Imperative Multi-way Streams

by

John Franco, Daniel P. Friedman and Steven D. Johnson

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 263

Imperative Multi-way Streams

by

John Franco, Daniel P. Friedman, and Steven D. Johnson

Revised: December, 1988

Research reported herein was supported in part by the Air Force Office of Scientific Research, under Grant Number AFOSR 84-0372, and by the National Science Foundation, under grants numbered DCR 85-03279, CCR 87-0211 and MIP 87-07067.

Imperative Multi-way Streams

John Franco, Daniel P. Friedman, Steven Johnson

Department of Computer Science,

Indiana University, Bloomington Indiana 47405

December 14, 1988

Abstract

We present a mechanism for the maintenance of streams based on the Scheme facility of `call-with-current-continuation` or `call/cc`. The mechanism supports stream sharing and has overhead cost which is independent of top-level program parameters if `call/cc` is implemented in heap-based systems. It is shown how the control structure of `call/cc` can save programming effort in cases where multiple procedures output to the same stream in irregular order.

1 Introduction

It is commonly believed that streams are relevant only in a functional language, or, as in the case of Scheme or ML [7], only in a functional style. It is this premise which we shall refute in this paper. In the process we will demonstrate the use of a new programming tool, called imperative streams, for such computations.

The stream-network ideas of Kahn and MacQueen [8] are a basis for imperative streams. We extend those ideas to admit multi-way streams and show how they can be used to develop programs of minimal asymptotic complexity under a certain reasonable Scheme implementation assumption. We impose a restriction which improves stream-network semantics. We provide an implementation in Scheme based on `call-with-current-continuation` which we abbreviate as `call/cc`. Thus, some important escape properties of `call/cc` are inherited by imperative streams.

A stream producing procedure may be regarded as a loop which outputs at least one stream token per iteration when a token is demanded. Implemented with imperative streams, a stream producer can interface with arbitrarily many stream consumers on the same iteration, in any desired order. As a result, computation to produce an output token need not be duplicated. This will be demonstrated later in the paper using the example of n-way stream splitting. Typical functional lazy solutions to this problem require n tests of each input token or padding token but an imperative stream solution requires one test per input token.

Stream producing procedures that are implemented using imperative streams can be escaped from immediately. Thus, procedure unwinding that is part of typical functional solutions is eliminated using the proposed tool.

Imperative streams fit naturally in top-level functional programs. Although assignments are required to maintain them, the streams can be fixed. Thus, the side-effects of stream maintenance can be prevented from showing up at the top-level.

A property of imperative streams is that certain programming structures become simpler versus conventional functional stream manipulation. In particular, programs in which multiple communicating processes contribute output to the same stream in irregular order benefit. In this case, control information must be processed in order to provide proper maintenance of conventional streams. Since this control information is not part of the functional specification at hand, it results in awkward additions to code and detracts from the purity of functional programs. However, with this tool, all stream-producing programs look essentially the same as their list-output counterparts.

Imperative streams are implemented using the powerful control construct, `call/cc`. By redefining car, cdr, and cons in terms of continuations we can, at the moment an output token is sent to a stream, hold the future of the computation of the stream-producing process. By holding onto the continuation of the stream-consuming process and swapping between the two continuations we get the desired behavior. The idea is general enough to work for multiple, shared streams in a stream network [8].

Since the purpose of this paper is to demonstrate the value of a philosophy of stream computation we have provided a bare-bones implementation that is sufficient for running simple examples. The code only reflects the ideas we wish to present and does not include detailed error checking and other embellishments that would make the proposed facility complete. We have sacrificed some efficiency for readability.

In Section 2 we review enough Scheme essentials to make the paper self-contained. In Section 3 we briefly describe `call/cc`. In Section 4 we consider the problem of stream-filtering and demonstrate the need for supporting stream control in order to develop maximally efficient solutions. In Section 5 we describe the language and operational aspects of imperative streams. In Section 6 we illustrate the use of the control mechanism on three simple problems. However, for these examples, not much is gained by using the proposed mechanism. Therefore, in Section 7 we present solutions to the stream-filtering and topological sort problems to demonstrate that our proposed mechanism allows efficient and clear top-level code for a class of solutions that can be stream-networked. In Section 8 we justify our claims for efficiency.

