First-Class Synchronization Barriers

Franklyn Turbak
Wellesley College Computer Science Department
fturbak@wellesley.edu

Abstract

Our purpose is to promote a second-class mechanism — the synchronization barrier — to a first-class value. We introduce the *synchron*, a novel synchronization mechanism that enables the coordination of a dynamically varying set of concurrent threads that share access to a first-class synchronization token. We demonstrate how synchrons can be used to modularly manage resources in cases where existing techniques are either inapplicable or non-modular. In particular, *synchronized lazy aggregates* enable the first space-efficient aggregate data decomposition of a wide range of algorithms. We also introduce *explicit-demand graph reduction*, a new semantic framework that we have developed to describe concurrency and explain the meaning of a synchron rendezvous.

1 Overview

Significant expressive power can be harnessed by capturing programming language idioms in the form of first-class values. A value is said to be *first-class* when it can be (1) named (2) passed as an argument to a procedure (3) returned as a result from a procedure and (4) stored in a data structure. First-classness augments expressive power by allowing previously idiosyncratic and artificially limited mechanisms to be manipulated in general and orthogonal ways. Classic examples of powerful first-class values include first-class procedures and continuations.

In this paper, we argue for the first-class citizenship of a new kind of value: the synchronization barrier (henceforth abbreviated as "barrier"). A barrier is a mechanism for coordinating the execution of concurrent threads. When a thread *waits* at a barrier, its execution is suspended until all threads participating in the barrier are also waiting at the barrier. After this *rendezvous*, execution of the suspended threads is resumed. By constraining the order of computational events among threads, barriers help to manage shared resources. For example, threads atomically depositing money in a shared bank account can be forced to wait at a barrier in order to guarantee that each observes the same final balance after crossing the barrier. Even in a functional language, barriers are helpful for limiting the amount of memory used by a program.

Barriers serve a purpose different from that of mutual exclusion mechanisms like semaphores [Dij68], locks [Bir89], and monitors [Hoa74]. Whereas mutual exclusion mechanisms prevent threads from simultaneously accessing resources, barriers manage resources (e.g., shared variables and memory) by limiting the extent to which threads can be "out of step".

Although mutual exclusion mechanisms have been encapsulated in first-class values like semaphores and locks, barriers are traditionally second-class mechanisms. In most data parallel models, selected processors engage in an implicit rendezvous after the execution of a data parallel operator. The communication protocols in channel-based concurrent process models similarly involve an implicit rendezvous [Hoa85, Mil89, CM90]. Some languages (e.g., Id [AAS95]) permit the programmer to insert explicit barriers, but they are not manipulable as first-class values.

One approach to making barriers first-class is to manipulate them via the following interface:

- `barrier` : *integer* $\rightarrow$ *barrier*. Create a new barrier value for synchronizing *integer* threads. A rendezvous will occur at the barrier when *integer* threads are suspended at the barrier.
- `pause` : *barrier* $\rightarrow$ *unit*. Causes the thread in which the `pause` is called to be suspended at the given barrier until a rendezvous occurs, at which point the thread resumes with the return of the `pause`. It is an error to call `pause` after a rendezvous has occurred.

This sort of first-class barrier captures the essence of most conventional barrier constructs, which are designed so that it is easy to determine exactly how many threads will participate in the barrier. This number is either known statically or can conveniently be determined at run-time when the structure implementing the barrier is created. In these cases, a barrier can be implemented as a pair of a counter and a set of suspended threads. The counter is initialized to the number of participating threads. When a thread encounters a barrier, the counter is decremented and the thread is suspended and added to the set. When the counter reaches zero, the rendezvous occurs, and all the threads in the set are resumed. Note that all threads are treated symmetrically in this synchronization protocol, in contrast with other protocols in which some threads wait for a signal generated by others.

Although straightforward, the `barrier/pause` form of first-class barriers suffers from some important modularity problems. The main limitation of this form of first-class barrier (as well as existing second-class barrier mechanisms) is that it cannot handle situations in which the number of threads participating in the barrier cannot be predicted at the point where the barrier is created. Section 2 presents examples where first-class barriers having a dynamically varying set of participating threads are used in an essential way. Furthermore, the `barrier/pause` style of barrier requires the programmer to keep track of the number of threads participating in the barrier — an error-prone process better handled by the language implementation.

We have invented a novel first-class barrier, the *synchron*, that makes it possible to coordinate a set of threads whose size is not known when the barrier is created. Instead, a rendezvous occurs when the following dynamically determined *rendezvous condition* is met: every thread that *could ever* wait at the barrier *is* waiting at the barrier. All threads with access to the synchron are potential participants in the barrier. A thread participates in the barrier by waiting on the synchron. A thread can leave the set of potential participants by dropping access to the synchron. A thread can also prevent a rendezvous by maintaining access to the synchron without waiting on it. A synchron is a one-shot barrier; after a rendezvous, a synchron "expires" and cannot be used again.

It is helpful to think of synchrons in terms of temporal constraints. A synchron represents the (as-yet undetermined) instant of time at which the rendezvous occurs. Operations can be constrained to happen before or after this instant, or left unordered with respect to it. Similarly, one synchron can be constrained to represent an instant before, after, or simultaneous with that of another synchron. A language supporting synchrons is responsible for solving the temporal constraints; if the constraints are inconsistent, deadlock results.

The interface to synchrons is specified by the following three procedures:

- `synchron` : *unit* $\rightarrow$ *synchron*. Returns a new first-class synchron value. A synchron is a first-class barrier that represents the instant of time at which the barrier rendezvous occurs.
- `wait` : *synchron* $\rightarrow$ *unit*. Causes the thread in which the `wait` is called to suspend until a rendezvous occurs at the given synchron, after which the thread resumes with the return of the `wait`.
- `simul` : *synchron* $\times$ *synchron* $\rightarrow$ *unit*. Constrains both argument synchrons to represent the same instant of time.

All temporal constraints involving operations and synchrons are specified via `wait` and `simul`. Although `simul` declares an explicit temporal constraint, the temporal constraints expressed via `wait` are implicit in the control flow of individual threads.

The design of synchrons was driven by the goal of expressing space-efficient algorithms as the modular composition of aggregate data operators. Following Hughes [Hug84], we argue in Section 2.2 that concurrency and synchronization are essential for preserving the space complexity of non-modular algorithms when decomposing them into reusable components that communicate via aggregate data. In particular, synchrons are the first run-time mechanism to enable the space-efficient decomposition of algorithms that manipulate aggregates non-linearly.

