Chapter 5

Synchronized Lazy Aggregates

In Chapter 3, we saw how existing techniques for programming in the signal processing style exhibit tensions between modularity and control. This chapter introduces a new technique, *synchronized lazy aggregates*, that relaxes some of these tensions. This technique makes concrete the notion, first presented in Chapter 2, that slivers are slices through a monolithic recursive computation.

5.1 A Lock Step Processing Model

Synchronized lazy aggregates are based on a model of processing in which slivers compute in a lock step manner to simulate the behavior of monolithic recursive procedure. Here I present a brief motivation for and overview of the lock step model. The rest of the chapter fleshes out the details of how lock step processing can be achieved.

The lock step model is designed to satisfy two important goals:

1. *Operational faithfulness*: A sliver network as a whole should simulate the operation scheduling and space behavior of a monolithic computation. A mechanism may use additional operations and storage for management purposes as long as it maintains the monolithic computation's order of growth in space and time.

2. *Reusability*: The slivers should share a standard interface so that they can be recombined in a mix-and-match way to model a wide range of computations.

We begin by considering the following question: *Why is it so difficult to model the storage behavior and operation scheduling of a monolithic procedure using exisiting SPS techniques?* To answer this question, we first need to see how operational control is achieved in monolithic procedures. Then we can investigate what goes wrong in the modular case. Finally, based on our analysis, we can propose a fix.

5.1.1 Strict Calls Provide Control

The desirable operational properties of monolithic recursive procedures are due to strict procedure calls. Strictness means that all argument values must be computed before the procedure is applied. Every strict call defines a barrier that clearly delineates the operations that must be performed before the application from the operations that must be performed after the application. For instance, in the monolithic versions of `mean-age` (see Chapter 2), strictness guarantees that each iteration of the loop will read the next record from the database, add the age of the current record to the running sum, and increment the running length. The operations of the next iteration cannot be performed until all the operations of the current one have been completed. In this way, strictness effectively manages the interleaving of operations from separate idioms.

In contrast, non-strict strategies (lazy and eager evaluation) decouple the time-based ordering of the argument computations from the time of the procedure application (see [HA87]). While non-strictness is a powerful and useful language feature ([Hug90, Tra88, Hal85, Mil87]), the lack of effective barriers thwarts efforts to reason about operational details like storage requirements and operation order. For example, both lazy and eager strategies give rise to insidious space leaks and introduce new complexities in programs with side-effects.

It is important to note that non-tail calls define *two* barriers. The *down barrier* is the barrier described above that delineates argument computation from the computation of the procedure body. The *up barrier* delineates the computation of the procedure body from the computation that uses the results of the call. In the tile diagrams of Chapter 4, a call boundary represents both the down and up barriers.

Up barriers can constrain the order of operations performed in the up phase of a computation. For example, consider the following procedure:

```
(define (sum&list-squares lst)
  (if (null? lst)
      (list 0 '())
      (mlet ((sq (square (car lst)))
             ((sum sqrs) (sum&list-squares (cdr lst))))
        (list (+ sq sum) (cons sq sqrs)))))
```

Given an input list, the procedure returns two results (packaged as a list): (1) the sum of the squares of the numbers in the input list and (2) a list of the squares of the numbers in the inputs list. In the down phase of a computation generated by `sum&list-squares`, each call is preceded by one occurrence each of `null?`, `car`, `cdr`, and `square`. In the up phase of the computation, each return is preceded by one occurrence each of `list`[1], `cons`, and `+`. Here the up barrier of the recursive call forces the list collection and summation idioms to proceed in lock step.

5.1.2 Distributing Strict Calls Loses Control

The main problem with decomposing a monolithic procedure into slivers is that this eliminates the barriers that provide control. Consider Figure 5.1, which depicts the computations associated with a monolithic program and the associated sliver program. Each dotted horizontal line represents a call boundary. The decomposition process splits each monolithic call boundary into one boundary for each sliver (the labels emphasize which boundaries match up).

The control supplied by the monolithic call are lost when it is distributed over the slivers. Whereas the down and up barrier of the monolithic call forces the idioms to work in lock step, the corresponding barriers of the slivers are only loosely coupled. Modulo dataflow dependencies, nothing prevents one sliver from racing ahead of its neighbors. For example, sliver S_1 may cross the down barriers of A, B, C, and D before S_2 has even crossed A. The lack of a shared barrier results in two problems:

1. *Unsynchronized Operations:* In the sliver computation, the corresponding operations of different idioms are no longer guaranteed to be synchronized. It is no longer possible to rely on properties that depend on the synchronized order.

[1]In actuality, `list` performs some `cons`es, but we'll treat it as a primitive for now.

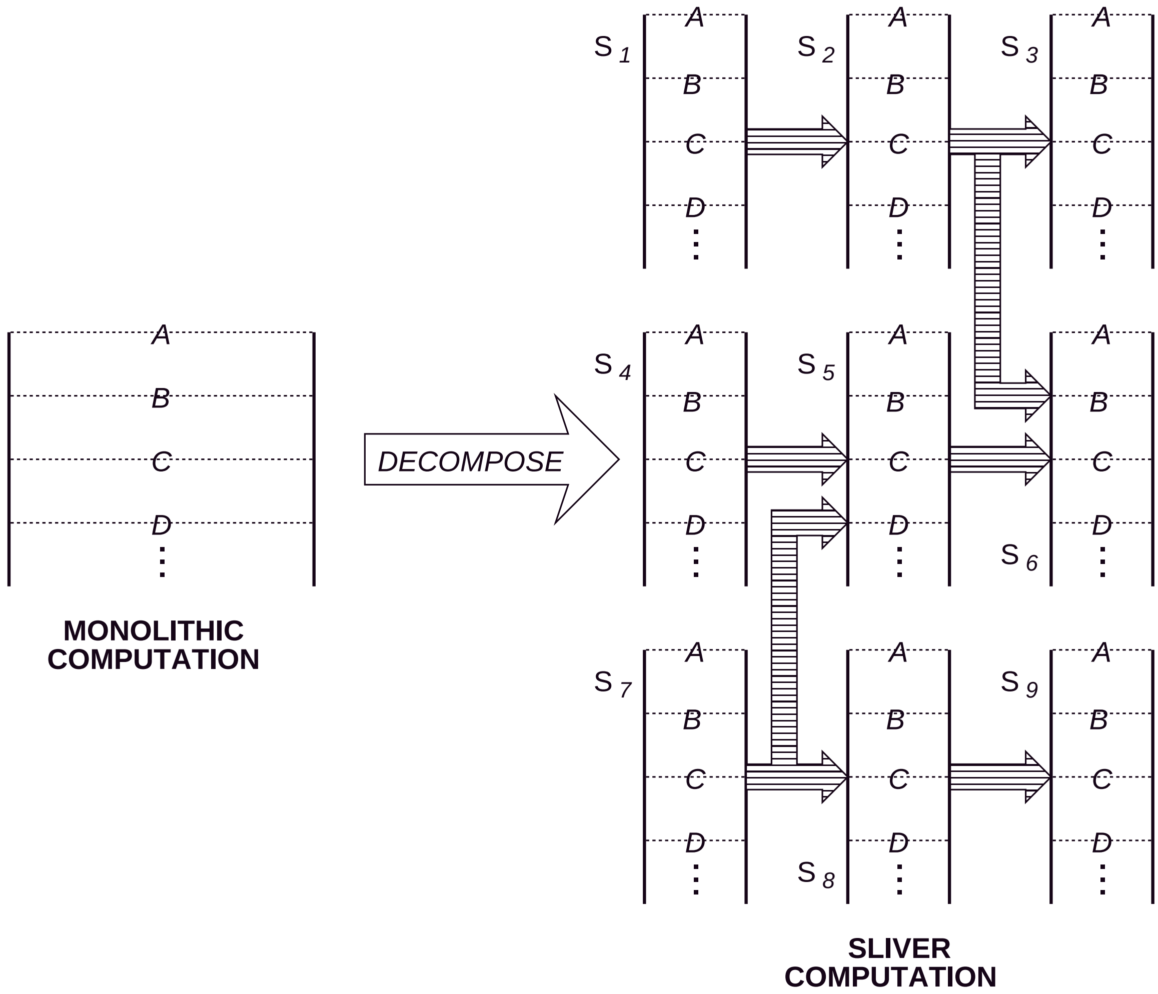


Figure 5.1: Decomposing a monolithic computation into a network of slivers.

2. *Buffering of Intermediate Results*: If a producing sliver races ahead of a consuming sliver, then the values produced by the producing sliver must be buffered somewhere until the consuming sliver consumes them. This buffering requires storage beyond that inherently implied by the computation.

In practice, the problem of buffering is far more serious than whether operations are synchronized. But the two problems are closely related. If the operations are not synchronized appropriately, then buffering will be required.

The space of possible operation schedules (i.e., orderings of operations) that are consistent with the sliver computation are generally much larger than the schedules consistent with the monolithic computation. Each of the SPS techniques studied in Chapter 3 picks a particular schedule from this larger set. For example, a strict aggregate approach chooses to perform all operations in one sliver before moving on to another one. None of the techniques is guaranteed to pick a schedule that is consistent with the monolithic computation. While the concurrent approaches (e.g., the concurrent channel technique and Hughes's `par/synch` technique) may pick a monolithic schedule, they are not guaranteed to.[2]

5.1.3 Simulating Strict Calls Regains Control

Intuitively, the lock step processing model guarantees desirable behavior for a sliver network by gluing the corresponding call boundaries of the slivers together so that they simulate the call boundaries of the monolithic call. Each sliver call boundary locally provides down and up barriers for the operations performed within the sliver. If the corresponding call boundaries were aligned so that all calls occurred at the same time and all returns occurred at the same time, then the sliver network would be forced to follow an operation schedule consistent with the monolithic computation. This approach clearly solves the problem of unsynchronized operations; and as long as the gluing process itself does not consume an unreasonable amount of storage, it solves the unwanted buffering problem as well.

An important wrinkle on the gluing idea concerns the handling of tail calls. Some of the sliver call boundaries may locally be tail calls. We want to guarantee that the gluing

[2] It's still possible to guarantee good space behavior without following one of the monolithic schedules.

process does not force a sliver to push stack when there is no pending work to be done. In particular, if all the corresponding call boundaries are tail calls, we want to guarantee that the entire network effectively performs a monolithic tail call. This behavior makes it possible to compose iterative computations out of iterative slivers, and has desirable consequences for general tree-recursive computations as well.

The notion of gluing together the corresponding call boundaries of networked slivers is very simple. The hard part is designing a mechanism that accomplishes it. We can argue from first principles the important properties that such a mechanism must possess:

- *Concurrency*: As Hughes has shown, any sequential evaluation strategy is insufficient for achieving desired space behavior in certain networks with fan-out [Hug83, Hug84]. The lock step model requires that the slivers are somehow executing concurrently.

- *Synchronization*: Concurrency *allows* the desired behavior but it doesn't necessarily *guarantee* it. Some form of synchronization is required to get the effect of gluing the call boundaries together. This form of synchronization must be more stringent than the synchronization associated with the other concurrent techniques we have studied. For the sliver computation diagram in Figure 5.1, for instance, both Hughes's technique and the concurrent channel technique allow sliver S_1 to be at the call labelled D while S_9 is at the call labelled A. The lock step model requires that no sliver can reach a call labelled B until all have passed through the call labelled A.

- *Non-strictness*: To simulate demand-driven evaluation in a monolithic computation, the lock step model requires that the values produced by a sliver are not computed unless they are actually needed by the consuming sliver. This implies that the model supports some form of laziness.

5.1.4 Lock Step Components

The lock step processing model is fine for simulating a single monolithic recursive procedure. But a typical computation is specified by many monolithic procedures. It is not appropriate for the slivers corresponding to one procedure to be acting in lock step with the slivers from another procedure. To deal with this situation, the model assumes that the slivers of a computation are partitioned into *lock step components*, where each lock step component consists of slivers that are intended to proceed in lock step.

Lock step components can be wired together to yield a loosely coupled network of tightly coupled parts. Such a network is depicted in Figure 5.2. The figure contains four lock step components, which are denoted by dotted outlines. Each component consists of slivers that are connected by thick cables, while the components themselves are connected by thin wires. Henceforth, a cable connecting two slivers shall be taken as a declaration that they are in the same lock step component. All other communication is accomplished by wires. As indicated by the figure, non-sliver computational devices (`X`, `Y`, and `Z`) may be attached to the wires.

5.1.5 The Details

The remaining sections of this chapter describe the details by which the lock step model outlined above can be achieved. The presentation consists of the following parts:

- *Sliver Decomposition*: Based on the ideas introduced in Chapter 4, this section describes how to decompose a monolithic computation into a network of slivers.

- *The Structure of Synchronized Lazy Aggregates*: This section motivates the requirements for a data structure that supports the lock step processing model. Most importantly, it introduces a novel synchronization technology, the *synchron*, that permits call boundaries to be glued together.

- *Slivers Revisited*: Synchronized lazy aggregates provide the raw materials for gluing call boundaries together, but the slivers are responsible for hooking everything up in the right way. This section describes the details of the gluing process.

- *Filtering*: One of the trickiest aspects of synchronized lazy aggregates is handling filtering. This section explains how it's done.

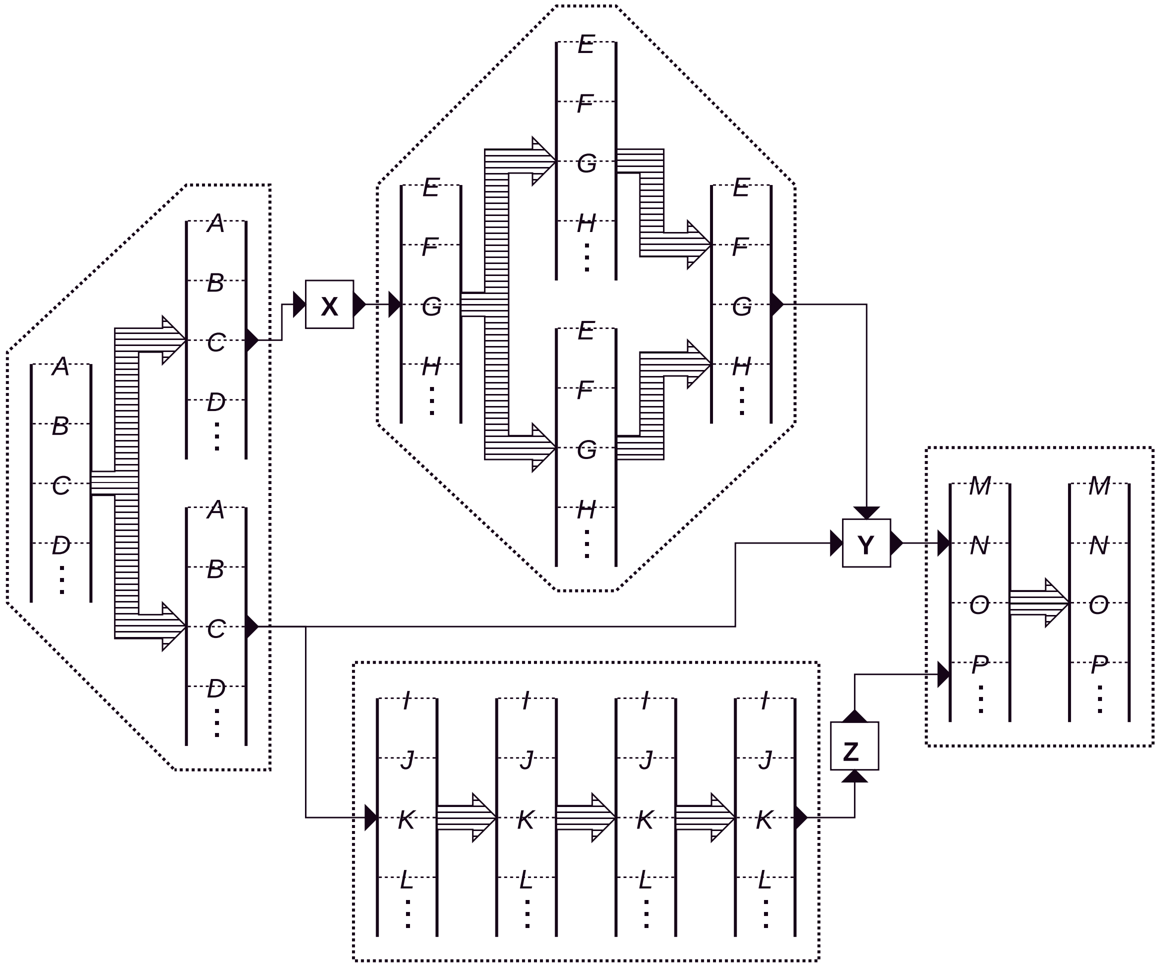


Figure 5.2: A network consisting of four lock step components. Slivers connected by thick cables form a lock step component. Lock step components are connected by thin wires.

