` test until the *up* phase. `Up-scanQ` is similar, but is complicated by the need to create a valid filton in the *down* phase even though all the applications of `op` will occur in the *up* phase.

Expressing timing constraints for *up* slivers proves to be quite tricky in the filtered case. In `upQ` and `up-scanQ`, the goal is to require each computation of `op` to occur between a consecutive pair of up synchrons. But the slag manipulation abstractions developed above only rendezvous on the up synchron corresponding to the input of `op`; nothing explicitly forces `op` to return before the up synchron corresponding to its output. This does not cause a problem in the unfiltered case, because every position of the recursion is occupied with an instance of `op`. But in the filtered case, some positions are not occupied, and the output of an `op` may not be properly constrained.

To fix this problem, it is necessary to extend the slag abstractions to pass down an up synchron from above. Figure 7.13 shows new versions of these abstractions in which each `proc` takes an additional argument, called a *returner*. A returner is a unary procedure that maintains a non-waiting pointer to an up synchron. The procedure acts as the identity, but calling the procedure also releases the synchron pointer, which in turn may enable a rendezvous at the synchron. Both `upQ` and `up-scanQ` use the modified slag abstractions to obtain access to a returner (bound to the variable `return`). The return is wrapped around non-trivial up computation to guarantee that it is bounded from above as well as from below.

7.2.5 Syndrites

The implementation of syndrites follows the same ideas used in the implementation of synquences. In fact, the same slag abstractions that aid the synquence implementations are useful for many of the syndrite implementations. However, the fact that a syndrite skeletal node generally has multiple children can introduce many headaches.

To give the flavor of syndrite slivers, I will briefly describe the implementations of three simple unfiltered examples. Figure 7.14 shows the implementations of `mapD`, `preD`, and `down-scanD`.

The `mapD` syndrite mapper closely resembles synquence mappers. The main difference

```
(define (upQ init op synq)
  (define (accum lst return)
    (if (null? lst)
        (return init)
        (unfilton (car lst)
          (lambda (down-elt? down-gap? $elt)
            (if down-gap?
                (unslag/down-return accum (cdr lst))
                (let ((sub-acc (eagon (unslag/down-up-return accum (cdr lst)))))
                  (return
                    (if (gap? $elt)
                        (touch sub-acc)
                        (op $elt sub-acc))))))))))
  (unslag/down-up-return accum synq))

(define (up-scanQ init op synq)
  (define (up-list lst return)
    ;; Returns a pair of (1) A new list (2) A (lazy) new accumulator
    (if (null? lst)
        (cons '() (lazon (return init)))
        (let ((fil (car lst))
              (subsynq (cdr lst)))
          (unfilton (car lst)
            (lambda (down-elt? down-gap? $elt)
              (if down-gap?
                  ;; Return a non-gap in either case.
                  (reslag/down-return-receiver up-list subsynq
                    (lambda (new-list $sub-acc)
                      (cons (cons (filton #f #f $sub-acc)
                                  new-list)
                            $sub-acc)))
                  (reslag/down-up-return up-list subsynq
                    (lambda (new-list $sub-acc)
                      ;; This is tricky.  In all cases, MUST return a pair of
                      ;; a synchron and a lazy accumulator immediately.
                      (nex ((up-acc (if (gap? $elt) $sub-acc (op $elt $sub-acc))))
                        (cons (cons (filton #f #f (lazon up-acc))
                                    new-list)
                              (lazon (return up-acc)))))))))))))
  (reslag/down-up-return up-list synq (lambda (final-synq $final-acc) final-synq)))
```

Figure 7.12: Filtered implementations of `upQ` and `up-scanQ`.

```
(define (unslag/down-return proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down down
        proc (eagon (touch $snode))
                  (lambda (x) (seq1 x up)) ; Returner
                  args))))

(define (unslag/down-up-return proc slag . args)
  (unsynter slag
    (lambda (down up $snode)
      (apply call/down-up down up
             proc (eagon (touch $snode))
                  (lambda (x) (seq1 x up)) ; Returner
                  args))))

(define (reslag/down-return-receiver proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                                 proc (eagon (touch $snode))
                                      (lambda (x) (seq1 x up)) ; Returner
                                      args)))
        (nex ((new-snode (car new-snode&other))
              (other (eagon (touch (cdr new-snode&other)))))
          (let (($other (lazon other)))
            (receiver (synter down up (lazon (seq1 new-snode other)))
                      $other)))))))

(define (reslag/down-up-return proc slag receiver . args)
  (unsynter slag
    (lambda (down up $snode)
      (nex ((new-snode&other (apply call/down down
                                 proc (eagon (touch $snode))
                                      (lambda (x) (seq1 x up)) ; Returner
                                      args)))
        (nex ((new-snode (car new-snode&other))
              (other (eagon (seq1 (touch (cdr new-snode&other))
                                  (wait up)))))
          (let (($other (lazon other)))
            (receiver (synter down up (lazon (seq1 new-snode other)))
                      $other)))))))
```