The first-classness of synchrons raises an important question with semantic and pragmatic implications: how is the rendezvous condition computed? Informally, a rendezvous should only take place when all threads holding a synchron are waiting on it. Pointers to a synchron can be classified into two types: *waiting* and *non-waiting*. A suspended call to `wait` holds a waiting pointer to a synchron; all other references are non-waiting. A non-waiting pointer prevents a rendezvous because a thread with non-waiting access to a synchron might later call `wait` on it or share it with other threads. However, when all references to a synchron are waiting references, it is clearly safe for a rendezvous to occur. After the rendezvous, no references to the synchron can remain; this accounts for the one-shot nature of synchrons. Thus, the semantics of synchrons is intimately tied to details of automatic storage management. The semantics presented in Section 5 formalizes this relationship in a high-level way.

The rest of the paper is organized as follows: Section 2 presents examples that illustrate the utility of synchrons. Section 3 surveys related work. Section 4 introduces OPERA, a concurrent version of Scheme [CR+91] that supports synchrons. Section 5 gives a brief overview of EDGAR, a graph-rewriting framework that formalizes the semantics of synchrons. Section 6 concludes with a brief description of our experiences with synchrons.

2 Synchron Examples

2.1 An Event Scheduler

To introduce the power of first-class barriers, we present a simple event scheduler. The interface to the scheduler is defined by the following two Scheme procedures:

- `(event` *thunk*`)` creates an event for performing the action specifed by the parameterless procedure *thunk*.
- `(->` *pre post*`)` declares that the event *pre* must be performed before the event *post*.

The goal of the scheduler is to perform the actions of all events in an order that is consistent with the `->` declarations. A subtlety of the system is that performing the action of an event may introduce new temporal constraints. Consider the following example:

```
(let ((a (event (lambda () (display "A"))))
      (b (event (lambda () (display "B"))))
      (c (event (lambda () (display "C")))))
  (let ((d (event (lambda ()
                    (begin (-> b c)
                           (display "D"))))))
    (-> a d)))
```

The `(-> a d)` forces `A` to be displayed before the declaration `(-> b c)` is encountered. The possible outputs of this expression are `ABCD`, `ABDC`, and `ADBC`. Unsolvable constraints result in deadlock; the following example deadlocks after displaying `A`:

```
(let ((b (event (lambda () (display "B"))))
      (c (event (lambda () (display "C")))))
  (let ((a (event (lambda ()
                    (begin (-> c b)
                           (display "A"))))))
    (begin (-> a b) (-> b c))))
```

Figure 1 shows a complete implementation of the scheduler in a concurrent version of Scheme. An event is represented as a pair of synchrons that specify when the action associated with the event starts and stops. As in other concurrent versions of Scheme, the **future** construct immediately returns a placeholder and commences the concurrent evaluation of the placeholder value [Mil87, Hal85, For91]. Within **event**, **future** forks a thread that forces the thunk application to be sandwiched between the start and stop instants. The **->** procedure forks a thread that guarantees that the stopping instant of the first event must precede the starting instant of the second event. The language implementation is responsible for dynamically solving the temporal constraints introduced by the synchrons.

The event scheduler makes essential use of the first-class nature of synchrons and of the rendezvous condition. If synchrons were not first-class, it would not be possible to bundle them up into an event. Furthemore, since the scheduling constraints cannot in general be determined without executing the program, a barrier mechanism requiring advanced knowledge of the number of participating threads would not be helpful in this situation.

```
(define (event thunk)
  (let ((start (synchron))
        (stop (synchron)))
    (begin
      (future (begin (wait start)
                     (thunk)
                     (wait stop)))
      (cons start stop))))

(define (event-start event) (car event))
(define (event-stop event) (cdr event))

(define (-> pre post)
  (let ((pre-stop (event-stop pre))
        (post-start (event-start post)))
    (future (begin (wait pre-stop)
                   (wait post-start)))))
```

Figure 1: Synchron-based event scheduler.

2.2 Space-Efficient Aggregate Data Operators

A standard modular programming technique is to express monolithic (i.e. non-modular) programs as the composition of mix-and-match operators on aggregate data (lists, streams, arrays, trees). One drawback of this *aggregate data paradigm* is that intermediate data structures can cause modular programs to require more time and space than their monolithic counterparts. Numerous strategies have been developed to reduce these overheads in aggregate data programs (e.g., laziness [Hug90], algebraic transformations [DR76, Bir88], listlessness [Wad84], deforestation [Wad88, GLJ93], series [Wat90]). However, these strategies either unduly restrict the style of aggregate data program allowed (e.g., trees are not allowed; aggregates may only have a single consumer), or they provide no guarantees (e.g., a transformation may increase overhead instead of decreasing it).

The synchron is the first run-time mechanism that enables a broad class of algorithms to be modularized into aggregate data programs that exhibit the same asymptotic time and space complexities as the original algorithms. Time complexity is easy to preserve, but the presence of aggregates can make it difficult to preserve space complexity. For example, list-based decompositions of constant-space algorithms can require linear space, and tree-based decompositions can require space proportional to the number of elements in the tree rather than the height of the tree. Space-efficient aggregate data programs were the primary motivation for inventing synchrons.

We illustrate this technique in the context of a simple function for averaging a sequence a numbers. Given a function *g: integer* $\rightarrow$ *integer*, a predicate *p: integer* $\rightarrow$ *boolean*, and an integer a, consider the sequence $g^0(a)$, $g^1(a)$, $g^2(a)$, ..., $g^{n-1}(a)$, where n is the smallest non-negative integer for which $p(g^n(a))$ is true. Intuitively, a function that averages the numbers in this sequence should run in constant space because it only needs to keep track of three state variables: the current number, the running sum of the numbers, and the length of the sequence. Yet, as we show below, standard approaches for expressing this problem in an aggregate data style require space linear in the length of the sequence. In contrast, a corresponding concurrent program with synchrons is guaranteed to run in constant space.

In the aggregate data style, the averaging function exhibits the structure of the following block diagram:

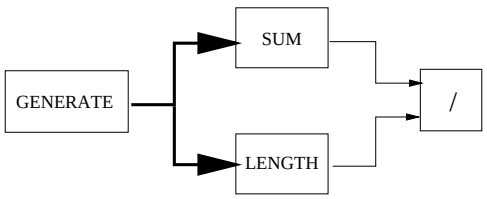

Figure 2: Block diagram for an averaging function.

GENERATE produces a sequence determined by the parameters g, p, and a. This sequence is consumed by blocks that calculate the sum and length of the sequence; these two numbers are divided to give the average.[1]

Figure 3 shows a realization of the block diagram in Scheme. The **GENERATE**, **SUM**, and **LENGTH** blocks are implemented in terms of the higher-order sequence operators **generate** and **accumulate**. **(generate** *seed next done?***)** creates a sequence of values starting at *seed* and iteratively applies a *next* function until the *done?* predicate is true. **(accumulate** *op init seq***)** iteratively accumulates the elements of *seq* with the binary operator *op* starting with *init* as the initial accumulator.