5.2 Sliver Decomposition

Chapter 2 introduced sliver decomposition as a way of decomposing monolithic computations into slivers. Here, I use the shape concepts developed in Chapter 4 to describe sliver decomposition more precisely. As in Chapter 4, I will limit the discussion to unitilable monolithic recursions.

Without the synchronization supplied by synchronized lazy aggregates, it will not be possible to guarantee lock step behavior for the result of the decomposition. This section describes the first in a series of approximations to the final model.

5.2.1 Linear Subtiles

Figure 5.3 suggests a simple strategy for decomposing a unitilable monolithic computation into slivers: break the tile for the monolithic computation into communicating fragments called *subtiles*, and then replicate each subtile to reflect the structure of the monolithic computation. Although the figure shows a linear computation whose slivers communicate

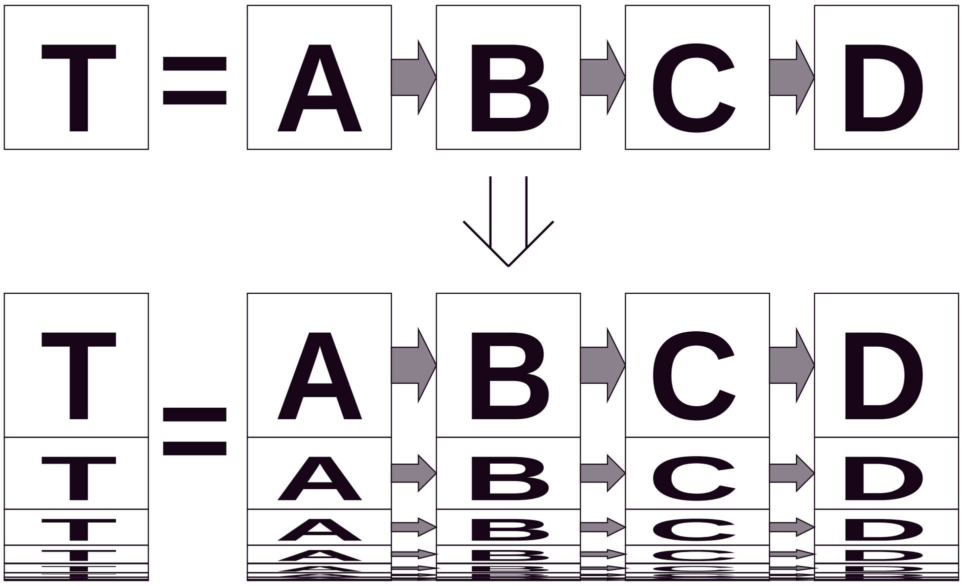
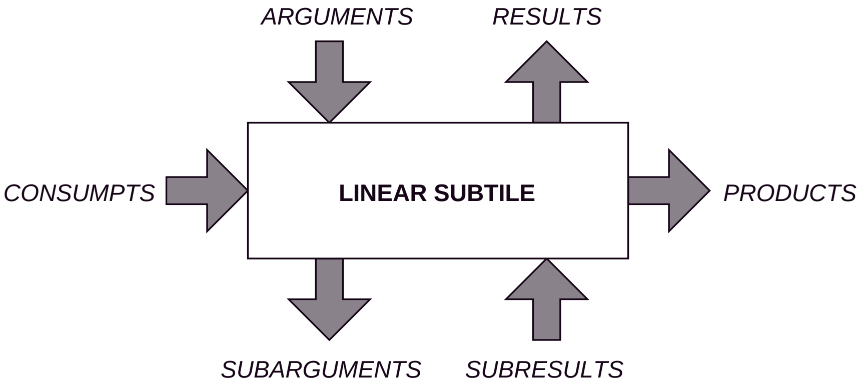


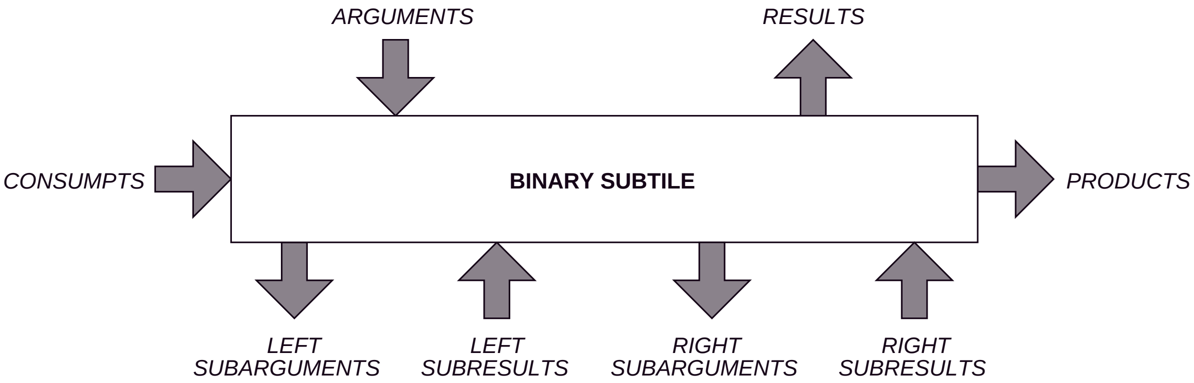
Figure 5.3: A simple strategy for sliver decomposition. If a tile (**T**) can be decomposed into communicating subtiles (**A**, **B**, **C**, and **D**), then the computation corresponding to the replicated tiles can be decomposed into the slivers corresponding to the replicated subtiles.

in a straight line, the strategy applies to tree computations and sliver networks that exhibit fan-in and fan-out.

A subtile is a generalization of a tile that can communicate horizontally with other subtiles in addition to communicating vertically with copies of itself (see Figure 5.4). The



(a) Linear subtile.



(b) Binary subtile.

Figure 5.4: Subtile interfaces.

horizontal inputs are called *consumpts* and the horizontal outputs are called *products*. As indicated by Figure 5.3, each subtile instance within a trellis can communicate horizontally only with instances of other subtiles at the same trellis location. A tile can be viewed as special kind of subtile whose consumpts and products are both empty.

Figure 5.5 illustrates a first cut at using this strategy to modularize the recursive sum-of-squared-evens tile from Figure 4.6. The tile is decomposed into four subtiles that correspond

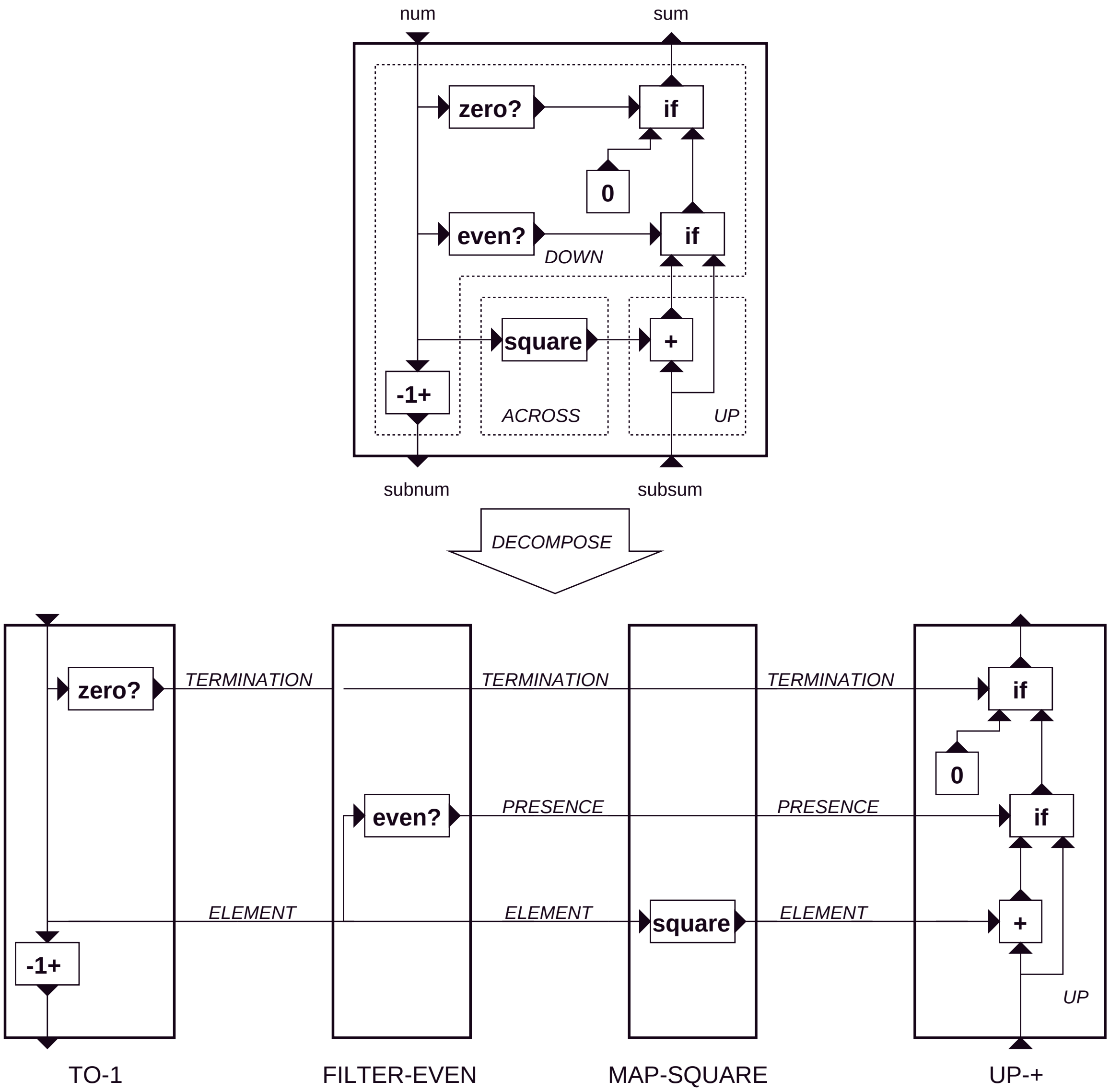


Figure 5.5: A naive decomposition of the sum-of-squared-evens tile into four subtiles.

to common programming idioms:

1. `TO-1` generates a sequence of integers from a given integer down to 1 (inclusive).
2. `FILTER-EVEN` is a sequence filter that passes only the even numbers from its input to its output.
3. `MAP-SQUARES` squares each of the elements in the input sequence.
4. `UP-+` performs a recursive sum accumulation on a sequence.

The subtiles communicate information horizontally via three classes of wires:

1. *Termination wires* transmit a boolean that indicates whether or not the end of the sequence has been reached.
2. *Presence wires* transmit a boolean that indicates whether the associated element has been passed by a filter.
3. *Element wires* transmit the current element of the sequence.

Unfortunately, the decomposition depicted in Figure 5.5 suffers from two very important problems:

1. *Non-standard interfaces*: Reusability is hindered because the components don't all share the same interface. In particular, some subtiles handle presence wires, while others do not. For instance, `FILTER-EVEN` produces a presence wire even though it does not consume one. Two such filters cannot be cascaded — a disaster for composability. Naive decompositions for other tiles presented earlier can easily lead to subtiles in which some of the three wires are absent or duplicated.
2. *Inadequate control*: Operational faithfulness is threatened by the inadequate specification of when subtile operations or subcalls are performed. For example, the simple operational rules outlined previously dictate that, within the tile, the `-1+` is performed only after the subresult has been requested. However, in the decomposed version, different subtiles have different subcall boundaries. How does the `TO-1` tile know when

to perform `-1+`? Similarly, the `MAP-SQUARE` subtile does not explicitly specify that the `square` operation is performed only on even integers; yet this is clear in the tile. And none of `TO-1`, `FILTER-EVEN`, and `MAP-SQUARE` indicate *when* a subcall should be initiated. Since slivers are supposed to mimic the operational behavior of a monolithic computation, this underspecification of control is disconcerting.

The improved tile decomposition in Figure 5.6 addresses some of these concerns. First, all of the subtiles share a standard interface in which each sequence element is represented by a triple of termination, presence, and element wires. The presence wire is manipulated by all subtiles in a composable way: the generator initializes the presence wire to a true value (`#t`); the filter combines it with local filtering information; the mapper passes it along untouched; and the accumulator uses it to control accumulation. The subtiles also maintain the invariant that when the presence wire carries the false value, the element wire carries a distinguished *gap* value (written `#g`). The gap value indicates a position in the sequence where an element has been filtered out.

Second, control details are much more explicit in the subtiles of Figure 5.6 than those of Figure 5.5. Each subtile has its own copy of the termination control assembly from the tile's control arm. The generator, filter, and mapper are all assumed to ultimately return the boolean truth value (`#t`) just so that the request for this value can be used as a means of specifying local control. For example, the `-1+` operation in the `TO-1` tile will only be performed after its local subresult has been requested. The fact that `MAP-SQUARE`'s `square` operation is guarded by a test of the presence wire is another example of control being made explicit.

The more sophisticated decomposition still leaves some important control problems unsolved. The subcall boundary of the whole tile effectively synchronizes all subresult requests with all subargument evaluations. When the subcall boundary is distributed across subtiles, one sliver can easily race ahead of another. For example, nothing prevents the sophisticated `TO-1` sliver from merrily generating all the numbers in the sequence before any of the other slivers have processed the first one. This would imply the need for storage buffers between the components, which is exactly the kind of behavior the lock step model is supposed to avoid. We will see shortly how synchronized lazy aggregates solve this problem. Until then,

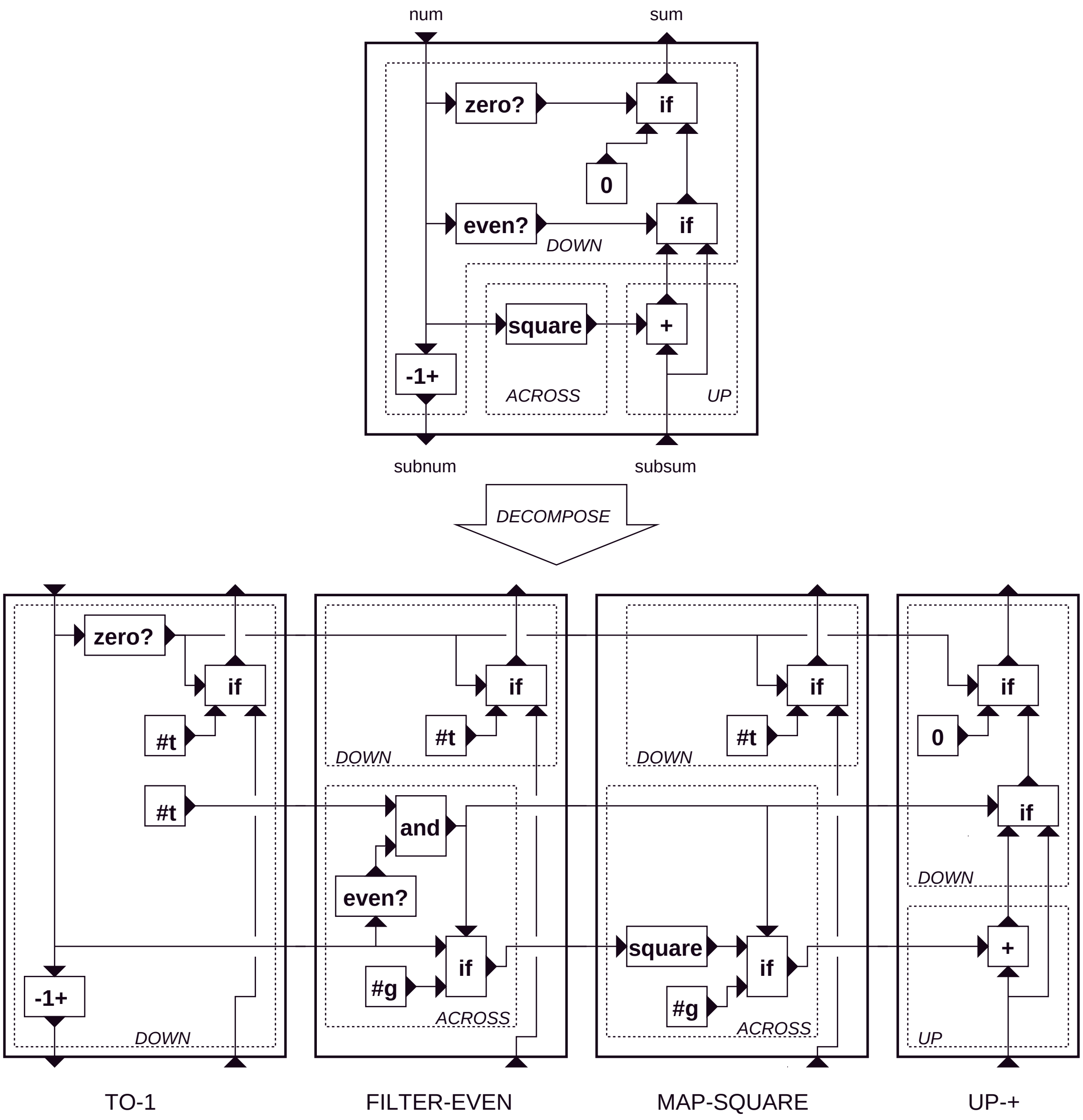


Figure 5.6: A sophisticated decomposition of the sum-of-squared-evens tile into four subtiles.

we will assume that connected slivers magically compute in a synchronous manner.