Figure 7.13: Modified slag abstractions used by upQ and up-scanQ.

```
(define (mapD f synd)
  (define (map-tree tr)
    (let (($elt (elt tr)))
      (tree (lazon (f $elt))
            (map (lambda (kid) (mapD f kid))
                 (kids tr)))))
  (reslag/down map-tree synd))

(define (down-scanD init op synd)
  (define (accum tr acc)
    (let (($elt (elt tr))
          (subdrites (kids tr)))
      (let ((new-acc (op $elt acc)))
        (tree new-acc
              (map (lambda (synd) (reslag/down accum synd new-acc))
                   subdrites)))))
  (reslag/down accum synd init))

(define (preD dir init op synd)
  (define (walk-tree tr return acc)
    (let (($elt (elt tr))
          (ks (kids tr)))
      (let ((new-acc (op $elt acc)))
        (if (null? ks)
            (return new-acc)
            (walk-trees ((permuter dir) ks) new-acc)))))
  (define (walk-trees synds acc)
    (let ((fst (car synds))
          (rest (cdr synds)))
      (if (null? rest)
          (unslag/down-return walk-tree fst acc)
          (walk-trees rest (unslag/down-up-return walk-tree fst acc)))))
  (unslag/down-return walk-tree synd init))
```

Figure 7.14: Implementations of three unfiltered syndrites.

is that the internal `map-tree` manipulates a tree shaped skeletal node rather than a linear one. The `tree` constructor makes a tree out of an element and a list of subtrees; the `elt` and `kids` selectors extract these components from a tree. `MapD` is recursively mapped over the subtrees using OPERA's higher order `map` procedure. The base case that terminates the recursion is applying `map` to an empty list of children.

`Down-scanD` is a straightforward parallel down scanner that resembles the `down-scanQ` sliver for synquences.

Recall that `preD` is a pre-order accumulator that takes a direction argument (`dir`) in addition to the usual initial value, combiner, and slag. `PreD` is implemented as a pair of mutually recursive procedures, `walk-tree` and `walk-trees`. `Walk-tree` returns the accumulated value from a pre-order walk of a tree, while walk-trees returns the accumulated value from a pre-order walk of a list of syndrites. The `permuter` procedure used in the body of `walk-tree` permutes the children of a tree as specified by the direction.

The implementation of `preD` is carefully crafted to preserve important operational characteristics of a pre-order walk. To preserve tail recursion, the case where `walk-trees` is given a singleton list is handled specially; `walk-trees` is never called on an empty list. Returner-supplying versions of the slag abstractions are used to constrain the pre-order walking operations to happen at the right time with respect to other slivers that might be manipulating the same syndrite.

The simplicity of the above examples is somewhat deceiving. Many syndrite slivers, especially the scanners, are *very* complex and have repeatedly resisted simplification attempts. For example, the pre-order scanner is so intricate that it doesn't even fit on a single page! The scanners are particularly difficult to write because they must immediately return new syndrite structure while at the same time laying down the proper dependencies and time constraints for the accumulation computation that will take place later. Thus far I have been unable to find appropriate abstractions that elegantly express these intricate processes.

Another fly in the syndrite ointment is filtering. The two-flag synquence filtering technique does not solve the reusability problem for general syndrite filtering. In the synquence case, an element is filtered either in the *down* phase or the *up* phase. But in the syndrite case, filtering can happen *between* subcalls as well. This greatly complicates the design of filtered syndrites, and I have not worked out the details. The interactions between tree shapes and filtering remains an area for future study. The current implementation of SYNAPSE uses a straightforward approach to gap representation that is sufficient for the simple examples tested so far.