Definitions of **generate** and **accumulate** appear in Figure 4. These in turn use the abstractions **pack** and **unpack**

[1] In the diagram, thick lines designate the transmission of a sequence, while thin lines designate the transmission of a single number.

```
(define (average g p a)
  (let ((nums (generate a g p)))
    (/ (accumulate + 0 nums)
       (accumulate (lambda (x y) (+ 1 y))
                   0
                   nums))))
```

Figure 3: Modular implementation of an averaging function in terms of higher-order sequence operators.

whose purpose is to abstract over different sequence implementations. In the implementation of sequences as strict lists (Figure 5), the **average** function requires space linear in the length of the file because the entire **nums** sequence must be generated before any accumulation operations can be performed.

```
(define (generate init next done?)
  (if (done? init)
      '()
      (pack init
            (generate (next init) next done?))))

(define (accumulate op init seq)
  (if (null? seq)
      init
      (unpack seq
        (lambda (hd tl)
          (accumulate op (op hd init) tl)))))
```

Figure 4: Higher-order sequence procedures.

```
(pack E1 E2) desugars to (cons E1 E2)

(define (unpack lst f)
  (f (car lst) (cdr lst)))
```

Figure 5: Strict Implementation of sequences.

Even if lazy lists are used (Figure 6), the **average** function requires linear space in a sequential language. Laziness does permit producer/consumer coroutining between the generation of **nums** and the accumulation in *one* of the arguments to **/**. But without some form of concurrency, evaluation of one argument to **/** must finish before the evaluation of the other argument can begin. At this juncture, the entire **nums** sequence must be in memory, implying a linear space requirement. Hughes argues in [Hug84] that *any* sequential evaluation strategy for **average** must use linear space.

Concurrency alone does not guarantee efficient space requirements for **average**. Suppose **average** is executed in a concurrent version of Scheme in which all subexpressions of an application expression are evaluated in parallel but the procedure call itself is strict. In the worst case, the sequence implementation of Figure 6 can still require linear space because the evaluation of one argument to **/** may race ahead of the evaluation of the other argument, forcing the entire **nums** sequence to be stored in memory at one time.

```
(pack E1 E2) desugars to (cons E1 (delay E2))

(define (unpack lst f)
  (f (car lst) (force (cdr lst))))
```

Figure 6: Lazy implementation of sequences.

```
(pack E1 E2)
  desugars to (list (synchron) E1 (delay E2))

(define (unpack seq f)
  (let ((sync (first seq))
        (hd (second seq))
        (tl (third seq)))
    (begin (wait sync)
           (f hd (force tl)))))
```

Figure 7: Synchron-based implementation of sequences.

Constant-space behavior for **average** *can* be guaranteed when synchrons are used to augment the lazy implementation of sequences in a concurrent language (Figure 7); we call this technique *synchronized lazy aggregates*. **pack** associates a new synchron with each element of the sequence; **unpack** waits on this synchron before performing any operation on the element. The barrier provided by the synchron prevents the accumulation in one argument to **/** from racing ahead of the accumulation in the other argument. In fact, the synchron forces the two accumulations to proceeed in lock step, so that the computation only uses constant space.

Intuitively, each synchron in a synchronized lazy aggregate models a strict procedure call boundary in the corresponding monolithic version of an algorithm. Figure 8(a) is a stylized depiction of some of the operations performed by a monolithic iterative averaging function. The horizontal dotted lines indicate the boundary of a loop or tail-recursive procedure call in the execution of such a function. In a language with strict procedure calls, all operations above a dotted line must complete before any operations below the line are initiated. This is true whether or not the arguments are evaluated concurrently. A strict procedure call thus acts as a kind of barrier between the evaluation of the arguments and the computation of the body. This barrier is critical for guaranteeing that the averaging computation can be performed in constant space. Indeed, the space consumption problems that arise in the presence of non-strict (e.g., lazy and eager) procedure calls are typically due to the lack of such barriers.

By decomposing a computation into blocks with local procedure calls, the aggregate data style replaces each global barrier by a collection of local barriers (Figure 8(b)). Data transmission provides a loose coupling betweeen the blocks that is generally not good enough to simulate the global barrier. Some additional mechanism is needed to constrain corresponding local barriers to behave like the global barrier of the monolithic computation. Synchrons are such a mechanism. As first-class values, synchrons can be transmitted between blocks. When the same synchron is shared by several blocks, it serves the role of a global barrier (Figure 8(c)). In this way, synchrons constrain the network of blocks to exhibit the same asymptotic space complexity as

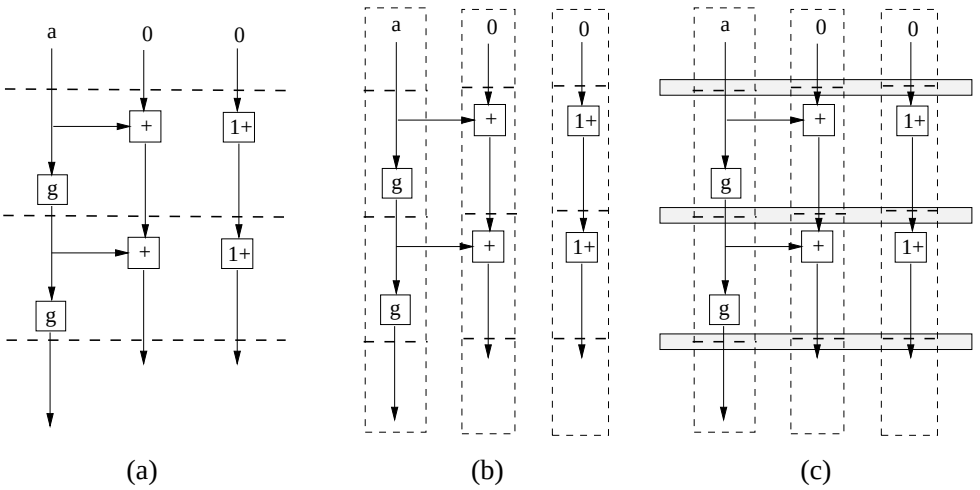

Figure 8: Shared synchrons (c) allow local strict procedure call barriers (b) to simulate the global strict procedure call barriers of monolithic computations (a).

the corresponding monolithic computation.