Another drawback of the sophisticated decomposition is that the four subtiles obviously do more total work than the single tile. Some examples:

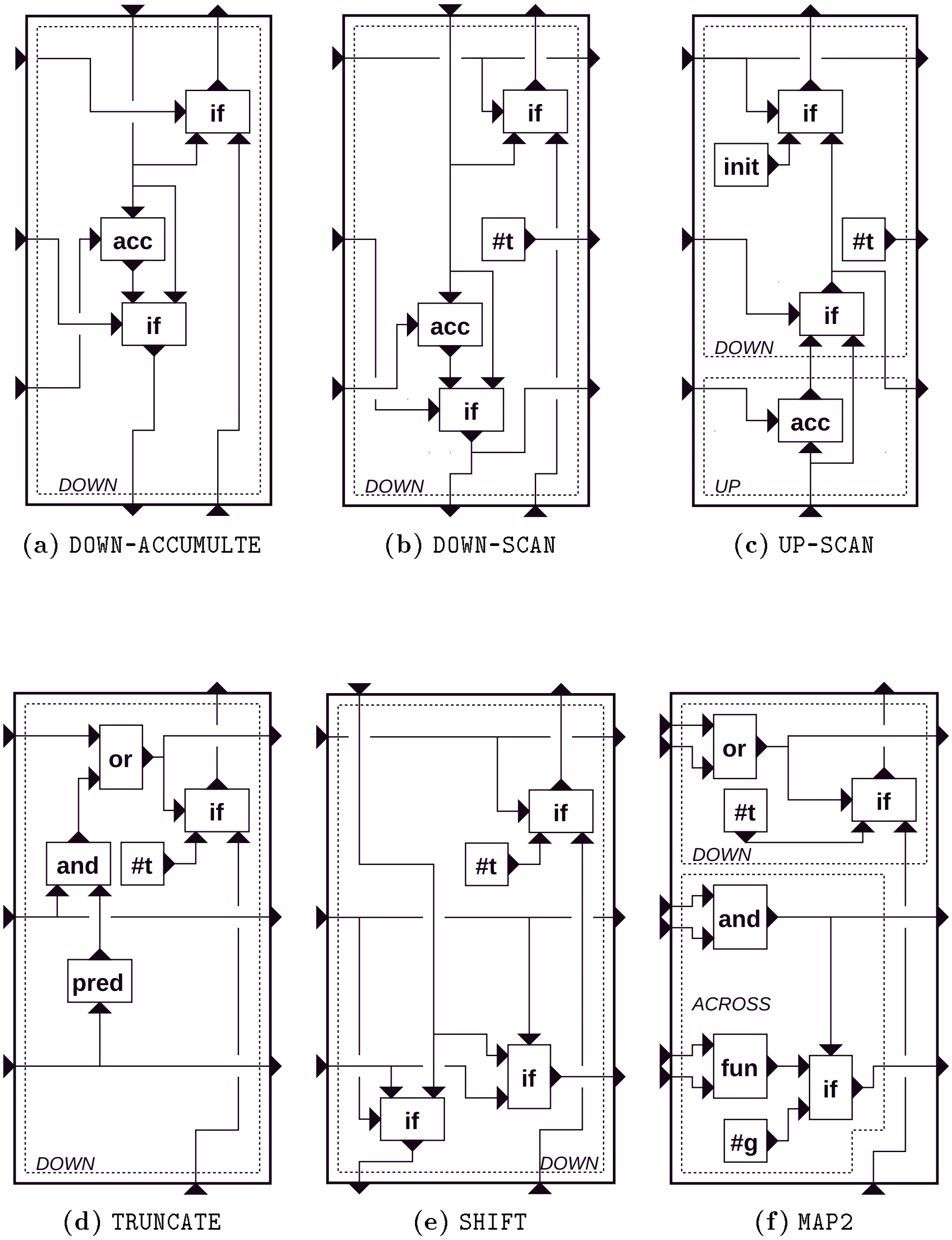
- The termination test is performed once by the tile, but four times by the subtiles.
- Manipulating presence wires in a standard way requires extra `if` and `and` operations.
- The communication and synchronization between subtiles (introduced later) will undoubtedly require additional overhead as well.

It is possible that executing the slivers on multiple physical processors could reduce these overheads. But this is unlikely[3] and, more relevant, unimportant. Most any attempt to provide standard interface within a given system is bound to result in overheads. These are often justified by a significant gain in simplicity. The kinds of overhead enumerated above can be insignificant when compared to the mental overhead of having to express computations in a non-modular way. The *real* benefit of sliver decompositions is that they suggest new ways of analyzing and synthesizing computations.

A wide range of useful subtiles can be designed with the standard three-wire-per-element interface introduced above. Figure 5.7 illustrates some more linear subtiles. Subtile **(a)** is an iterative accumulator; if the `UP-+` subtile of Figure 5.6 were replaced by an instance of this down accumulator, the network would correspond to an iterative sum-of-squared evens rather than a recursive one. Subtiles **(b)** and **(c)** are *scanners* that emit as products the intermediate accumulated values in a *down* or *up* accumulation.[4] Note that a scanner product is always present even if the corresponding consumpt was not present. Subtile **(d)** is a *truncater* that terminates a sequence as soon as `pred` is true of an element; the (short-circuit) `and` prevents `pred` from being applied to a gap. Subtile **(e)** is a *shifter* that moves non-gap elements down to the next non-gap rung in a linear trellis. Subtile **(f)** maps a

[3]Since the slivers do so little computation between communication and synchronization events, parallel execution of slivers is likely to offer few practical performance benefits.

[4]There are two very different meanings of “scan” in the literature on the signal processing style of programming. In the data parallel literature, “scan” refers to a partial accumulator [Ble90, Sab88, GJSO92]. In Waters’s series package, though, “scan” refers to a kind of generator [Wat90]. I adopt the former meaning here.



(a) DOWN-ACCUMULTE (b) DOWN-SCAN (c) UP-SCAN

(d) TRUNCATE (e) SHIFT (f) MAP2

Figure 5.7: A gallery of linear subtiles.

function `fun` over the elements of two consumpts to give a single product. The termination wires of `MAP2` are joined with an `or` so that the product terminates as soon as one of the input terminates. The presence wires of `MAP2` are joined with `and` so that the product is only present (i.e., the function is only applied) if both consumpts are present.

The visual complexity of subtile diagrams has limited this discussion to very simple sliver networks. Examples of more complex networks specified in a textual form appear in Chapter 6.

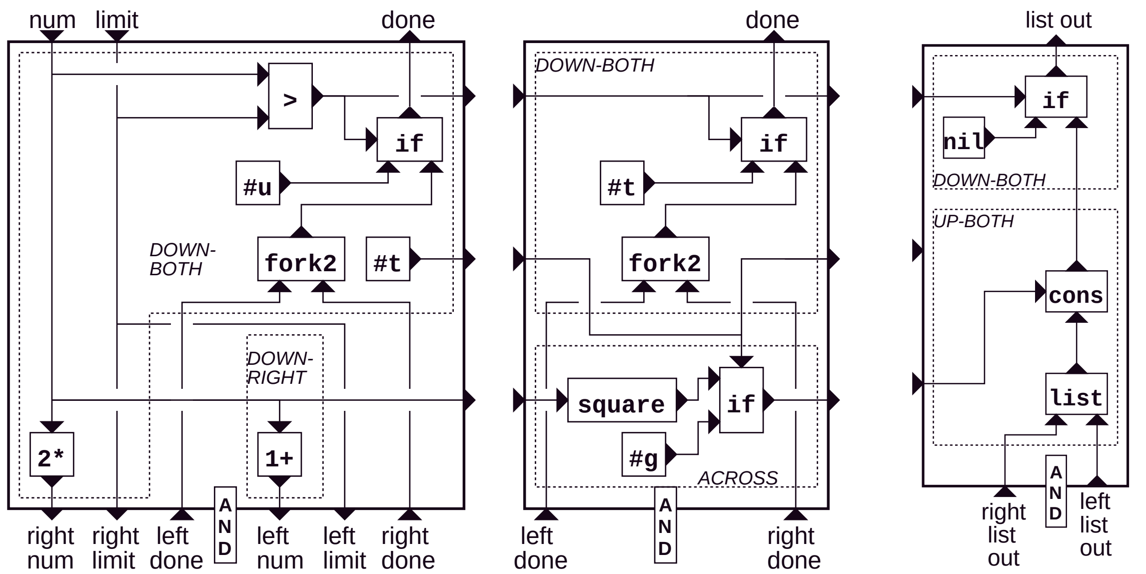
5.2.2 Binary Subtiles

The kinds of subtile decompositions introduced above extend naturally to tree-shaped computations. Figure 5.8 shows some sample binary subtiles. As in the linear case, each tree node is represented by a triple of termination, presence, and element wires. Conceptually, the `BREADTH-INDEX` generator (subtile **(a)**) can be combined pairwise with the four accumulating subtiles (**(c)** – **(f)**) to yield the tiles in Figure 4.19. Accumulating the squares of the elements of a breadth index tree could be accomplished by inserting the `BINARY-MAP-SQUARE` tile between generator and accumulator.

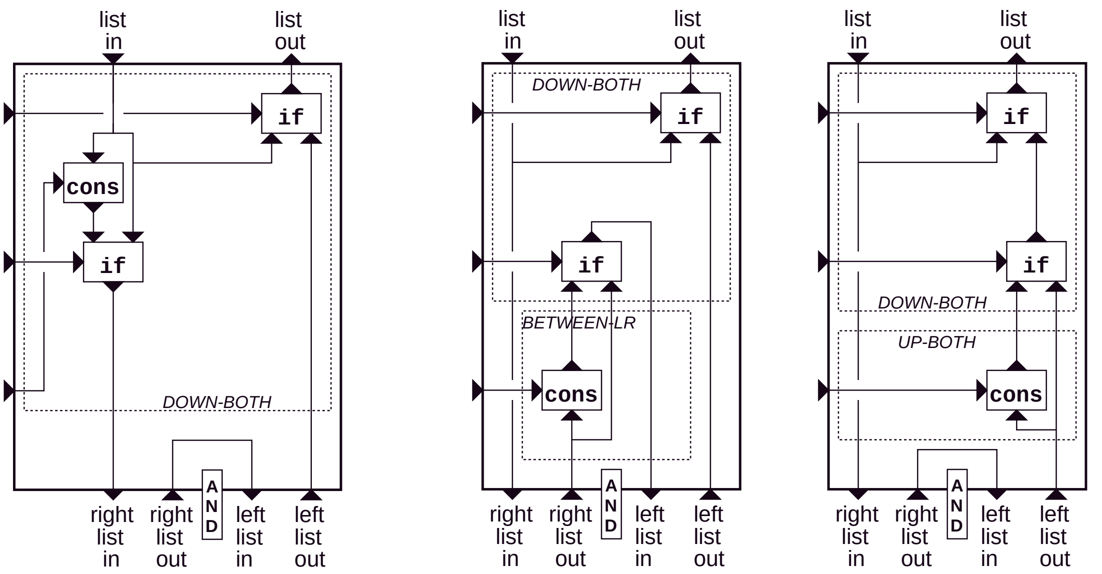
Perhaps the biggest surprise in Figure 5.8 is the presence of the non-strict `fork2` node in both `BREADTH-INDEX` and `BINARY-MAP-SQUARE`. Neither of these subtiles naturally accumulates any meaningful value. The duty of the `done` port in both cases is not to return a value but to propagate requests to the subcalls. In order to maintain the essential parallel, non-accumulating character of these subtiles, it is imperative to propagate requests in a way that both (1) does not specify the order of the subcalls and (2) does not leave behind a pending operation to be performed. Single-threading the request through the two subcalls would satisfy (2) but not (1). On the other hand, the eager, parallel behavior of `fork2` fits the bill perfectly. So while we will not permit whole tiles to use `fork2`, we will use `fork2` in a restricted way to represent subtiles with a *binary down* character.[5]

The only other surprise is `COLLECT-TREE`'s treatment of gaps. Note that the presence wire is never checked! When performing a *parallel up* accumulation on a tree with gaps,

[5] Conceptually, when subtiles are glued together to form tiles, there will necessarily be some subtile with an accumulation component that renders the `fork2`s unnecessary. So no tile composed out of subtiles requires a `fork2`.



(a) BREADTH-INDEX (b) BINARY-MAP-SQUARE (c) COLLECT-TREE



(d) BINARY-PRE-CONS (e) BINARY-IN-CONS (f) BINARY-POST-CONS

Figure 5.8: Sample binary subtiles.

it is generally necessary to specify *two* accumulators: one for the case where the current element is present, and one for the case where it is not. `COLLECT-TREE` makes the simplifying assumption that it's alright to include a gap value in the returned tree. But if a sum accumulation were being performed instead, the innards of the sliver would be more complex.

5.2.3 Subtile Shapes

Subtiles can be classified by a notion of shape similar to that for tiles. As with tiles, the operations of subtiles can naturally be partitioned into shards according to their relationship with subcall initiation and return. For example, such shards are labelled in the subtiles of Figures 5.6 – 5.8.[6]

The subtile shards stand out from tile shards in several ways:

- In Figure 5.6, the *down* shards of `FILTER-EVEN`, `MAP-SQUARE`, and `UP-+` consist purely of control arms. In contrast, the *down* shards from all the linear tile examples presented thus far have all been connected to the subcall boundary via a subargument wire. Similar remarks hold for the *down-both* shards of `BINARY-MAP-SQUARE` and `COLLECT-TREE` in Figure 5.8.

- Subtiles can have a shape different from that of the tile from which they are derived. For example, the `TO-1` subtile in Figure 5.6 has a fundamentally *down* shape even though the tile as a whole has an *up* shape. This captures the intuition that the generator fragment of the tile works in a fundamentally iterative way. Similarly, the `BREADTH-INDEX` subtile in Figure 5.8 has a fundamentally *binary down* shape even though none of the tiles in Figure 4.19 (page 154) has *binary down* shape.

- Due to horizontal communication, subtiles support patterns that are not possible for whole tiles. For example, `FILTER-EVEN` and `MAP-SQUARE` in Figure 5.6 each has non-trivial *down* and *across* shards but a trivial *up* shard. This pattern is not possible in a tile because the results of the *across* shard would necessarily be consumed by

[6] For simplicity, trivial shards have been omitted in these and subsequent figures.

a non-trivial *up* shard. In subtiles, an *across* shard can find an alternate outlet as subtile products. We shall use the term *across shape* to refer to a linear subtile that consists only of an *across* shard in addition to a simple termination-controlling *down* shard. Similarly, `BINARY-MAP-SQUARE` will be said to have a *binary across* shape.

Except for the *across* shapes mentioned above, subtile shapes are determined in a manner similar to tile shapes. For example, the shape definitions in Table 4.2 (page 155) hold for binary shapes.

An interesting feature of *across* subtiles is that their operational behavior is context dependent. Consider the three different uses of the `MAP-SQUARE` subtile in Figure 5.9:

1. In network **(a)**, `MAP-SQUARE` feeds a *down* accumulator that forces the squaring operation to be performed before subcall initiation.

2. In network **(b)**, `MAP-SQUARE`'s position between a *down* generator and an *up* accumulator leaves the squaring operation unconstrained with respect to the subcall.

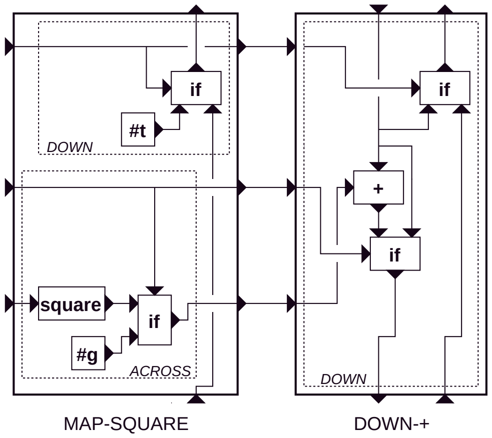
3. In network **(c)**, `MAP-SQUARE` is wedged between an *up* scanner and an *up* accumulator, which force the squaring operation to occur after the subcall returns.[7]

These examples illustrate that an *across* subtile operation does not necessarily act like an *across* operation in a network of subtiles; its behavioral fate is determined by the surrounding context. In contrast, *down* and *up* subtile operations are unaffected by context. So while cascading two linear subtiles always yields another linear subtile, it may require the relabelling of *across* shards.

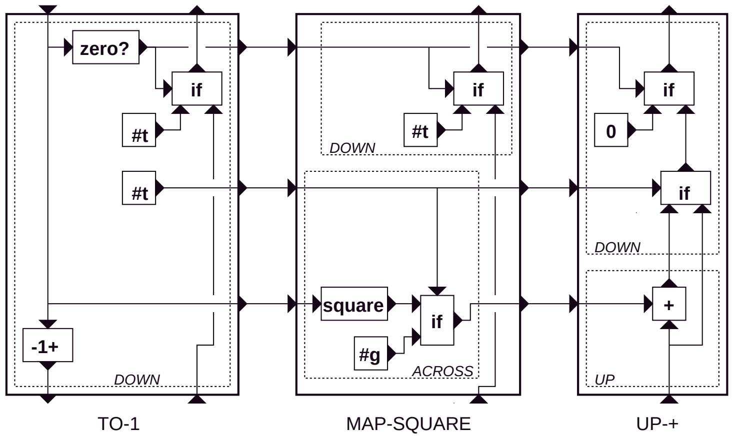
5.2.4 Sliver Computations

Just as replicating a tile throughout a trellis gives rise to a monolithic computation, replicating a subtile throughout a trellis conceptually generates a *sliver computation*. Like monolithic computations, sliver computations have a dynamic shape property determined by the time structure of their call events.

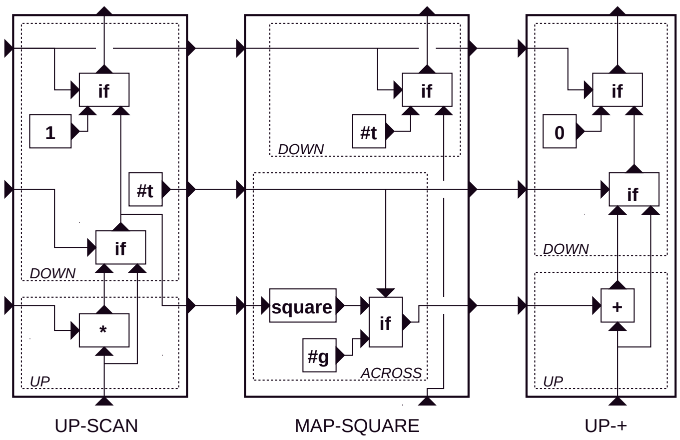
[7]There is actually a nasty technical problem lurking in network **(c)** that will be discussed in Section 5.5.



(a) `MAP-SQUARE`'s *across* shard forced down.



(b) `MAP-SQUARE`'s *across* shard unconstrained.



(c) `MAP-SQUARE`'s *across* shard forced up.

Figure 5.9: Examples showing the context-dependent nature of *across* operations within a subtile.

Due to consumpts and products, sliver computations within the same lock step component can be viewed as consuming and producing horizontally transmitted values. Because of the demand driven nature of the components out of which they are built, no product is computed unless it is actually needed by a consuming sliver. Thus, this version of the sliver decomposition technique is operationally faithful in this detail.

5.2.5 Subtile Compatibility

While subtiles share a standard interface, not all combinations of subtiles make sense. First of all, there are some obvious structural constraints:

- Connected subtiles must have the same number of subcalls. It doesn't make sense to mix linear and binary subtiles.[8]
- Connections are only allowed between result/argument ports and between product/consumpt ports. It is illegal to wire a result port to a consumpt port or a product port to an argument port.
- A product/consumpt connection is only legal if the termination/presence/element ports are correspondingly wired. That is, a triple of termination, presence, and element ports is treated as a single connection point.

All of these are local constraints that can be simply checked between any pair of subtiles.

More subtly, there are some important non-local constraints. Consider the subtile network in Figure 5.10, in which an up-multiplying scanner feeds an down-summing accumulator through a square mapper. According to the operational model, the `+` must be performed before subcall initiation in `DOWN-+`, and the `*` can only be performed after subcall return in `UP-*`. But the subcall boundaries of connected slivers are conceptually glued together to form a single boundary. So the shaded path indicates a dependency circularity in the subtile network that will cause deadlock in any computation based on this network. This circularity is non-local because it cannot be discovered by considering any connected pair of subtiles.

[8] Actually, this isn't quite true. It is possible to imagine subtiles that "convert" between linear and tree slivers.

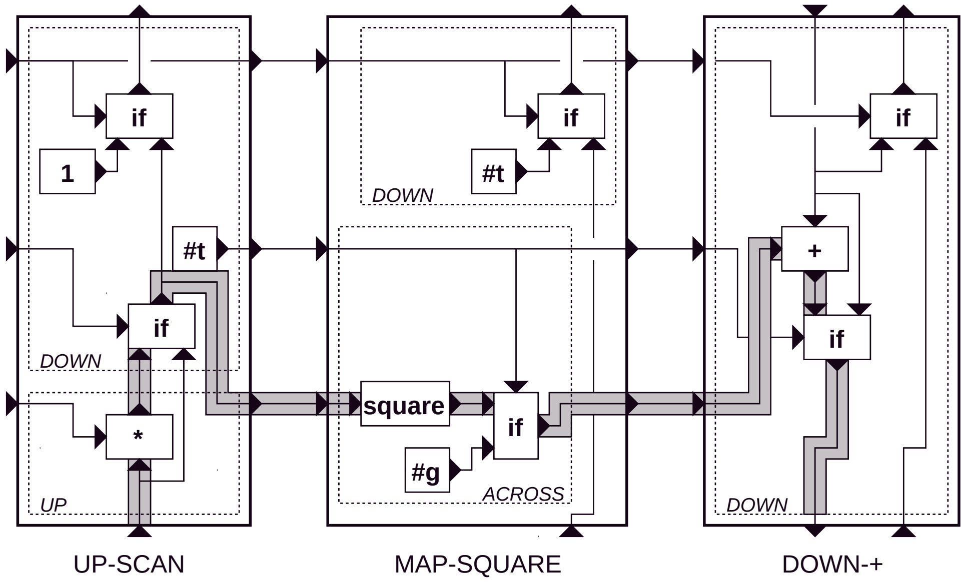


Figure 5.10: This network of subtiles is non-sensical because the corresponding tile is guaranteed to deadlock.

Another way to view the deadlock problem of Figure 5.10 is in terms of *up* and *down* phases. Recall that every unitilable linear computation has a single *down* phase followed (optionally) by a single *up* phase. The problem with any computation based on the sample network is that it attempts to force a second *down* phase to occur after the *up* phase. Intuitively, this cannot be done in a single recursive pass; it requires two recursive passes communicating via an aggregate data structure. By design, lock step components model only computations with one recursive pass, so no single subtile network can represent the intended computation. (However, it is possible to encode the computation as two lock step components.)

There are two basic approaches for detecting deadlock within a subtile network:

1. *Dynamic deadlock detection*: In a dynamic approach, deadlock is only detected during the execution of the computation generated by a subtile network. Deadlock is detected by the execution engine when no progress can be made on a computation that has not terminated.

2. *Static deadlock detection*: In a static approach, a subtile network is analyzed for potential deadlocks before it is executed. Deadlock is detected by the analyzer when it discovers a potential dependency circularity.

These approaches exhibit the classic tradeoffs found in other dynamic/static dichotomies (e.g. dynamic vs. static type-checking). Static deadlock detection is helpful for finding logical errors as early as possible. However, it requires the conceptual overhead of developing a theory of deadlock and inventing a sound deadlock detection algorithm. Moreover, deadlock is an undecidable property (see below), so any sound static deadlock analysis will conservatively ascribe deadlock to some deadlock-free computations.

In contrast, dynamic deadlock detection offers simplicity and expressibility. The existing operational model is already powerful enough to detect deadlock. And by delaying deadlock detection as long as possible, the dynamic approach finds no spurious deadlocks. On the other hand, this approach provides no guarantees that a program is deadlock-free. And while a system can provide debugging information on a deadlocked state, it can be difficult to trace the deadlock back to its source.

Since shapes encode dependency information, it should be possible to develop a form of static deadlock detection based on subtile shapes. For example, in the case of linear subtiles, we expect that an *up* tile feeding a *down* tile can cause deadlock, but we know that it is always safe for a *down* subtile to feed another *down* subtile. Similarly, with binary subtiles, we expect that left-to-right and right-to-left subtiles never mix (because this would imply incompatible traversals of the computation tree). However, *binary down* subtiles should be able to precede any binary subtile, while *binary up* subtiles should be able to follow any binary subtile. We shall refer to this shape-based approach for deadlock detection as *shape checking*, in analogy to type checking.

Although the possibility of shape checking is alluring, I have not yet developed an elegant formulation for it. While the combination rules mentioned in the previous paragraph are helpful heuristics, the notion of subtile shape defined earlier is too coarse for handling the nuances of deadlock in a reasonable way. Presumably, something more like the circularity detection analysis of attribute grammars [DJL88] is required here, but I will not explore this avenue. For simplicity, I will assume dynamic deadlock detection throughout the rest of this document.

I conclude this section with the sketch of a proof that deadlock is undecidable. Consider the simple computation diagram sketched in Figure 5.11. It contains a single-argument procedure `F`, `not` (the boolean negation operator), and two copies of an `EITHER-SCAN-+` sliver. The `EITHER-SCAN-+` sliver is designed to behave like `DOWN-SCAN-+` if its argument is

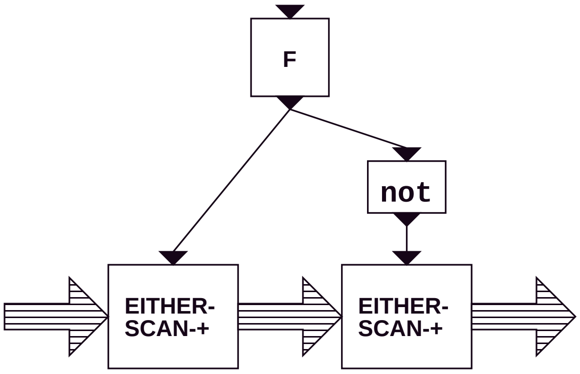


Figure 5.11: A network used to illustrate the undecidability of deadlock detection.

true, but behaves like `UP-SCAN-+` if its argument is false. Assuming that `F` is deadlock-free, the network can only exhibit a deadlock when the output of `F` is false. So the question of whether the network exhibits deadlock is reducible to the question of whether a given procedure is identically true for all inputs. The latter is obviously undecidable in general.

5.3 The Structure of Synchronized Lazy Aggregates

Subtiles are an abstract solution to the problem of partitioning a monolithic recursive computation into reusable fragments. There are many concrete questions they do not address:

- How is the computational pattern specified by a subtile replicated to generate a sliver computation?
- How are the products of one subtile communicated as consumpts to another subtile?

- What synchronization mechanism glues the corresponding call boundaries of connected subtiles together?

All of these issues are resolved by *synchronized lazy aggregates* (abbreviated as *slags*) a novel class of lazy data structures that carry explicit synchronization information:

- Since it represents a fragment of a monolithic recursive procedure, a sliver is naturally implemented as a recursive procedure that manipulates slags. The usual recursion mechanism accounts for subtile replication.

- As an aggregate data structure, a slag is well-equipped to communicate termination, presence, and element information from one sliver to another. The lazy nature of slags guarantees that communication happens in a demand-driven manner that faithfully models the operational behavior of a corresponding monolithic procedure.

- The synchronization information carried by a slag manages the lock step computation of connected slivers. Every sliver that abides by the rules of a synchronization contract is guaranteed not to race ahead or lag behind its compatriots.

5.3.1 Overview

We know from Chapter 3 that aggregate data is an elegant and easy-to-use technique for communicating values between sliver-like program components. We will adapt the technique to solve the problem of gluing together the corresponding call boundaries of all the slivers in a lock step component. The call boundaries influence the aggregate data approach in two fundamental ways:

1. *Synchronization*: In order for slivers to engage in lock step processing, the aggregate data structures connecting them must transmit some sort of synchronization tokens that represent the down and up barriers of the desired composite call boundary. I will call these tokens *synchrons*.

2. *Laziness:* Elements computed above a call boundary will have to be successfully communicated between slivers before elements below the call boundary have been

computed. This means that an aggregate connecting two slivers is generally only partially determined at a given point in time. A form of laziness will be used to represent the time-unfolding nature of the aggregate.

We will use the term *synchronized lazy aggregate* (abbreviated *slag*), to refer to an aggregate structure that addresses these two issues. Slags are a particular implementation of the cables that appear in sliver diagrams.

Intuitively, a slag is a data structure that represents the values of a program variable over time. Waters uses the term *temporal abstraction* [Wat78] to refer to this notion. In fact, his series data structure [Wat90, Wat91] is a particular instance of the more general synchronized lazy aggregates. Whereas a series represents the successive values of the state variable of a loop, synchronized lazy aggregates can represent the conceptual tree of values taken on by an identifier within an arbitrary recursive procedure. Another way to say this is that series corresponds to a register while synchronized lazy aggregates correspond to a register plus a stack.

Slags can fruitfully be viewed as a hybrid between lazy aggregates and synchronous communication channels for concurrent processes. Like lazy aggregates, slags are compound, potentially tree-shaped, data structures whose parts are not computed until they are required. Like synchronous communication channels, slags synchronize separate threads of control and manage inter-process storage resources. (However, unlike many other synchronization models (e.g. [Hoa85, Mil89]), slags decouple communication and synchronization.)

5.3.2 Synquences and Syndrites

For simplicity, we will focus on two particular kinds of slags: *synquences* and *syndrites*. A synquence (*syn*chronized se*quence*) is a synchronized lazy list while a syndrite (*syn*chronized den*drite*) is a synchronized lazy tree. Synquences represent the cables between linear slivers, while syndrites represent the cables between tree-shaped slivers. While trees generally subsume lists as a special case, we distinguish the two structures because synchronized lists have important properties that not shared by synchronized trees. In particular, synchronized lists permit forms of iteration and filtering that are not supported by synchronized trees.

Intuitively, a synquence is a linear chain of value-bearing nodes linked together by pointers annotated with synchronization information. Figure 5.12 compares the structure of a traditional list to the structure of a synquence. In a traditional list **(a)**, the component values (`2`, `3`, and `5`) hang off of a chain of *skeletal nodes* consisting of three pairs and a `nil`. The synquence structure **(b)** is similar, except that each pointer to a skeletal node in **(a)** has been replaced by a compound structure holding onto two synchrons: a *down synchron* representing the down barrier of a call boundary, and an *up synchron* representing the up barrier of a call boundary. Structure **(c)** is an abbreviation of the synquence **(b)** that emphasizes how the synchronization information has essentially annotated the pointers of the list structure. A *barrion* stands for the pair of down and up synchrons.

A pointer annotated with a barrion will be called a *syn*chronized poin*ter*, or *synter* for short. The term "synquence" refers not to a skeletal node, but to a synter that points to a skeletal node. Synters only connect the skeletal nodes that form the backbone of the aggregate; component values hanging off of the backbone are held by unannotated pointers. A valueless `nil` node terminates a finite synquence; an unterminated synquence is conceptually infinite in length.

In a synquence, the structure between two synters encodes the information in one triple of termination, presence, and element wires. The termination boolean is represented by whether or not the skeletal node is `nil`. The presence boolean and element are assumed to be encoded in the value held by a non-`nil` node.

The structure of syndrites is similar to the structure of synquences (see Figure 5.13). The two main differences are:

1. Syndrite nodes are connected by synters to multiple children rather than just one.

2. A finite branch of a syndrite is terminated by a `leaf node` that has a value but zero children. In contrast, the `nil` node terminating a synquence carries no value.