This approach to the lock step processing of aggregate data operators can be generalized to handle recursively-processed sequences and tree-structured data. For example, alpha-renaming of abstract syntax trees can be expressed as the composition of reusable tree operators. The resulting modular program requires intermediate space proportional to the height of the tree, not the number of nodes. Lock step processing in the presence of recursion and tree-structured data is more complex than the case of linear iteration because it is necessary to synchronize at procedure return as well as procedure call. The modular handling of tail-recursion in the extended system is especially tricky and requires an extension to the synchron interface. Space does not permit a detailed discussion of these issues; see [Tur94] for in-depth coverage.

The `average` example emphasizes that concurrency and first-class barriers are important tools for managing resources (in this case, memory) within a modular program. The feature of `average` that is hard to handle is the fact that the aggregate `nums` is used non-linearly (i.e., more than once). When all aggregates are used linearly, simpler mechanisms, like laziness and deforestation, suffice to make programs space-efficient. These mechanisms effectively match the producer of an element of an aggregate structure with its single consumer. But when there are multiple consumers for a component of an aggregate, it is tricky to direct control between the producer and the consumers to obtain lock step processing. This is why block diagrams exhibiting fan-out of aggregates are difficult to make space-efficient. In general, some form of concurrency and synchronization are necessary to process the producers and multiple consumers in lock step.

Fan-in (blocks with multiple aggregate inputs) is much more straightforward to handle than fan-out. The goal is to unify corresponding barriers from multiple inputs. This is the purpose of the `simul` operator on synchrons. The `simul` operator is not illustrated by the `average` example, but would be required in a program that maps a binary procedure over two sequence arguments. Figure 9 shows this `map2` procedure and the associated `repack2` abstraction. `repack2` uses `simul` to express that the time instants associated with the two input synchrons and single output synchron are all identical.

```
(define (map2 f seq1 seq2)
 (if (or (null? seq1) (null? seq2))
     '()
     (repack2 seq1 seq2 f
       (lambda (rest1 rest2)
         (map f rest1 rest2)))))

(define (repack2 seq1 seq2 f g)
  (let ((sync1 (first seq1))
        (sync2 (first seq2))
        (hd1 (second seq1))
        (hd2 (second seq2))
        (tl1 (third seq1))
        (tl2 (third seq2)))
    (list (simul sync1 sync2)
          (f hd1 hd2)
          (delay (g (force tl1)
                    (force tl2))))))
```

Figure 9: Binary mapping procedure that illustrates `simul`.

3 Related Work

The synchronization mechanism most closely related to the synchron is Hughes's `synch` *E* construct [Hug84]. This is similar to Scheme's `delay` in that it creates a promise for evaluating *E*. The difference is that there are two distinct functions (let's call them `force1` and `force2`) for forcing the computation of *E*. When one of these functions is called on a promise, the thread executing the call is suspended until the other function is called. After both functions have been called, the value of *E* is computed and both threads are resumed with this value as the result of the call.

The value returned by `synch` acts as a limited kind of first-class barrier. Hughes's barrier only permits a rendezvous between exactly two threads; coordinating more threads requires a collection of such barriers. Hughes describes a modular, constant-space averaging function, but his generators and accumulators are not general; they include hardwired assumptions about the particular structure of the averaging problem. In contrast, the `generate` and `accumulate` functions presented above are completely general.

The main drawback of Hughes's approach (as well as of the `barrier/pause` mechanism suggested in Section 1) is that knowledge of the number of threads participating in a barrier must be explicity encoded in a program, obstructing modularity. For example, using Hughes's mechanism, it is not possible to implement `generate` and `accumulate` from Section 2.2 in such a way that they maintain the same interface and still yield lock-step computations. The problem is that `generate` doesn't "know" how many barriers to create, and each call to `accumulate` doesn't "know" which forcing function to call at a barrier. Synchrons are more more modular than Hughes's barrier because the number of threads participating in a barrier is automatically determined by the flow of synchron values through a program, and all threads enter a rendezvous via the same `wait` primitive.

Waters's series system [Wat90, Wat91] is the only other system we know of that can execute aggregate data programs like `average` in constant space. A series is an abstraction of an iteratively processed sequence of values. Programs structured as blocks communicating via series are guaranteed to compile into efficient loops as long as the block diagrams satisfy certain conditions that the programmer can readily verify. (Programs not satisfying these conditions are rejected by the compiler.)

The run-time nature of synchrons makes them more flexible than a compiler-based approach like series. The series compiler requires that the data dependencies between all aggregate data operators be statically determinable. Synchronized lazy aggregates are more general than series because they allow the operators to be configured dynamically. For example, synchronized lazy aggregates can express a constant-space function that takes a list of accumulators and returns the results of each accumulator on the aggregate produced by a single generator. Such a function cannot be expressed via series because the accumulators are not statically known.

Moreover, whereas series is limited to iteratively processed sequences, synchronized lazy aggregates can handle recursively processed sequences and tree-structured data. It would be worthwhile to adapt series compilation techniques to handle these more general kinds of processing; we plan to pursue this line of research.

Deforestation [Wad88, GLJ93] is program transformation technique that removes intermediate data structures from programs. It is more powerful than series in the sense that it can remove tree-structured data. However, unlike series and synchronized lazy aggregates, current deforestation technology cannot deal with fan-out of aggregates. To handle cases like `average`, transformations that combine accumulators have been proposed [GLJ93], but these are ad hoc and it is not clear how to generalize them to general block diagrams.

Barriers are common in parallel processing systems. The operators in many data parallel languages are implicitly sandwiched between global barriers that limit the computation and communication that can occur between successive rendezvous. In other parallel systems, programmers must explicitly insert barriers to ensure synchronization between processors that share memory. Such barriers are second-class mechanisms that coordinate a set of processes known at barrier-creation time.

The Id language supplies a barrier construct for managing program resources and scheduling side-effects in a language based on eager evaluation [Bar92, AAS95]. Barriers are indicated by a dotted line that separates groups of eagerly evaluated expressions; no computation is initiated below the line until all computations initiated above the line have terminated. Because Id barriers can be localized to a particular set of parallel activities, they can express more fine-grained coordination than is possible with the global barriers of data parallel languages. But the inability to manipulate barriers as first-class entities restricts their expressiveness; the examples given in Section 2 cannot be expressed using Id barriers.

On the other hand, synchrons are powerful enough to simluate Id barriers. A barrier can be represented as a single synchron. All threads executing above a barrier can be modified to call `wait` on this synchron as their final action, while all expressions appearing below the barrier can be prefixed with a call to `wait` on the synchron. The first-classness and variable membership of synchrons makes it easy for threads above the barrier to transitively pass along the synchron to all of their descendent threads.