At first glance, representing leaves by value-bearing nodes may seem at odds with the termination/presence/element model. However, experience suggests that in a large percentage of tree-shaped computations, subtiles transmit/receive elements when the corresponding termination boolean is true. The essence of termination in tree computations is not a value-

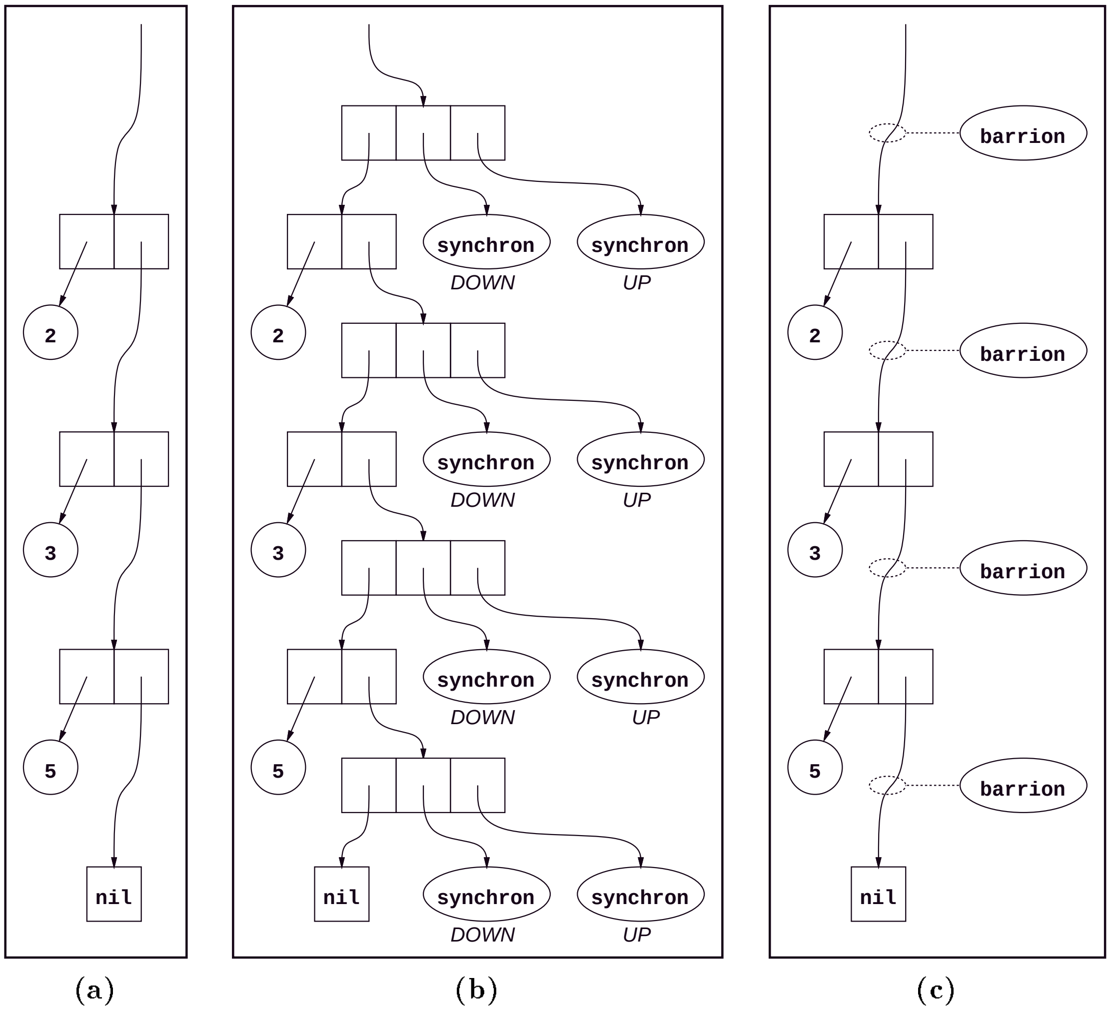


Figure 5.12: Comparison of list structure and synquence structure. **(a)** is a list of three elements. **(b)** is a synquence of three elements. **(c)** is an abbreviation of synquence **(b)**.

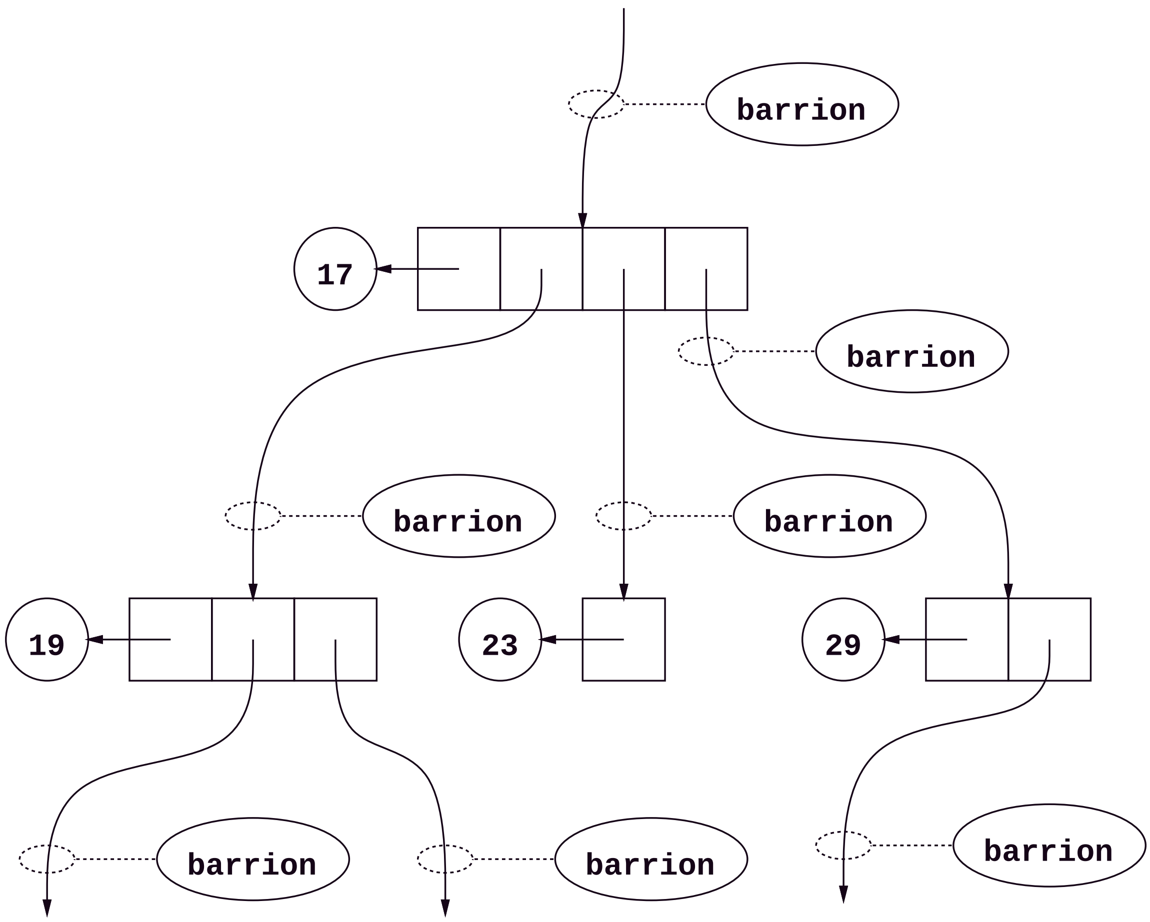
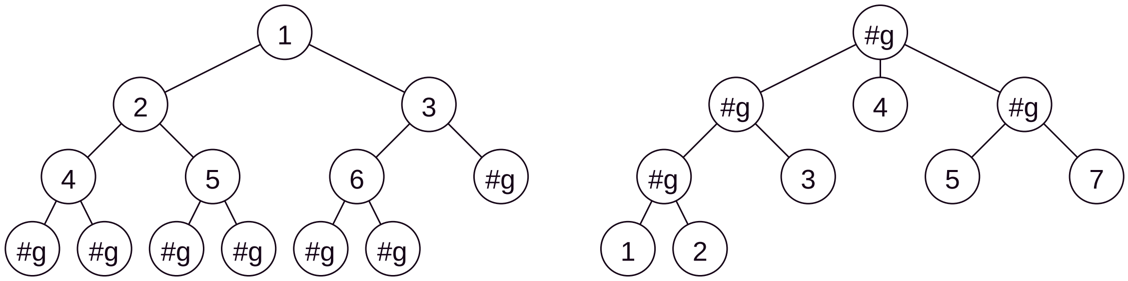


Figure 5.13: Structure of a sample syndrite.

less node, but a childless node. Trees with null leaves (like the one pictured in Figure 4.20) are represented as syndrites whose leaves carry null or gap values (Figure 5.14(**a**)). The common case of trees with valued leaves but valueless internal nodes is handled by syndrites with gaps at the non-leaf nodes (Figure 5.14(**b**)).



(**a**) A tree with valueless leaves. (**b**) A tree with valueless internal nodes.

Figure 5.14: Using gaps (`#g`) to represent valueless nodes.

5.3.3 Synchrons

A synchron is an entity that represents the down or up barrier of a call boundary. It is the fundamental mechanism by which slivers synchronize with each other to achieve lock step processing. We assume that synchrons are propagated in such a way that all slivers in the same lock step component share access to the same synchron for the same call or return event. For example, Figure 5.15 indicates the sharing of synchrons (represented by barrions) in a simple sliver network. We shall see shortly how this sharing is achieved. Right now we concentrate on what must be true of the synchrons themselves.

Synchrons support the following operations:

- *Create* makes a new synchron from scratch. This operation is invoked by generating slivers when constructing a slag.

- *Unify* glues two synchrons together so that they become the same synchron. Slivers with fan-in invoke this operation on the corresponding synchrons of their inputs. This

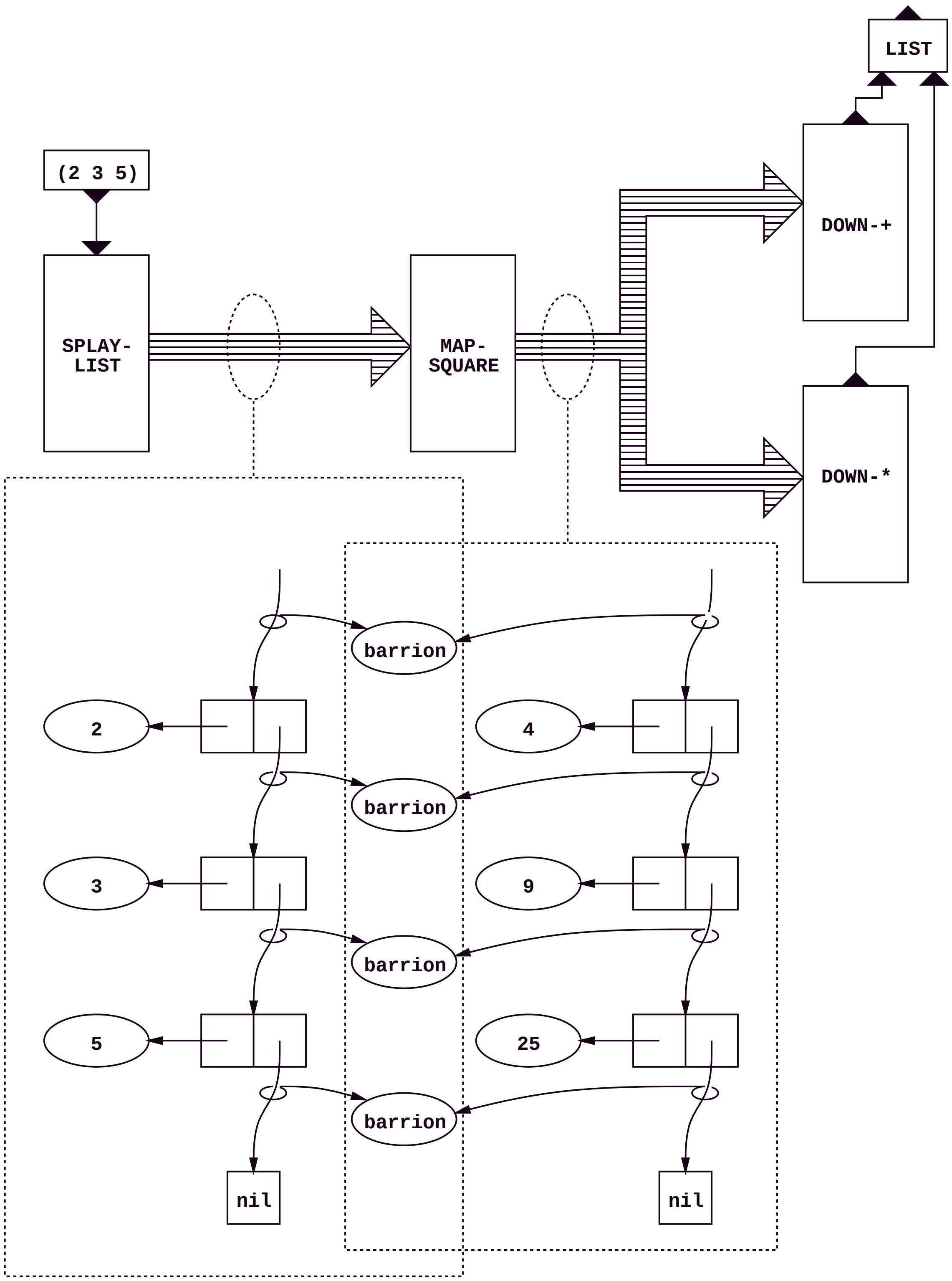


Figure 5.15: Barrion sharing in a simple linear computation. Two intermediate synquences share the same synchrons, permitting all four slivers to synchronize with each other.

guarantees that synchrons generated independently in different parts of the same lock step component will eventually come to denote the same barrier.

- *Wait* is called by a sliver on a synchron when it is ready to participate in a *rendezvous* with all the other slivers in the network. The rendezvous occurs when all the slivers in a lock step component have called *wait* on a shared synchron. It is only possible to rendezvous once at a given synchron. The details of how a rendezvous is determined are rather subtle; they are expanded on in Chapter 7 and formally described in Chapter 8.

- *Precede* declares that the rendezvous at one synchron must occur before the rendezvous at another synchron. It turns out that this operation is necessary to ensure that up synchrons rendezvous in the proper order in the presence of filtering.

5.3.4 Slag Dynamics

The slag diagrams in Figures 5.12, 5.13, and 5.15 are somewhat misleading because they don't accurately portray the time-dependent nature of slags. For example, in Figure 5.15, neither synquence ever exists as a complete entity at any point in time. Instead, a synquence grows downward only as new elements are demanded; meanwhile, it shrinks from above because slivers eagerly drop references to skeletal nodes as soon as they can. In fact, at most one skeletal node of a synquence actually exists at any point in time![9]

Figure 5.16 presents a selected sequence of snapshots that illustrate how the synquences from Figure 5.15 actually unfold over time. The dotted boxes containing question marks represent suspended synquence computations that do not resume until after a rendezvous has taken place on the down synchron of the barrion. The snapshots indicate how the lazy nature of slags leads to a constant storage requirement for this example.

Suppose we modify the two accumulators from the previous example to have up shape rather than down shape. Figure 5.17 shows a snapshot of the modified computation when it first reaches the end of the list. Even though the intermediate synquences have disappeared,

[9] Clarification: By "exists" I mean "is accessible". I assume throughout that inaccessible structures that will be reclaimed by the garbage collector do not count towards the space consumed by a computation.

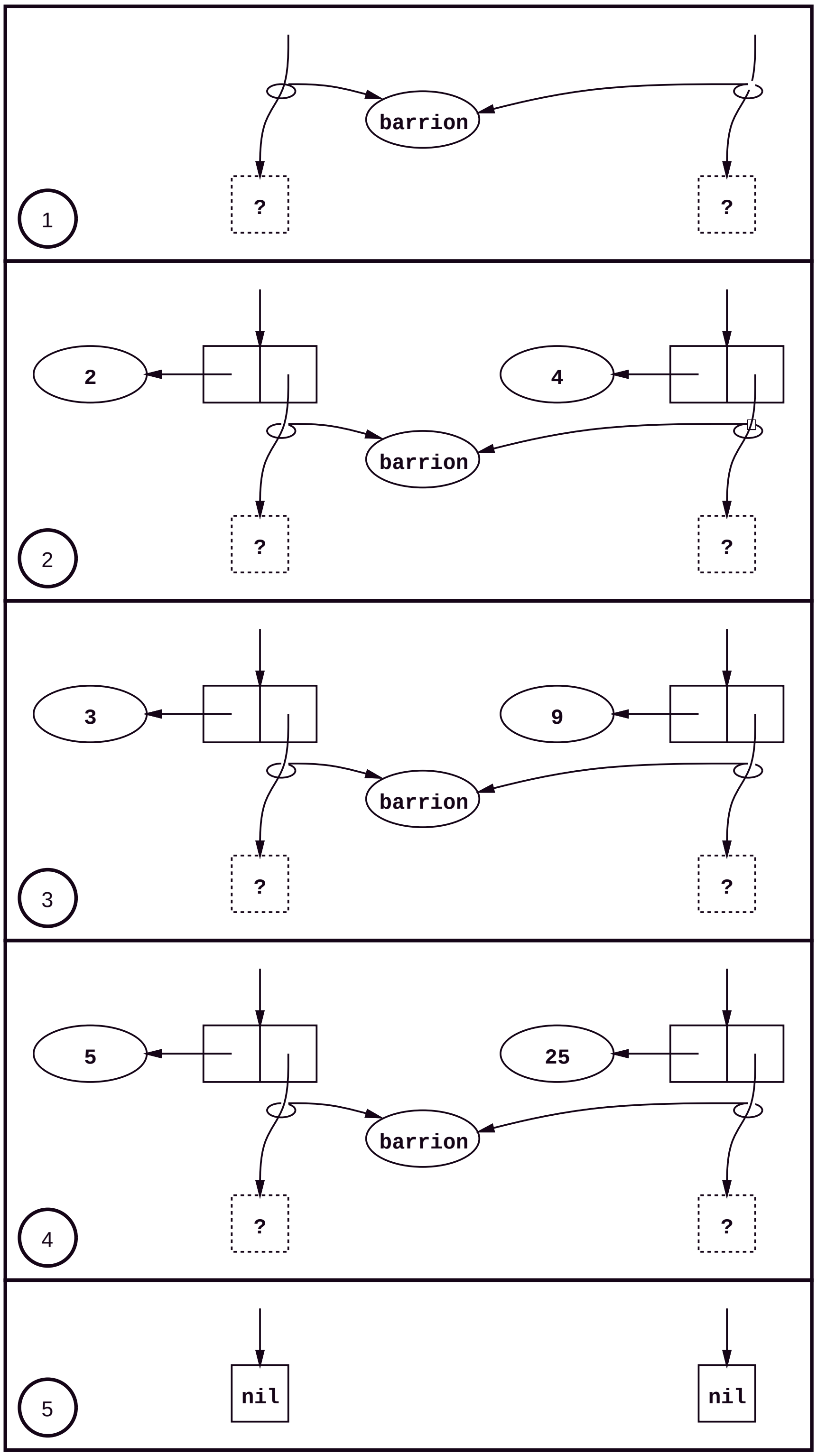


Figure 5.16: A "movie" showing how synquences unfold over time.