The communication handshake in channel-based concurrent languages (e.g., [Hoa85, Mil89, CM90]) involves a form of barrier. Neither the sending thread nor receiving thread(s) can proceed until a rendezvous of all the participants. In these languages, synchronization is not separable from communication, and communication events are not first-class. Reppy's first-class events [Rep91] really act as first-class *event generators*; every call of his `sync` on a given "event" causes the current thread to wait for a *different* rendezvous. Synchrons can be viewed as a way of permitting the multiway handshake of CSP [Hoa85] to be integrated into the aggregate data paradigm.

A variant of synchrons can be implemented in Bawden's linear language [Baw92], in which objects can only be shared via an explicit copy operation. The copy operation for synchrons can maintain a count of non-waiting pointers; a rendezvous occurs when this number drops to zero. A language implementing synchrons removes the need for explicit copy operations by managing this count automatically.

The synchron is a new member of a class of high-level programming language features whose semantics are inextricably tied to garbage collection. Other examples of such *GC-dependent features* include object finalization and weak pairs [Wil]. The semantics of these features are defined in terms of garbage collection; even if memory were infinite and there were no need for storage reclamation, implementing such features would still require some form of garbage collection.

4 Opera: A Concurrent Scheme with Synchrons

Synchrons are viable in any concurrent language that supports automatic storage managment. However, for the sake of concreteness, we will focus on one such language: Opera, a concurrent dialect of Scheme. (A concurrent version of ML would have been another reasonable choice.) The synchron

examples from Section 2 are written in OPERA. A grammar for the OPERA kernel appears in Figure 10.[2]

Unlike Scheme, OPERA's application has a concurrent semantics (subexpressions of `call` are evaluated in parallel). But, as in Scheme, the application itself is strict (all arguments must be evaluated to values before the application occurs).

P	$\in$	Program
E	$\in$	Expression
I	$\in$	Identifier
L	$\in$	Literal
O	$\in$	Primitive Operator

P	::=	(program E_{body} (define I_{name} E_{def})*)
E	::=	L | I | (primop O)
	|	(lambda (I_{formal}*) E_{body})
	|	(call E_{rator} E_{rand}*) ; keyword optional
	|	(if E_{test} E_{then} E_{else})
	|	(set! I_{name} E_{body})
	|	(delay E_{body}) | (future E_{body})
	|	(exclusive E_{excl} E_{body})
L	::=	*usual Scheme literals*
I	::=	*usual Scheme identifiers*
O	::=	+ | * | cons | car | cdr
	|	*other Scheme primitives*
	|	synchron | wait | simul | synchron?
	|	touch | excludon | excludon?

Figure 10: Grammar for the OPERA kernel.

The default strictness of OPERA application can be circumvented with two classic forms of non-strictness found in other dialects of Lisp [Mil87, Hal85, For91]. `(delay E)` supports lazy evaluation by suspending evaluation of E until its value is required. `(future E)` supports eager evaluation by immediately returning a placeholder for the value of E, which is evaluated concurrently with the rest of the program. `future` and the default parallel argument evaluation strategy are the two sources of concurrency in OPERA. The evaluations associated with the placeholders produced by `delay` and `future` can be explicitly forced by a `touch` primitive, but they are also implicitly forced by *touching contexts* that require the value of the placeholder (e.g., the operator position of `call`, the test position of `if`).

In addition to the implicit synchronization performed by a strict `call`, OPERA supports two explicit forms of synchronization: barrier synchronization and mutual exclusion. Barrier synchronization, in the form of synchrons, is expressed via the `synchron`, `wait`, and `simul` primitives. Mutual exclusion is provided by *excludons*, first-class locks that are used in conjunction with the `exclusive` construct. The form `(exclusive` E_{lock} E_{body}`)` evaluates and returns value of E_{body} while it holds exclusive access to the lock denoted by E_{lock}.

5 Edgar: the Semantics of Opera.

This section sketches our new semantic framework — *Explicit Demand GrAph Reduction* (EDGAR) — which we use to formalize the meaning of OPERA programs, particularly the details of a synchron rendezvous. EDGAR is a graph-rewriting system that is distinguished from other such systems by its explicit representation of the flow of demand through a computation. This feature simplifies the description of OPERA's concurrency, non-strictness, and synchronization. [3] The EDGAR framework was largely motivated by the desire to express the rendezvous condition for synchrons in a simple, high-level way. Because of the close tie between synchrons and automatic storage management, an important feature of the EDGAR semantics for OPERA is that it formalizes garbage collection in a way that programmers can reason about.

5.1 Edgar Overview

The overall structure of the EDGAR framework follows the recipe for an operational semantics [Plo81]: OPERA programs are compiled into an *initial snapshot* (an EDGAR graphical configuration), and *transitions* are made between snapshots in a step-by-step manner according to a collection of *rewrite rules*. A sequence of snapshots encountered in consecutive transitions is called a *trace*. A trace from an initial snapshot to a *final snapshot* (a snapshot from which no transitions are possible) is a *terminating computation* while an infinite trace starting with an initial snapshot is a *non-terminating computation*. Each computation is characterized by a *fate*:

- A non-terminating computation has *bottom* as its fate.
- A terminating computation whose final snapshot is a *value snapshot* has as its fate a *result* whose value is determined by the snapshot.
- A terminating computation whose final snapshot is not a value snapshot has *deadlock* as its fate.

The *behavior* of an initial snapshot is the set of all computations that begin with that snapshot. A behavior often contains numerous computations because transitions may be *non-deterministic* (several transitions are possible from a given snapshot). The *outcome* of an initial snapshot is the set of all fates for the computations in its behavior.

Because OPERA supports side effects (data mutation and I/O), the non-determinism of transitions can lead to an outcome containing multiple fates. The EDGAR semantics for full-featured OPERA is clearly not Church-Rosser, but we suspect Church-Rosser may hold for a functional subset of OPERA.

The key difference between EDGAR and other graphical frameworks [Tur79, Pey87, B$^+$87, AA95] is its use of explicit demand tokens to encode evaluation strategies. EDGAR specifies evaluation order by annotating some graph edges with a demand token that indicates where evaluation steps can take place. The implicit demand propagation implied by traditional inference rules (e.g., "if asked to evaluate a + application, evaluate the left-hand argument") can be encoded in EDGAR as explicit demand propagation steps (e.g. "if the + node is annotated with a demand token, propagate a demand token to the left-hand argument"). In the presence

[2] Non-kernel constructs like `let`, `begin`, and `cond` can be defined as syntactic sugar for kernel constructs.

[3] We have recently learned [Ari96] that the semantics of OPERA can be expressed in a more traditional graph rewriting system using graphical contexts similar to those use in [AF94]. Recasting OPERA semantics in this new form is a future goal.

of explicit demand tokens, a global "reduce any redex" rule suffices because the details of the evaluation strategy are already encoded in the graph itself.

EDGAR's explicit representation of demand was inspired by the demand tokens used in Gelernter and Jagannathan's Ideal Software Machine (ISM) [GJ90], a semantic framework that combines aspects of Petri nets [Pet77] and graph rewriting. A handful of other systems employ explicit representations of demand. Pingali and Arvind describe a mechanism for simulating demand-driven evaluation in a data flow model; they use data tokens to represent demand [PA85, PA86]. Ashcroft describes a system that combines demand flow (via entities called *questons*) with data flow (via entities called *datons*) [Ash86].

5.2 Snapshots

A snapshot is a graph consisting of interconnected labelled *nodes*. Each node can be viewed as a computational device that responds to a demand for a value by computing that value. Every node has a set of labelled *input ports* that specify the arguments to the node and a set of labelled *output ports* that specify the results of the node. The number of input ports and output ports is dictated by the label of the node. Typically, a node has several input ports and one output port.

A connection between nodes is indicated by a directed *edge* from an output port of the *source node* to an *input port* of the target node. Intuitively, an edge is used for a two-step communication protocol between its source and target ports: the target port can demand the value from its source port, and the source port can respond to the demand by returning a value. Every edge has a state attribute that indicates the status of this protocol. There are three possible edge states:

- *inactive*: No demand has yet been made on the edge.
- *demanded*: A demand has been made on the edge, but no value has been returned.
- *returned*: A demand has been made on the edge, and a value has been returned. The *value* returned by an edge in the returned state is defined to be the source node of the edge; typical values include constants, procedures, and data structures.

The protocol further dictates that (1) no value can be returned to an edge until one has been demanded and (2) once an edge is in the returned state, it cannot be used for further communication. An edge, then, acts as a one-shot communication fuse that can be used for transmiting a single demand and a single value before it is "used up". This protocol distinguishes EDGAR from dataflow graph models, in which edges carry a stream of value tokens.

The demanded state is the key feature of EDGAR that distinguishes it from traditional graph rewriting systems and makes it an "explicit demand" model. The returned state is *not* essential; it is just a convenient way to designate the class of value nodes (which could also be specified syntactically).

Figure 11 is a pictorial representation of a sample snapshot in the computation of $(7-3)+\sqrt{(7-3)}$, where the sharing of the `-` node specifies that the difference between 7 and 3 is calculated only once. A unannotated edge is inactive, an edge with an empty circle is demanded, and an edge with a filled circle is returned.

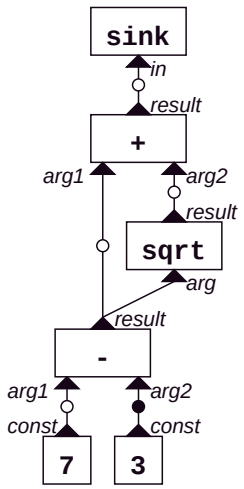

Figure 11: A simple snapshot.

In Figure 11, the `sink` node is a distinguished node that serves as the primitive source of demand in a computation. OPERA programs compile into initial snapshots that are rooted at a `sink` node. (The compilation process is straightforward and is not described here.) A final snapshot is a value snapshot when the input edge to the `sink` node is in the returned state. In this case, the source node of the edge represents the resulting value of the OPERA program.

5.3 Rewrite Rules

Allowable transitions between snapshots are specified by set of *rewrite rules*. The rewrite rules dictate the dynamic behavior of nodes and the flow of demands and values through a sequence of snapshots.

A rewrite rule has two parts: a *pattern* and a *replacement*. The pattern is a partial graph structure in which edges may be attached to *pattern variables* instead of ports. A pattern is said to *match* a snapshot if it can be embedded in the snapshot. The replacement specifies how the snapshot structure matched by the pattern should be replaced in order to construct a new snapshot. For example, Figure 12 shows a simple rewrite rule that propagates demand through a `sqrt` node.

A rewrite rule that matches a snapshot can be *applied* to the snapshot to yield a new snapshot. Removing the structure specified by a pattern from a snapshot that it matches leaves the *context* of the match. The new snapshot is constructed from the context by filling the hole left by the deleted pattern with the structure specified by the replacement. The part of the original snapshot that is not directly matched by the pattern is carried over unchanged into the new snapshot.

Rewrite rules are required to satisfy the following *continuity conditions*:

- All nodes in the pattern must map injectively to similarly labelled nodes in the replacement. The replacement can introduce new nodes, but it cannot delete existing ones.

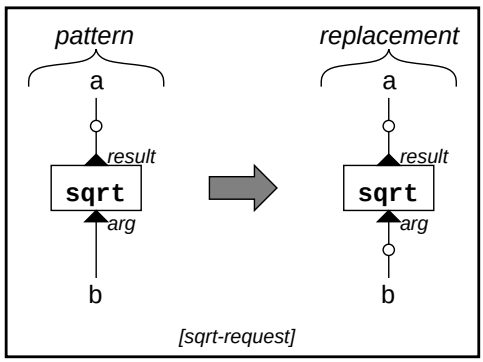

Figure 12: A rule that propagates demand through a `sqrt` node.

- For each edge whose source and destination ports are preserved from pattern to replacement, the pattern state and replacement state must be related by the *communication relation*, which formalizes the communication protocol sketched in Section 5.2. This relation specifies that the edge state may either (1) stay the same (2) change from inactive to demanded or (3) change from demanded to returned.

5.4 Garbage Collection

Rule applications can result in nodes that are inaccessible from `sink` and `future` nodes, which serve as *root nodes* for evaluation. A node is inaccessible from a root node if there is no directed path of edges from the output port of the node to the input port of the root.

In order to accurately model the space required by a computation and to avoid spurious deadlocks involving synchrons, it is necessary reclaim inaccessible nodes from a snapshot. We will assume the existence of a garbage collection function, gc, that maps snapshots to snapshots by removing any nodes that are not accessible from the root nodes.

Since rewrite rules cannot delete nodes, garbage collection is the only mechanism in EDGAR for removing nodes from a snapshot. Why not allow some forms of garbage collection to be specified in the rules themselves? There are two reasons:

1. The continuity conditions from Section 5.3 would be harder to state if all the pattern nodes did not appear in the replacement.

2. Rules performing garbage collection are tricky to write. Even if a node appears inaccessible in the replacement of a rule, it can't necessarily be deleted because it might be accessible from a root via edges that don't appear in the rule.