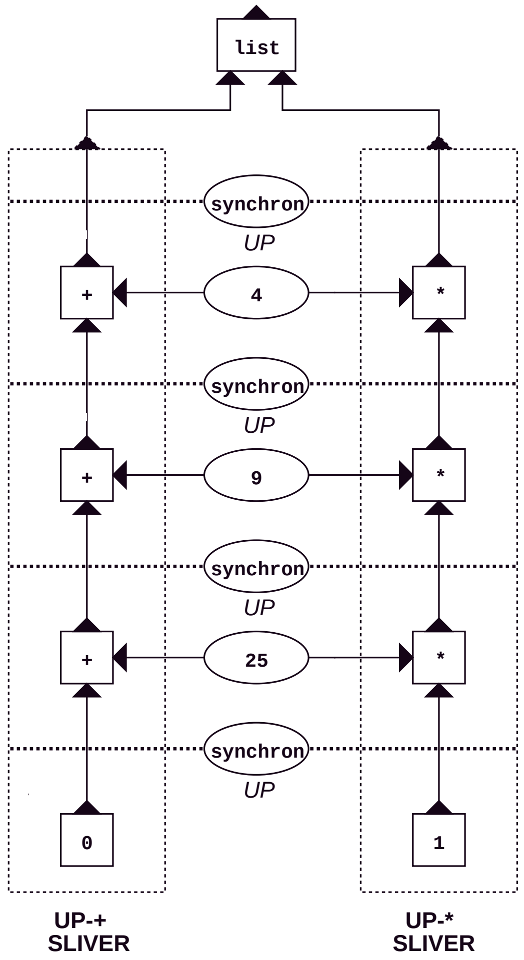


Figure 5.17: A snapshot illustrating how parts of a synquence can be stacked by slivers.

each of the accumulating slivers maintains its own stack of values and up synchrons. This means that the modified computation requires space linear in the length of the list argument. The up synchrons shared by the stack couple the remaining + and * operations so that they will be performed in lock step as the stacks are popped.

Computations involving syndrites are trickier to analyze. In computations that correspond to sequential tiles, a syndrite at any point in time is unfolded only along the branch currently being explored; branches previously explored have disappeared and those not yet explored are suspended. For computations that correspond to parallel tiles, a syndrite may in the worst case be completely unfolded. Although undesirable, this accurately models the space consumption of a monolithic tree-parallel computation. The storage requirements in this case can be greatly reduced by sequentializing the parallel computation.

5.4 Slivers Revisited

5.4.1 Sliver Classification

Slivers are just slag-manipulation procedures that observe some important local structural and behavioral constraints (described later). The slags consumed and produced by a sliver correspond a subtile's consumpts and products; the non-slag inputs and outputs of a sliver correspond to a subtiles arguments and results. To maintain these distinctions, we will say that a sliver *consumes* input slags, *produces* output slags, *takes* arguments, and *returns* results.

Slivers are classified into three categories according to how they manipulate slags:

1. *Generators.* A generator is a sliver that consumes no slags but produces one or more slags. Sample generators include a sliver that produces a synquence of integers between two limits and a sliver that converts a tree into a syndrite. Generators are responsible for creating fresh synchrons for the slags that they produce.

2. *Transducers.* A transducer is a sliver that both consumes and produces slags. A transducer with multiple slag inputs must unify the corresponding synchrons of its inputs and use the unified synchrons in its outputs. Common transducers include:

- *Mappers* that apply a function elementwise to a slag.
- *Filters* that selectively replace slag elements by gaps. (We will see below that, due to technicalities, filters cannot simply be represented as mappers.)
- *Scanners* that produce output slags containing the intermediate values of an accumulation over inputs slags.
- *Truncaters* that produce potentially truncated versions of their input slags.

3. *Reducers.* A reducer is a sliver that consumes input slags but returns results instead of producing output slags. Reducers include *accumulators* that accumulate a value from a slag and *selectors* that select an element from a slag. For instance: a sliver that sums the elements of a synquence, a sliver that collects syndrite elements into a tree, and a sliver that returns the last element of a synquence.

5.4.2 Sliver Requirements

Not every procedure that manipulates slags has what it takes to be a sliver. In order to be a sliver, a slag-manipulation procedure must observe some important local structural and behavioral requirements. Here we describe the requirements and discuss the behavior that that they engender.

A sliver can be viewed as a fragment of a monolithic recursive procedure in which every access to the next slag (via a synter) coincides with a recursive procedure call. We will associate each recursive call in a sliver with the down and up synchrons annotating the current consumpt and product slags.

Every sliver must obey the following requirements:

1. *Synchron propagation.* A sliver must ensure that all of its output slags use the same synchrons (in the same order) as any of its input slags (as in Figure 5.15). This guarantees that all slivers in a lock step component share the same synchrons. If a sliver has no input slags, it generates fresh synchrons that are then shared by its output slags. If a sliver has multiple input slags (which may have been produced by independent generators) it uses the synchron *unify* operation to combine the input synchrons into a single output synchron.

2. *Down synchronization.* As a part of making a local recursive call, a sliver must *wait* for a rendezvous at the down synchron with all the other slivers in the lock step component. This rendezvous simulates the down barrier of a monolithic recursive call. After the rendezvous, the slivers compute independently (modulo data dependencies) until the next rendezvous.

3. *Up synchronization.* If a sliver returns from a local recursive call, then it must *wait* for a rendezvous at the up synchron with all the other slivers that return from their calls. This rendezvous simulates the up barrier of a monolithic recursive call. However, a sliver that makes a tail-recursive call never returns, so it will *not* rendezvous with the other slivers. Thus, a lock step component of linear slivers only exhibits stacking behavior when at least one of its slivers has *up* shape. When all the slivers have *down* shape, the entire lock step component acts as an iterative computation. In this important case, slivers behave like the series procedures in Waters's series package. But slivers also preserve the shape aspects of tree computations as well. For example, a sequential tree computation built only out of slivers with *pre* shape has the tail-recursive characteristics of a pre-order accumulation (see Figure 4.24 on page 4.24).

4. *Lazy elements.* The demand-driven nature of the underlying computational model implies that an element wire does not transmit a value unless it is requested. In other words, while the call boundaries of subtiles are strict, the side-to-side product/consumpt boundaries are non-strict. Operational faithfulness dictates that slivers must delay the computation of every slag element so that the element value is never computed if it is never requested.

5. *Aggressive reference dropping.* In order to preserve expected storage behavior, slivers must aggressively drop references to slags, as well as to the values and synchrons held by slags, when they can no longer be referenced. For example, if a sliver only requires one component of a slag, it must extract that component as soon as possible so that the other components become inaccessible (and therefore garbarge-collectible). Aggressive extraction is required not only for preserving desirable space requirements, but also for avoiding spurious deadlock (see the deadlock discussion in Chapter 7).

6. *Appropriate treatment of gaps.* Due to the presence of filters, a sliver may sometimes discover a *gap* — a token indicating that the element at the given position has been filtered out. The sliver must handle the gap in a reasonable way. The conventions for handling gaps will be discussed in Section 5.5.2.

In practice, it is challenging to write procedures that embody all of the sliver requirements listed above. The sources of difficulty will be discussed in Chapter 7. However, it is possible to express a wide range of common sliver patterns as instances of a handful of carefully written sliver templates. The SYNAPSE language described in Chapter 6 is an example of this approach.

Note that the sliver requirements do not include any rule that corresponds to the unitilable assumption invoked in the discussion of tiles and subtiles. This assumption was made purely to simplify the presentation. As long as slivers obey the above requirements, nothing prevents them from being implemented in terms of nested loops or mutual recursions.

A rendezvous between slivers resembles interprocess synchronization in many models of concurrent processes (e.g., [Hoa85, Mil89, CM90]). However, there are several aspects that distinguish sliver synchronization from these other models.

- With slivers, communication and synchronization are decoupled. Communication is achieved by referencing a data structure, while synchronization is achieved by applying *wait* to a synchron. This approach contrasts with models in which every communication event synchronizes sender and receiver.

- Slivers engage in a multiway rendezvous that involves all the slivers in a lock step component. Most synchronous communication models support only a two-way rendezvous. While CSP [Hoa85] supports a multiway rendezvous, it is limited to communication between a single sender and multiple receivers.

- Since slags carry both down and up synchrons, slivers can naturally express computations (including tree-shaped ones) that can rendezvous upon return from a call return as well as initiaion of the call. This contrasts with the linear iterative nature of many concurrent process models.

5.4.3 Sliver Dynamics

The synchronization requirements obeyed by slivers constrains them to work together in a way that mimics the behavior of a monolithic procedure. As a simple example, consider an iterative list-averaging program expresssed as a sliver network:

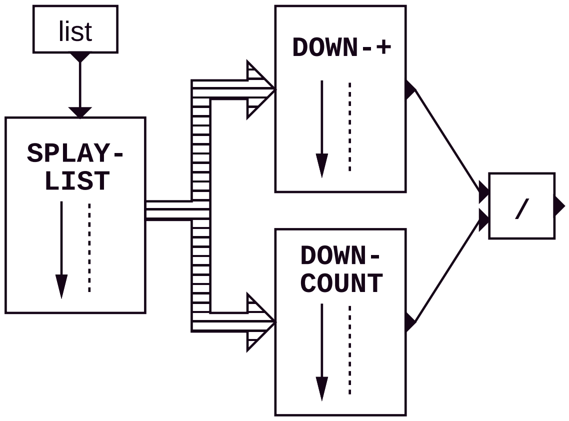


Figure 5.18: Sliver diagram for a program that iteratively computes the average of a list.

`SPLAY-LIST` converts a list to a synquence; `DOWN-+` iteratively sums a synquence of numbers; and `DOWN-COUNT` finds the length of a synquence. Each of the slivers maintains one state variable of the iteration (current list, current sum, and current count). The diagram exhibits fan-out because the synquence output of `SPLAY-LIST` is use as an input to both `DOWN-+` and `DOWN-COUNT`.

A "movie" of selected computation snapshots for the network on the input list `(7 2 6)` appears in Figure 5.19. As indicate by the snapshot movie, `average-list` is indistinguishable in behavior from a constant-space iteration in three state variables. This is remarkable because, as explained in Chapter 3, classical lazy data techniques would build up space linear in the size of the input for this example. In contrast, synquence version is *guaranteed* to use constant space due to the way that slivers make use of the synchronization information supplied by synchronized lazy aggregates.

Indeed, although slivers are composed in the aggregate data style, they behave more like concurrent processes communicating via unbuffered channels. However, as discussed in

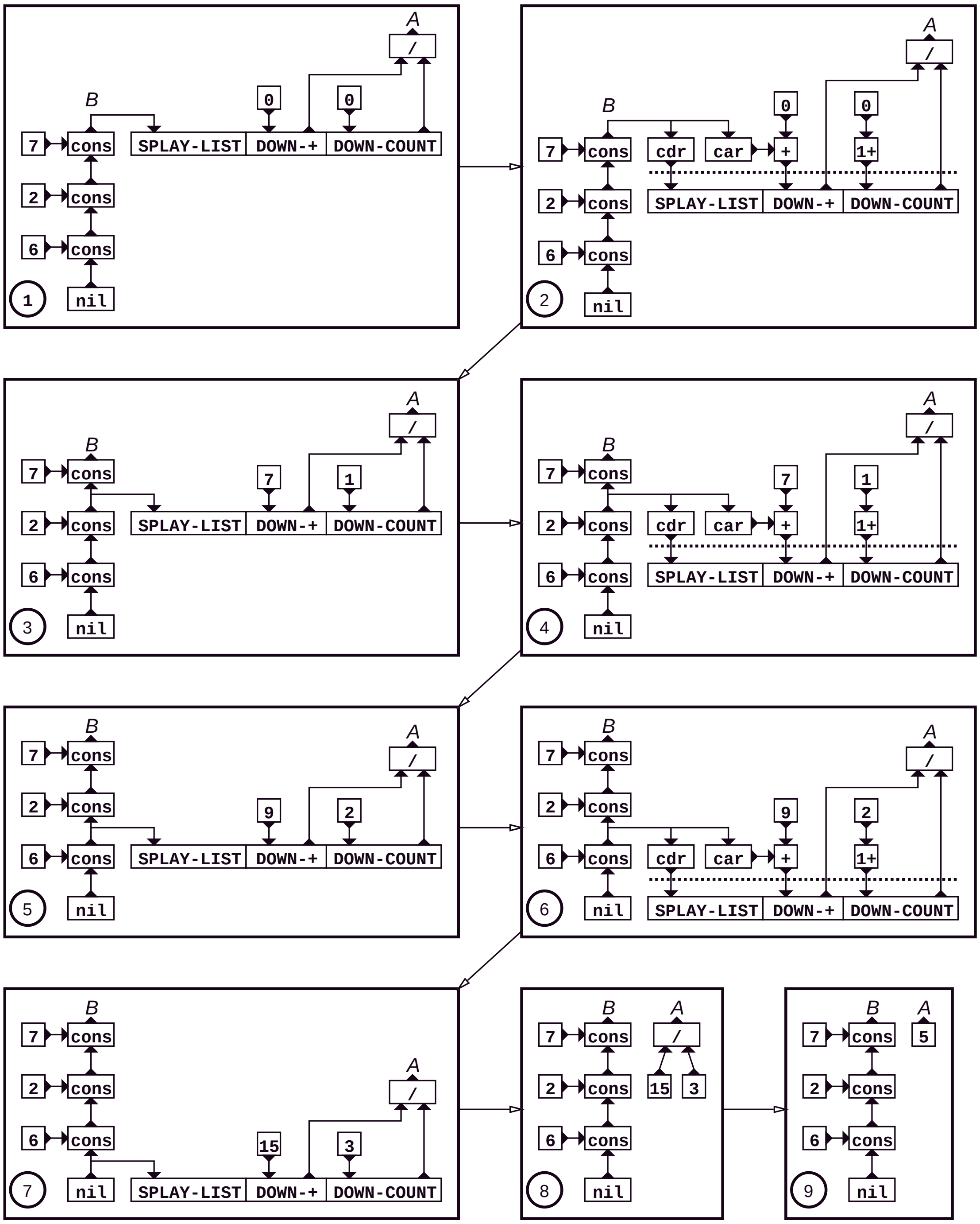


Figure 5.19: A stylized "movie" of snapshots from the sliver-based *down* computation of the average of the list `(7 2 6)`. Assume that A has requested the result. The dotted horizontal lines represent a down synchron shared by the slivers. The list is represented as the result of `cons` operators (see Chapter 8). The internal details and connectivity of the three slivers have been suppressed. Garbage collection of the input list structure is not shown because in general it might be accessible from some other point B in the program.

the previous chapter, the channel-based approach would require some sort of copy process to fan the result of the generator out to the two accumulators. Slags make it is possible to express sliver networks via the same mechanisms used to express other data dependencies in a given language.

We can modify the two *down* accumulators in the list-averaging problem to be *up* accumulators. Then the resulting network behaves like a recursive procedure of one argument (current list) that returns two results: the sum of the numbers in the current list, and the length of the current list. Figure 5.20 shows a snapshot movie for the modified computation. The movie indicates how the computation `cdr`s down the list in *down* phase, and then processes the two results in lock step in its *up* phase.

Of course, we could also modify the list-averaging network so that only one of the *down* accumulators were changed to an *up* one. The resulting computation would have an *up* shape, but would exhibit characteristics of both of the previous examples.

Tree computations and more interesting linear computations have too high a "visual complexity" to illustrate their dynamics in the movie format. We will study such computations through a text-based interface in Chapter 6. I have also developed a dynamic program animator (the DYNAMATOR) that automatically animates computations using a visual representation similar to that in Figures 5.19 and 5.20. The animator is described in Chapter 9.

5.5 Filtering

The model of sliver decomposition presented so far glosses over a few important complications. All of these complications are related to filtering, which is surprisingly difficult to handle in both an operationally faithful and reusable manner. Here we discuss filtering in more detail and extend the sliver decomposition technique to handle the complications that it introduces.

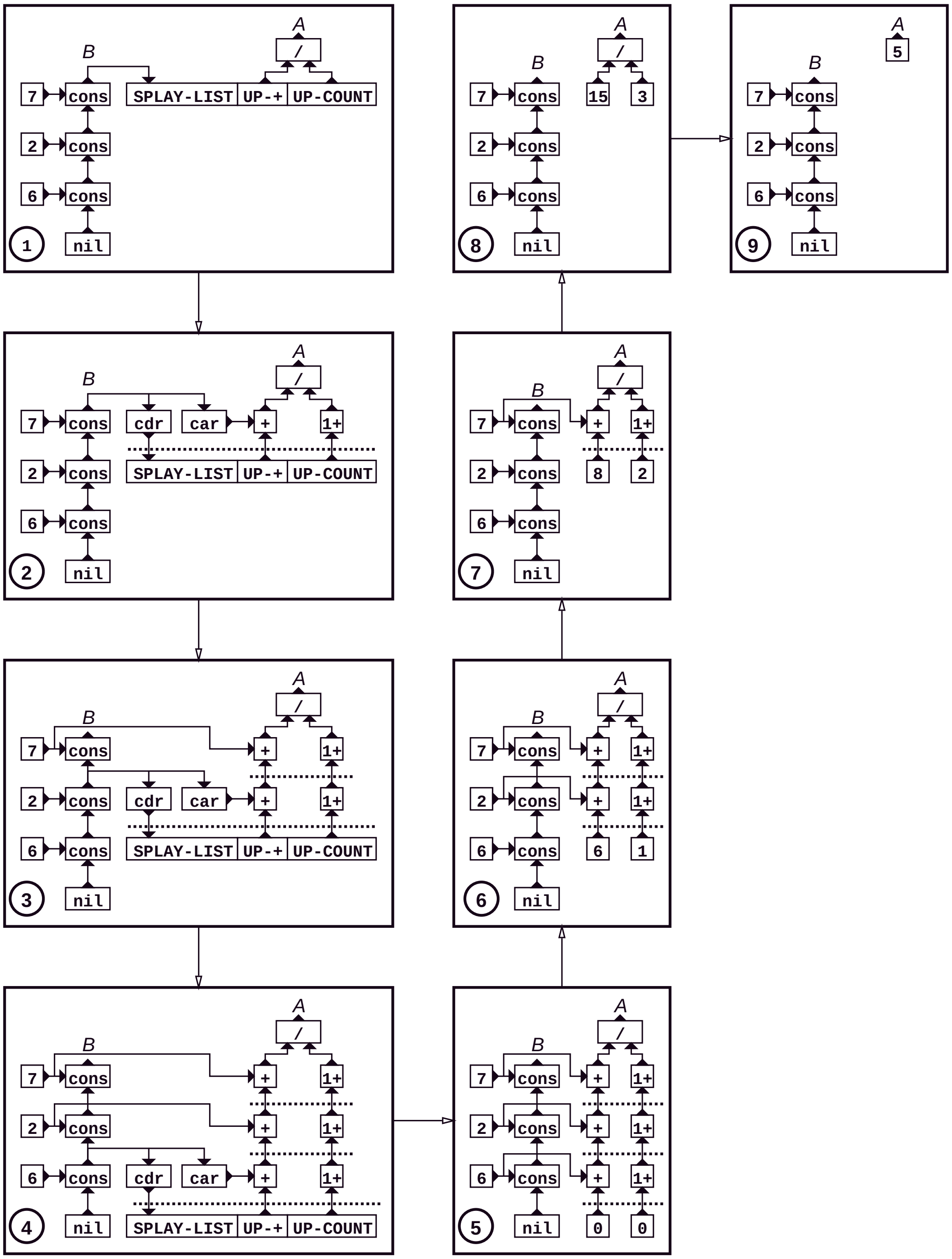


Figure 5.20: A snapshot movie of an *up* computation of the average of the list `(7 2 6)`. The long horizontal dotted lines represent shared down synchrons. The short horizontal dotted lines represent shared up synchrons. Note how the up synchrons force the otherwise independent stacks of pending operations to pop in lock step.

5.5.1 Gaps

A filter is a transducer that *passes* to the output slag only those elements from the input that satisfy a given predicate. Elements that do not satisfy the predicate are said to be *blocked*.

There are two basic approaches to filtering:

1. In the *compaction approach*, the output of a filter contains only the passed elements. For example, a compacting version of a filter that passes odd numbers would transduce the input list `(8 3 2 4 7)` into the shorter output list `(3 7)`. The compacting approach to filtering is commonly employed in aggregate data techniques for linear structures and in channel-based techniques.

2. In the *gap approach*, the output of a filter contains not only the passed elements, but also "holes" indicating where the blocked elements used to be. A gap version of an odd number filter would transduce the input list `(8 3 2 4 7)` into the output list `(#g 3 #g #g 7)`, where `#g` is a distinguished entity that represents the lack of a value. Gap filtering is used in situations where the location of elements is important. For example, languages with array-based or data-parallel features (e.g., [Ive87, Ame89, GJSO92, Sab88, Ble90, RS87, Ble92]) present models in which elements reside at a particular location in an data structure. Compaction filtering is typically not the default in these approaches because (1) elements are often indexed by their location and (2) moving an element between locations may be expensive (e.g., it may require interprocessor communication). Instead, blocked locations either hold a distinguished value, or they are tagged so that they do not participate in calculations.

The compaction approach is straightforward for linear structures, but does not naturally extend to more complex structures like trees and multi-dimensional arrays. Using compaction to filter out an internal node of a tree or two-dimensional array requires the other elements to be rearranged in arbitrary and non-trivial ways. In contrast, the gap approach works for structures with arbitrary topology since the filter always preserves the connectivity of the input.

Sliver decomposition employs the gap approach to filtering primarily because of the lock step nature of sliver processing. A synquence element not only appears at a particular index in the corresponding a sequence (or tree) of values, but it also appears at a location in the synquence that is conceptually bounded by a pair of barrions. Since all slivers in a lock step component share the same barrions, every location of an input synquence to a filter must be preserved in its output. In particular, even the locations of elements blocked by a filter must appear in the output. The location-dependent nature of synquences makes the gap approach the natural choice. In addition, gapped filtering easily generalizes to the tree structure of syndrites.

A SYNAPSE filter inserts a distinguished *gap* value (written `#g`) at every location corresponding to a blocked input element (see Figure 5.21). In terms of the subtile model introduced in Section 5.2.1, a gap corresponds to a triple of wires in which the termination wire and presence wire are both false but the element wire is unconstrained (i.e., its value doesn't matter).

5.5.2 Gap Conventions

Slivers must be carefully designed to handle gaps in an appropriate manner. Since gaps stand in for elements that have been blocked by a filter, a sliver should treat them as if they aren't there. This suggests that the functional arguments of a sliver should not be applied to gap values, and leads to two rules of thumb for handling gaps:

1. *Gap contagion*: Transducers should treat gaps in a contagious manner. Mappers, filters, truncaters, and shifters should map every location with an input gap to an output gap. For transducers that take more than one input slag, an output location should hold a gap if any of the corresponding input locations hold a gap.

2. *Gap apathy*: Linear and sequential tree reducers should ignore all gap positions when accumulating or selecting a result. For example, a length accumulator for the gapped list `(#g 3 #g #g 7)` should return `2`, not `5`. The second element of this list is `7`, not `3`. For reducers that take more than one input slag, all input location should be ignored if any of the corresponding input locations hold a gap.

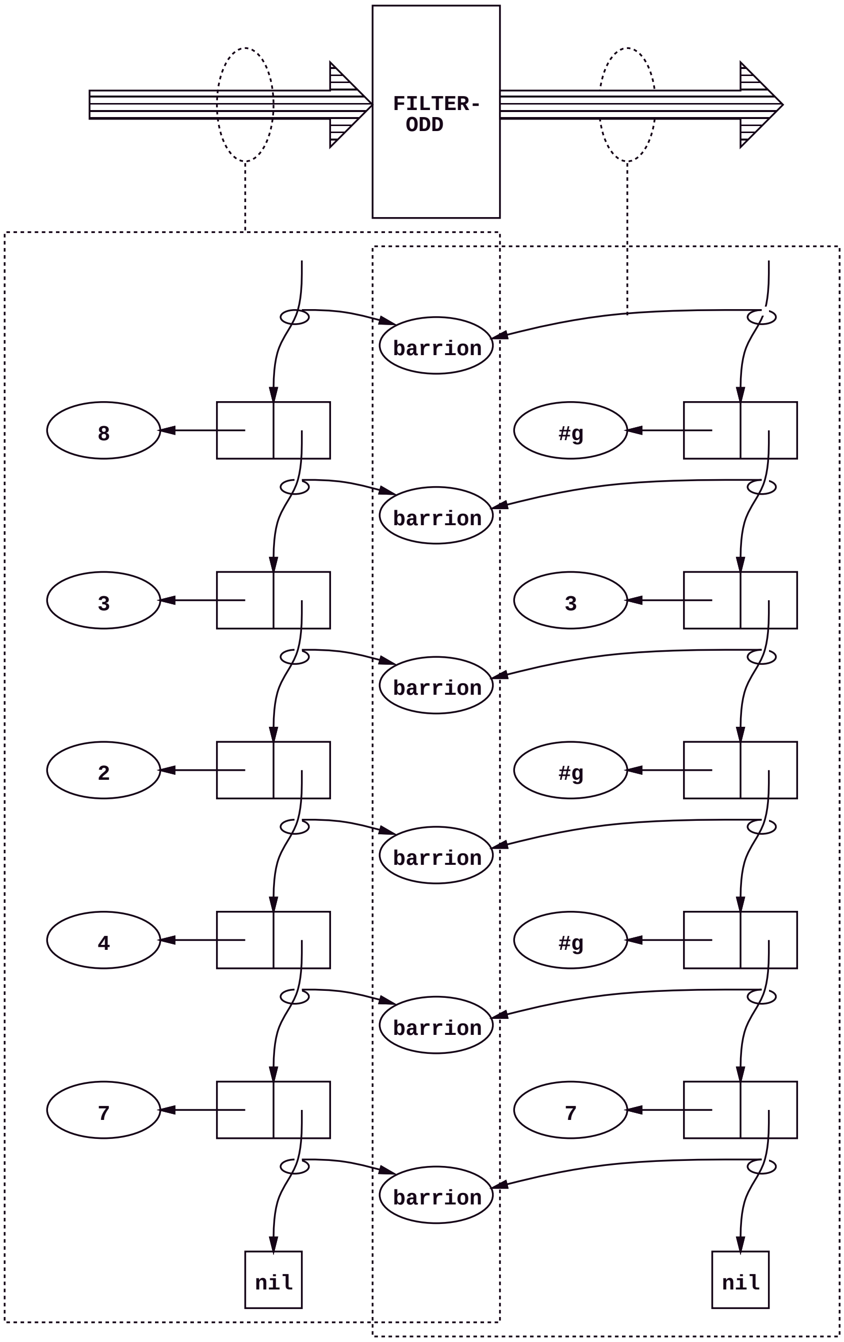


Figure 5.21: Distinguished gap values fill the locations in a synquence that correspond to elements blocked by a filter.

Unfortunately, these heuristics don't handle some important situations:

- Scanners are an important exception to the gap contagion principle for tranducers. Recall that a scanner annotates every node of a slag with the current value of an accumulator when an accumulator processes the node. Since the accumulator has a value even when a gapped node is reached, it is meaningful to annotate gapped nodes as well. In fact, several examples in Chapter 6 require scanners to provide this capability. By default, then, scanners produce slags that contain no gaps. If desired, it is always possible to reintroduce gaps by applying a two-input mapper to the input and output of the scanner (see Figure 5.22).

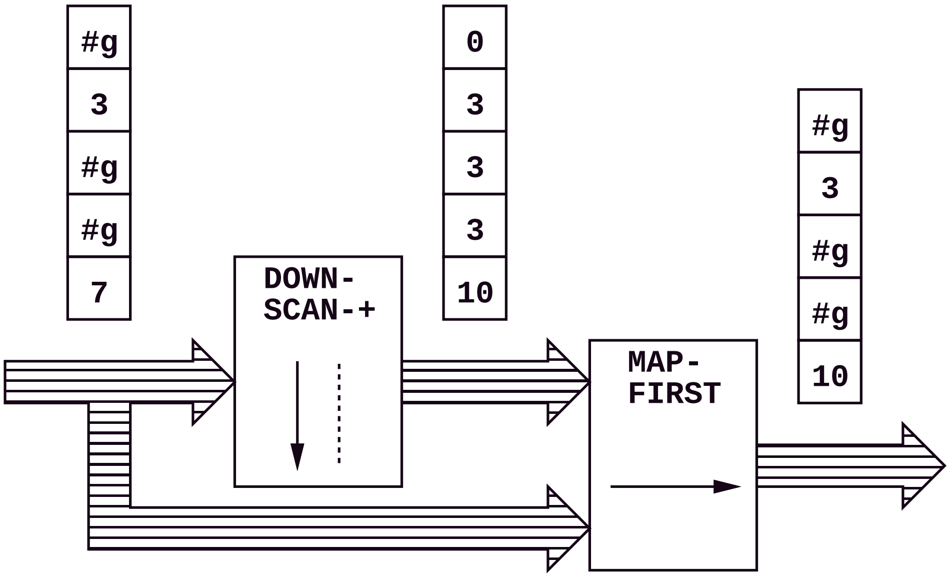


Figure 5.22: Scanners do not obey the gap contagion principle. However, the contagion property of two-input mappers can be used to reintroduce gaps into the result of a scan. Here, `MAP-FIRST` is a mapper that returns the elements from its top input.

- In certain situations it is desirable for the functional arguments to slivers to be able to manipulate gaps with the same status as other elements. For example, consider a numerical slag A that is filtered into three slags B, C, and D that contain, respectively, its positive, zero, and negative elements. Suppose we want to design an accumulator that takes B, C, and D as inputs and performs an operation at each location that depends on which slag has a non-gap element at that location. Since two of the inputs will have a gap at any given location, the gap apathy rule implies that no operation can be performed at any location! This is clearly absurd.

We will employ two methods for addressing this problem:

1. One approach is to provide mechanisms for *reifying* otherwise implicit gaps into explicitly manipulable gaps and *unreify* the explict gaps back into implicit ones. When necessary, higher order sliver functions can be designed to test for reified gaps. For instance, in the positive/negative/zero example, the accumulator could be applied to gap-reified versions of B, C, and D, and it could test for the presence of explicit gaps in order to determined which operations to perform.

2. An alternative is to design classes of slivers that handle gaps in special ways. For example, given a list of slags, a commonly desired processing pattern is to (1) check the slags in turn until one is found that has a (non-gap) element at the current position and (2) perform an action that may depend on which slag provided the element. This pattern could be captured by a specially designed sliver that ignores the gap apathy heuristic. Such a sliver would be able to handle the positive/negative/zero example described above.

The reify/unreify approach is general, but leads to sliver programs that have a brute force character. The special-purpose sliver approach leads to more elegant sliver programs, but is more ad hoc.

- Parallel tree reducers are trickier to handle than their sequential counterparts. Consider the behavior of a *binary up* accumulator at the node of a binary syndrite. Suppose that that the accumulated values from the left and right branches are l and r, respectively. If the node holds a non-gap value v, then the accumulator needs to combine three values: v, l, and r. But if the node holds a gap, there are only two values to combine: l and r. Since the same combiner cannot work for both cases, it is necessary to specify *two* combiners for a parallel tree accumulator.

How to handle selection for a gapped syndrite is less clear. In a gapped synquence, an element is specified by a single index, and gaps can simply be ignored in finding the (ungapped) element at the give index. However, a syndrite element is generally specified by a path that describes what branches to make at each node of the tree. If a node on the path holds a gap, the node cannot readily ignored because it generally will have more than one branch; which should be followed? If we just use the path information to choose a branch at every gapped node, then the path may ultimately terminate at a gapped node. In this case, what is the value of the selector supposed to be? Because of these difficulties, we provide no parallel tree selectors.