5.5 Transitions

In order to collect garbage as soon as possible, a transition combines a rule application and garbage collection into a single step. There is a *transition* between S and $gc(S')$ whenever a rule allows snapshot S to be rewritten into S', This aggressive approach to garbage collection guarantees that a synchron rendezvous can't accidentally be blocked by a non-waiting pointer held by an inaccessible node. This is a semantic simplification; a practical implementation of synchrons need not invoke a garbage collector at every evaluation step!

In general, it may be possible to make several different transitions from a given snapshot. In this case, transitions that rewrite different subgraphs of a snapshot loosely correspond to different threads. Because only one rewrite rule can be applied per transition, a single transition allows progress for only one thread.

5.6 Synchron Semantics

Here we briefly discuss the EDGAR rewrite rules that specify the semantics of synchrons (Figure 13). Space does not permit a presentation of all of OPERA's rewrite rules; for more detailed coverage, see [Tur94].

The [*synchron-return*] rule treats `synchron` nodes as self-evaluating values. In English: "If the synchron node has been demanded, then it is returned to the demander as a value."

The [*simultaneous*] rule ensures that all references to two unified synchrons point to the same synchron. The labelled triangles match the set of *all* edges leaving a node as long as *one* of those edges holds a demand token.

The [*rendezvous*] rule formalizes the rendezvous condition for synchrons. A synchron output edge that is in the returned state and attached to a demanded `wait` node is a *waiting pointer*. Any other output edge of a synchron is a *non-waiting pointer*. The rendezvous rule is only applicable when *all* of the output edges of a synchron are waiting pointers. The [*rendezvous*] rule returns a constant true node to all output edges of all the wait nodes participating in the rendezvous. Since the synchron is necessarily inaccessible after the [*rendezvous*] rule, it is garbage collected by the transition based on this rule. The [*rendezvous*] rule embodies a proof of the *rendezvous condition*: every thread that *could ever* wait on the synchron *is* waiting on the synchron.

6 Experience

To experiment with synchrons, we have built a prototype version of OPERA that directly implements the EDGAR rewrite rules. In this system, a node maintains the edges of its input and output ports, and an edge maintains its source and target ports. An application of a rewrite rule mutates the graph structure appropriately. Rather than performing a garbage collection (GC) after every transition, a reference-counting style of GC is supported; when the last output edge is deleted from a node, the node is reclaimed and its input edges are also deleted. Due to the possibility of unreclaimed cyclic structures, it is necessary to perform a reachability-based GC when memory is exhausted. It is also necessary to perform a reachability-based GC when no rewrite rules are applicable; this removes any non-waiting pointers to synchrons that are held by inaccessible cycles and may enable a rendezvous. If no rewrite rules are enabled by this last case of GC, the computation is deadlocked.

In a more traditional system, synchrons could be implemented in terms of object finalization. A synchron can be represented as a mutable cell holding the set of threads

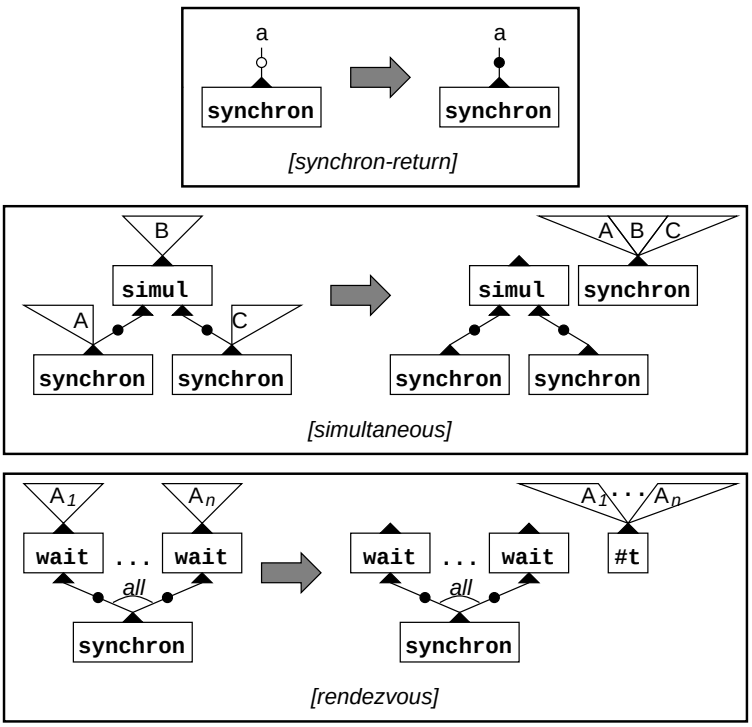

Figure 13: EDGAR rewrite rules for synchrons.

that are waiting at the barrier. Waiting on a synchron suspends the current thread, inserts it into the set of suspended threads, and drops the reference to the synchron. When the synchron becomes inaccessible, a finalization procedure resumes all the suspended threads in the set. `simul` can be handled by unioning the thread sets of two synchrons and sharing the result. It is also necessary to monitor the number of synchrons that hold the same set; the threads in a set can only be resumed when the last such synchron is finalized. Implementing synchrons in terms of object finalization could be prohibitively expensive: programs involving synchronized lazy aggregates make little progress between successive rendezvous, and each such rendezvous requires a full reachability-based GC.

An implementation of synchrons based on reference counts seems more viable. As in the graph-based implementation, a reachability-based GC needs to be triggered when memory is exhausted and when no threads are scheduled. In a statically-typed language supporting synchrons, reference counts would only be necessary for structures from which synchrons could be reached. We plan to experiment with such a language in the future.

Our prototype implementation includes a graphical program animator (the Dynamator) that displays the sequence of snapshots in a computation. The Dynamator has been invaluable for debugging EDGAR rules and OPERA programs. We plan to use it as a pedagogical tool for teaching programming language concepts.

Programming with synchrons is challenging. The main difficulty is that a thread waiting on a synchron may cause deadlock by "accidentally" holding a non-waiting pointer to the synchron. For example, according to the semantics of OPERA, evaluation of the following expression deadlocks:

```
(let ((a (cons (synchron) 17)))
  (begin (wait (car a)) (cdr a)))
```

The variable `a`, which is live after the `wait`, holds a non-waiting pointer to the synchron that prevents a rendezvous.

To avoid such spurious deadlocks, it is helpful to adopt a style of aggressively unbundling data structures that contain synchrons. The following is an alternative to the above example which the OPERA semantics guarantees will *not* deadlock.

```
(let ((a (cons (synchron) 17)))
  (let ((b (car a))
        (c (cdr a)))
    (begin (wait b) c)))
```

This last example underscores the somewhat disturbing fact that OPERA does not respect certain forms of substitution-based reasoning. The rules for when objects become inaccessible must be explicit and simple enough so that a programmer can use them as a basis for reasoning. The OPERA semantics adopts the liveness and tail-call optimizations described in [App92]. These space consumption rules describe the aggressive reclamation of space in a way that the programmer can understand at a relatively high level. They also prevent a correct implementation of OPERA from holding references to values longer than strictly necessary. For example, it is incorrect to evaluate `(begin (wait b) c)` in an environment that maintains a binding between `a` and the pair, because such a binding would lead to a spurious deadlock.

Because they push the envelope of our understanding of GC-dependent features, synchrons may seem unnecessarily complex. But we believe that synchrons merely highlight semantic complexities that are intrinsic to GC-dependent features and even to garbage collection itself. For example, since synchrons can be implemented in terms of object finalization, any complex issues in the semantics of synchrons will also appear in the semantics of object finalization. The main difference between the features is that object finalization is usually considered to be a rare event while a synchron rendezvous is a common event.

Furthermore, we view these complexities not as an indictment of synchrons, but as evidence that new idioms and better implementation techniques are needed to use GC-dependent language mechanisms more effectively. Raw synchrons are powerful but dangerous objects that are not intended for use by casual programmers. But it is possible to package them into abstractions that are accessible to a broader audience. We have used OPERA to implement a suite of sequence and tree operators that can be composed to yield computations that exhibit fine-grained operational characteristics of non-modular loops and recursions [Tur94]. As suggested by Section 2.2, synchrons are crucial for achieving this behavior.

Synchrons and synchronized lazy aggregates are the first steps in a research program whose purpose is to express algorithms in a modular way while preserving important operational properties like asymptotic time and space complexity. Even though concurrency and synchrons seem to be essential for *expressing* certain algorithms in a modular fashion, this does not imply that these features are required for *executing* such algorithms. In fact, we suspect that a compiler similar to Waters's series compiler [Wat91] should be able to automatically generate efficient sequential monolithic programs for many algorithms modularly expressed via synchronized lazy aggregates. Such a compiler would remove all overhead of concurrency and synchronization, as well as the overhead associated with packaging and unpackaging intermediate aggregates. Even when it is impossible to remove synchrons at compile time, synchrons can be replaced by the simpler `barrier/pause` barriers when the number of references to a synchron can be determined statically. It is worth exploring the expressive power of these restricted forms of synchrons because they are considerably less expensive to implement than full-fledged synchrons. Finally, a formal system for characterizing which combinations of synchronized lazy aggregate operators are safe and which lead to deadlock would greatly simplify reasoning about such operators.

Acknowledgments

We thank David Espinosa, Olin Shivers, David Gifford, and the anonymous referees for their helpful comments on drafts of this paper. We also benefited from discussions with Zena Ariola, Andrew Appel, and Simon Peyton Jones.

References

[AA95] Zena Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146:69–108, 1995.

[AAS95] Shail Aditya, Arvind, and Joseph E. Stoy. Semantics of barriers in a non-strict, implicitly-parallel language. Technical Report Computation Structures Group Memo 367, MIT Laboratory for Computer Science, January 1995.

[AF94] Zena Ariola and Matthias Felleisen. The call-by-need lambda calculus. Technical Report CIS-TR-94-23, Department of Computer Science, University of Oregon, October 1994.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ari96] Zena Ariola. Personal communication, January 1996.

[Ash86] E. A. Ashcroft. Dataflow and eduction: Data-driven and demand-driven distributed computing. In J. W. deBakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, pages 1–50. Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 224.

[B+87] H.P. Barendregt et al. Toward an intermediate language based on graph rewriting. In *PARLE: Parallel Architectures and Languages Europe, Volume 2*, pages 159 – 175. Springer-Verlag, 1987. Lecture Notes in Computer Science, Number 259.

[Bar92] Paul S. Barth. Atomic data structures for parallel computing. Technical Report MIT/LCS/TR-532, MIT Laboratory for Computer Science, March 1992.

[Baw92] Alan Bawden. *Linear Graph Reduction: Confronting the Cost of Naming.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1992.

[Bir88] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science (NATO ASI Series, Vol. F55)*, pages 5–42. Springer-Verlag, 1988.

[Bir89] Andrew Birrel. An introduction to programming with threads. SRC Report 35, Digital Equipment Corporation, January 1989.

[CM90] Eric Cooper and J. Gregory Morrisett. Adding threads to standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon University Computer Science Department, December 1990.

[CR+91] William Clinger, Jonathan Rees, et al. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3), 1991.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages (NATO Advanced Study Institute)*, pages 43–112. London: Academic Press, 1968.

[DR76] John Darlington and R.M.Burstall. A system which automatically improves programs. *Acta Informatica*, pages 41–60, 1976.

[For91] Alessandro Forin. Futures. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 219–241. MIT Press, 1991.

[GJ90] David Gelernter and Suresh Jagannathan. *Programming Linguistics.* MIT Press, 1990.

[GLJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming and Computer Architecture*, 1993.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, pages 501–528, October 1985.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Hug84] R. J. M. Hughes. Parallel functional languages use less space. Technical report, Oxford University Programming Research Group, 1984.

[Hug90] R. J. M. Hughes. Why functional programming matters. In David Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

[Mil87] James S. Miller. MultiScheme: A parallel processing system based on MIT Scheme. Technical Report MIT/LCS/TR-402, MIT Laboratory for Computer Science, September 1987.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[PA85] Keshav Pingali and Arvind. Efficient demand-driven evaluation (I). *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[PA86] Keshav Pingali and Arvind. Efficient demand-driven evaluation (II). *ACM Transactions on Programming Languages and Systems*, 8(1):109–139, January 1986.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–250, 1977.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, September 1981.

[Rep91] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation.*, pages 293–305, 1991.

[Tur79] D. A. Turner. A new implementation for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[Tur94] Franklyn Turbak. *Slivers: Computational Modularity via Synchronized Lazy Aggregates.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1994. Accessible from `http://www-swiss.ai.mit.edu/~lyn/slivers.html`. Also to appear as MIT Artificial Intelligence Laboratory AI-TR-1466.

[Wad84] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *ACM Symposium On Lisp and Functional Programming*, pages 45–52, 1984.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *2nd European Symposium on Programming*, pages 344–358, 1988. Lecture Notes in Computer Science, Number 300.

[Wat90] Richard C. Waters. Series. In Guy L. Steele Jr., editor, *Common Lisp: The Language*, pages 923–955. Digital Press, 1990.

[Wat91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

[Wil] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys (to appear).*