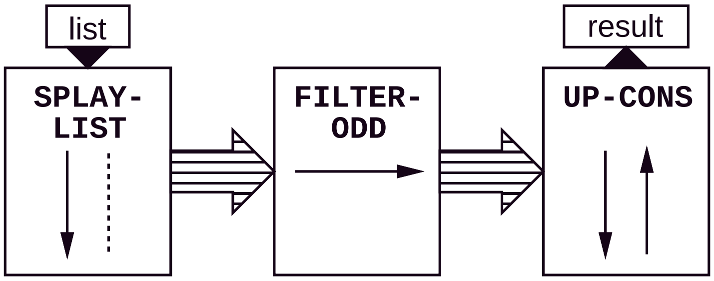
5.5.3 Reusability

Why aren't filters just a particular kind of map? The above discussion on gap conventions provides a partial reason: since gaps are "holes" in a slag, it's desirable for mapping functions never to deal with gaps at all. In particular, they shouldn't take them as arguments or return them as results; this eliminates expressing filter as a kind of map. But this explanation is somewhat unsatisfying because the gap reifying/unreifying mechanism introduced above could be invoked for the special case of filters.

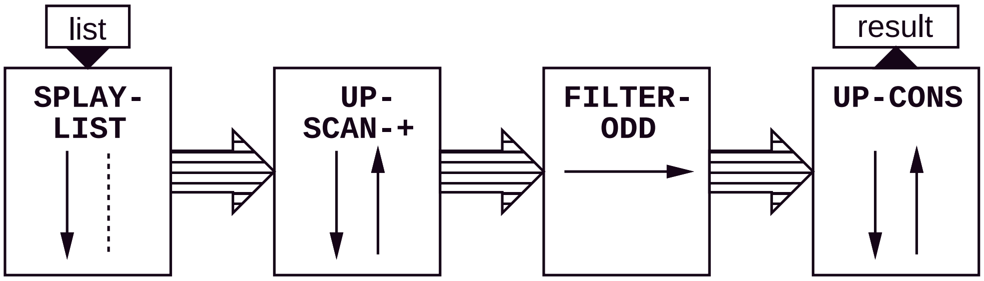
The real reason that filters aren't just maps is that the reusability goal for slivers requires filtering to be a more involved process than suggested by the simple gap model presented above. Using this model, it is impossible to design transducers and reducers that exhibit expected space behavior in every shape-compatible context. We demonstrate this fact via some concrete examples.

Consider the two sliver diagrams shown in Figure 5.23. In both diagrams, an `UP-CONS` sliver consumes the synquence produced by the `FILTER-ODD` sliver. Here is the behavior we expect for the diagrams:

1. The computation described by diagram **(a)** should return a list of the odd elements in the input list. It should test all of the list elements for oddness in the *down* phase. At the completion of the *down* phase, the space required by the computation should be proportional to the number of *odd elements* in the input list. In particular, if



(a)



(b)

Figure 5.23: Two sliver diagrams in which `UP-CONS` follows `FILTER-ODD`. As explained in the text, it is hard to design `FILTER-ODD` and `UP-CONS` so that both diagrams give rise to the expected computations.

the input list contains only even numbers, the *up* phase should require no space for pending computations, since there shouldn't be any.

2. The computation described by diagram **(b)** should return a list of the odd numbers resulting from a upward sum scan of the input list. Since the inputs to the filter are computed in the *up* phase, all of the tests for oddness must be performed in the *up* phase. At the completion of the *down* phase, there must be a pending `if` for every element produced by `up-scan-+`. The computation always requires space linear in the length of the list at this point, even if the input list contains only even numbers.

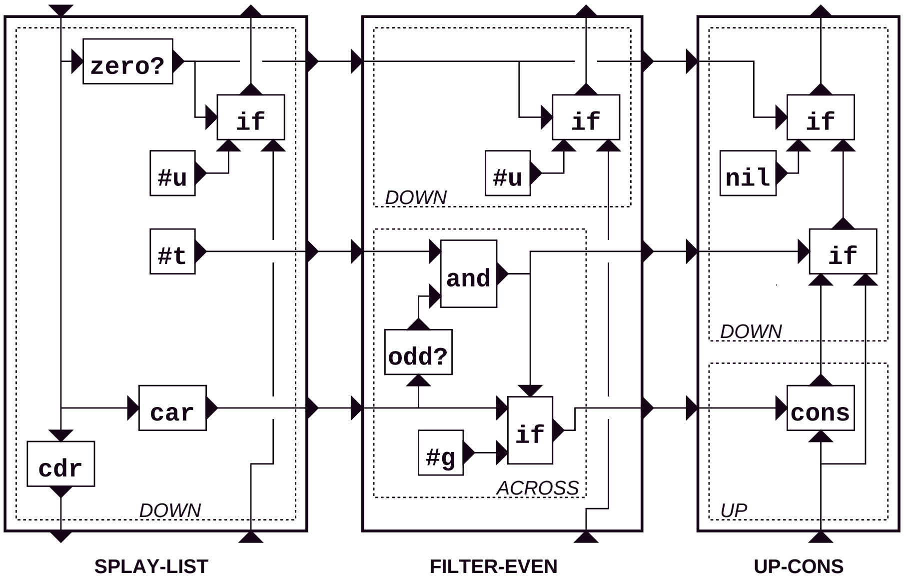
Using the standard sliver implementations developed earlier in this chapter for these two examples leads to the subtile diagrams in Figure 5.24. Unfortunately, the shaded path in subtile diagram **(b)** indicates the presence of deadlock. The problem is that the lower `if` in `UP-CONS` is a *down* operation that must perform its test before the subcall, but the test depends on the value of the result of the subcall of `UP-SCAN-+`.

It is possible to fix the deadlock in example **(b)** by modifying `UP-CONS` to perform the test of the lower `if` *after* the subcall has returned rather than before it. Figure 5.25 depicts the `UP-CONS-TEST-UP` sliver that results from this modification. `UP-CONS-TEST-UP` uses a `seqn` node to force the subcall to precede the test performed by the lower `if`.[10] Replacing `UP-CONS` with `UP-CONS-TEST-UP` in Figure 5.24**(b)** yields a specification for a computation with the desired behavior.

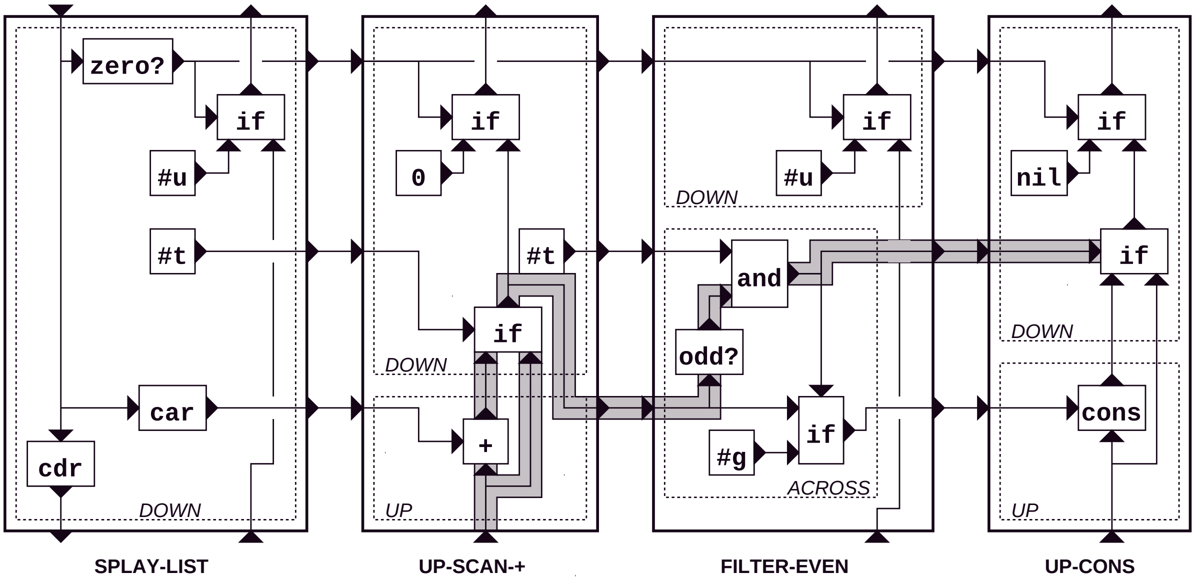
Unfortunately, if `UP-CONS-TEST-UP` replaces `UP-CONS` in subtile diagram **(a)**, the wrong behavior results. `UP-CONS-TEST-UP` guarantees that oddness tests are only performed in the *up* phase of the computation. This means that the resulting computation always requires space linear in the length of the input list at the completion of the *down* phase, even if the list contains only even numbers. This is inconsistent with the expected behavior of example **(a)**.

This example illustrates a tension between operational faithfulness and reusability. Operational faithfulness requires us to have accumulators like `UP-CONS` that preserve the tail-

[10] Recall that a requested `seqn` node first requests the value of its left input wire, and only when that wire has returned a value does it reroute its request to the right input wire. In this example, the request on the left subwire forces the subcall to occur before a request is propagated to the `if`.



(a)



(b)

Figure 5.24: The details of the two filtering problems from Figure 5.23. The shaded path in **(b)** indicates the presence of deadlock.

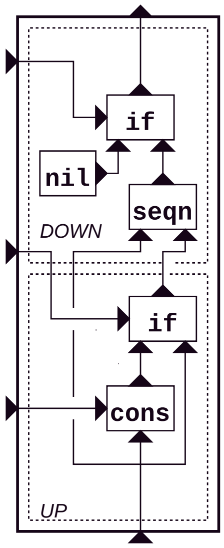


Figure 5.25: `UP-CONS-TEST-UP` results from the addition of a `seqn` operation to the `UP-CONS`. The `seqn` changes the lower `if` from a *down* operation to an *up* operation by forcing the subcall to return before the `if` is tested.

recursive nature of filtering and accumulators like `UP-CONS-TEST-UP` that can handle filters performed in the *up* phase. But reusability suggests that we should have only one *up* accumulator for `cons`, not two. This leaves us with a choice:

1. *The two-up approach*: Accept two versions of every *up* accumulator: one which performs gap tests in the *down* phase, and one of which performs gap tests in the *up* phase.

2. *The one-up approach*: Redesign filtering so that a single *up* accumulator suffices in both situations.

The two-up approach might seem simpler, but it leads to a disaster. The problem is that the prospects for reusability in the presence of filtered *up* scanners are even grimmer than indicated by this particular example. The result of a filtered *up* scan is essentially a different "type" of slag than the one used in all the other subtile decompositions. For every *across* or *up* sliver S that takes an input of the original slag type, the two-up approach requires developing a new sliver S' that takes an input of the new type. The situation is even worse for multi-input slivers: if a sliver takes n slag inputs, it is necessary to have 2^n different versions of the sliver to account for all combinations of argument types. (Usual polymorphism mechanisms are of no help here because each of the potential combinations actually works in a different way.) Finally, even this brute force fails to accurately model the expected behavior of computations like the one pictured in Figure 5.26. The problem in this case is that an ideal `UP-CONS` should only perform gap testing in the *up* phase for those slag locations at which the result of `SPLAY-LIST` had odd elements. Neither `UP-CONS` nor `UP-CONS-TEST-UP` is sophisticated enough to exhibit this behavior.

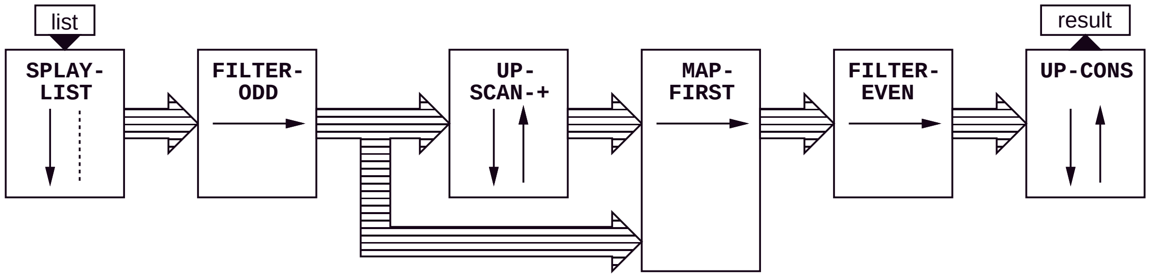


Figure 5.26: Sliver diagram whose expected behavior is difficult to model in the two-up approach.

The numerous problems with the two-up approach suggest that the one-up approach is a better tack. In Section 7.2.4, I describe a design for a one-up approach that works for synquences. The details of a one-up approach for syndrites have not yet been figured out.

5.5.4 Up Synchronization

Preserving operational faithfulness for the *up* phase of a computation in the presence of filtering turns out to be rather challenging. We would like to achieve the following two goals:

1. *Operation order*: The up synchrons of a synquence or syndrite have a natural time ordering from bottom up. Operations should respect this time ordering.

2. *Tail calls*: If all the slivers in a lock step component locally make tail calls at corresponding call boundaries, the entire lock step component should behave as if a

monolithic tail call were made. In particular, no pending operations or stack space should be associated with such a call.

Unfortunately, filtering makes it difficult to satisfy both of these goals. Figure 5.27 illustrates various approaches to a computation with two *up* slivers accumulating `f` and `g`. Due to filtering performed in the *down* phase, there are many locations at which accumulations do not have to be performed.[11]

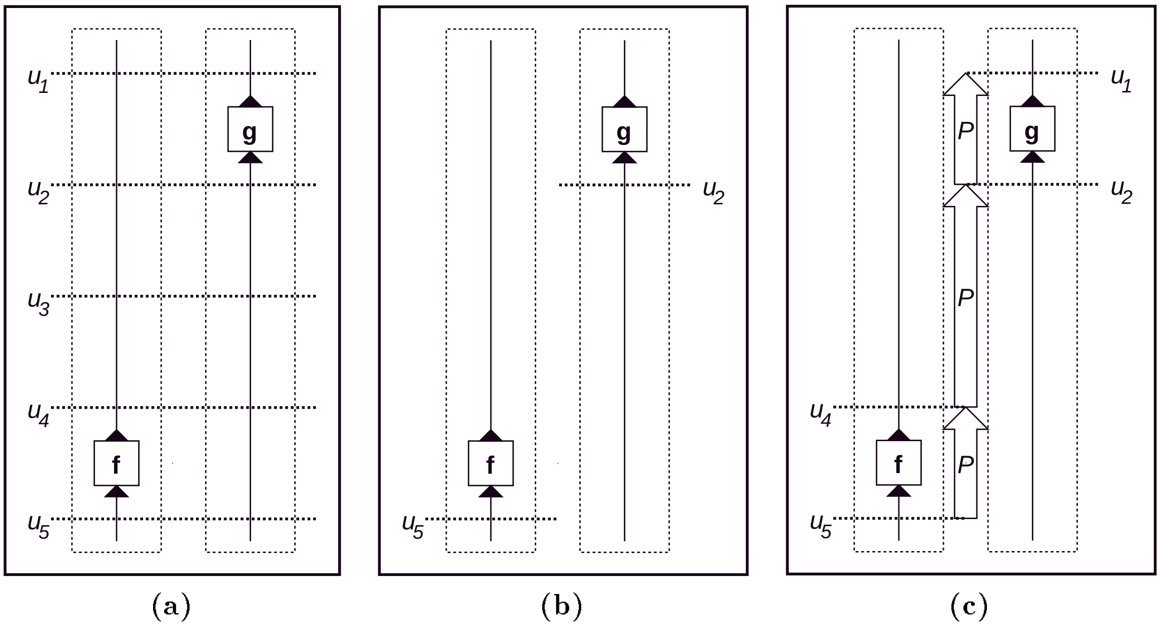


Figure 5.27: Different approaches to handling up synchronization in the presence of filtering.

Figure 5.27(**a**) illustrates a naive approach in which each sliver keeps track of all of the up synchrons of the computation. This makes it easy to satisfy the operation order goal, but tail recursion is lost. No pending operations are performed between up synchrons u_2 and u_3 or between u_3 and u_4, so no space should be allocated for these two locations. However, the up synchrons themselves (and the pending management operations needed to process them) take up space!

Figure 5.27(**b**) illustrates an alternate approach, in which each sliver locally keeps track of only those up synchrons immediately below a pending accumulation. This satisfies the

[11]The configuration depicted in the figure can also arise for reasons other than filtering. However, in practice, the problem is most commonly exhibited due to filtering.

tail recursion goal (no space is needed for u_3 and u_4), but operation order is lost. Since the `f` sliver does not have a handle on u_2, nothing forces `f` to be performed before `g`. The problem is that local tail calls do not keep track of global ordering information.

Figure 5.27(**c**) depicts an approach that satisfies both goals. For each pending accumulation, a sliver locally keeps track of the up synchrons both immediately above and below it. This bounds the time in which the accumulation must take place. In addition, a global precedence ordering (arrows labelled P) is maintained between all the up synchrons that forces the rendezvous on them to occur in the expected order. The synchron *precede* operator is used to specify this ordering. This approach is able to eliminate up synchron u_3 but not u_4. However, the space associated with u_4 can be charged to the accumulation `f` in such a way that the space required by *up* phase is proportional to the number of accumulations that need to be performed.

It is challenging to implement approach (**c**) in such a way that the management of synchron precedence doesn't destroy the desired space behavior. The discussion of the formal semantics of synchrons in Section 8.2.2 explains how this can be achieved.