2 An Overview of Scheme

Scheme is an applicative order, lexically scoped, and properly tail recursive dialect of Lisp [11]. Procedures and continuations are treated as first class objects. A subset of Scheme syntax that is suitable for the purposes of this paper follows:

```
⟨expression⟩ ::=
     ⟨constant⟩
   | ⟨variable⟩
   | '⟨object⟩
   | ⟨application⟩
   | (begin ⟨expression⟩+)
   | (lambda (⟨variable⟩*)
   | (let ([⟨variable⟩ ⟨value⟩]*) ⟨expression⟩+)
   | (let* ([⟨variable⟩ ⟨value⟩]*) ⟨expression⟩+)
   | (letrec ([⟨variable⟩ ⟨value⟩]*) ⟨expression⟩+)
   | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
   | (when ⟨expression⟩ ⟨expression⟩+)
   | (case ⟨tag⟩ [⟨symbol⟩ ⟨expression⟩+]+)
   | (cond [⟨tag⟩ ⟨expression⟩+]+)
   | (and ⟨expression⟩+)
   | (define ⟨variable⟩ ⟨expression⟩)
   | (set! ⟨variable⟩ ⟨expression⟩)
   | (set-car! ⟨expression⟩ ⟨expression⟩)
   | (set-cdr! ⟨expression⟩ ⟨expression⟩)
   | (extend-syntax (⟨macro-identifier⟩) [(⟨macro-identifier⟩ ⟨input-spec⟩) (⟨expression⟩)])

⟨application⟩ ::= (⟨procedure⟩ ⟨expression⟩*)

⟨input-spec⟩ ::= ⟨object-structure⟩*

⟨value⟩, ⟨tag⟩, ⟨procedure⟩ ::= ⟨expression⟩
```

The superscript * (+) denotes zero (one) or more occurrences of the preceding form. Square brackets have the same meaning as parentheses and are used to improve readability. The expression list of an application is not assumed to be evaluated in any particular order. An application applies the procedural value of the first expression to the values of the remaining expressions. `begin` expressions are evaluated in order and the value of the last expression is returned. `lambda`, `let`, `let*`, and `letrec` are implicit `begin`s. `lambda` expressions evaluate to first-class procedural objects that statically bind their variables when invoked. `let` makes lexical bindings, `let*` is a nested `let`, and `letrec` makes recursive lexical bindings. `if` evaluates its second expression if its first has value *true*, otherwise it evaluates its third expression. `when` evaluates its second and following expressions when its first expression is *true*, otherwise its value is unspecified. `case` evaluates the tag expression and returns the value of the first expression whose associated symbol matches the tag. The symbol `else` always matches. `cond` evaluates the expressions associated with the first tag that has value *true* and returns the value of the last expression. The symbol `else`, used as a tag, always has value *true*. `and` returns *false* if one of its expressions evaluates to *false* and returns *true* otherwise. Evaluation is from left to right until a *false* expression is found or until all expressions are evaluated. `define` assigns the value of the expression to the (global) variable. `set!` assigns the value of the expression to the (lexical) variable if possible. `set-car!` and `set-cdr!` are similar to `set!` except that they make assignments to the *car* and *cdr* of the specified variable which is assumed to be a cons cell. If it is not a cons cell, an error results. `extend-syntax` [4,9] is a macro expansion facility which is unusual in that it supports syntactic extensions by means of the symbol "...". That is, (⟨object-structure⟩ ...) expands to a list of as many copies of ⟨object-structure⟩ as stated in the invocation of the macro-identifier.

We use several primitive functions. `pair?` returns *true* if the expression is a cons cell. `zero?` returns *true* if the expression evaluates to the number 0. `null?` returns *true* if the expression is an empty list. `eqv?` returns *true* if the first and second expressions have the same value. `eq?` returns *true* if the first and second expressions are both the same object. `cons`, `car`, and `cdr` are as usual. The Boolean constants *true* and *false* are denoted `#t` and `#f`, respectively.

Most Scheme implementations provide a syntactic preprocessor that examines the first object in each expression. If the object is not a keyword then it is assumed to be an application. If the object is a macro-identifier then the expression is replaced by an appropriately transformed expression.

3 Call-with-current-continuation

We give a brief explanation of `call-with-current-continuation` or `call/cc` here. Some of its uses are described in [5], and [6]. `call/cc` is a sophisticated control construct that is intended for applications in which some action causes the computation of one process to be suspended and of another resumed. A process-computation (or continuation) that is suspended by `call/cc` is permanently lost unless it is somehow saved, usually in a global variable. Thus, `call/cc` can be used to implement escape mechanisms (simply do not save the continuation of the process being escaped from) and coroutine operations (save the continuations of all the coroutines somewhere). We make use of `call/cc` in the following style:

```
(let ([f (lambda (k) ⟨expression⟩*)] ... )
  ...
  (call/cc f) ... )
```

The variable `k`, a continuation, refers to the process (or procedure) in which its associated `call/cc` is invoked, at the point it is invoked. The continuation `k` may be invoked from within the body of the `call/cc` or, by assignment of `k` to a global variable, from another process. The syntax of the invocation of `k` is

```
(k ⟨expression⟩).
```

The value of this invocation is the value of ⟨expression⟩, if it completes, but the value is not returned to the usual place. Instead, it is returned to the invocation of the `call/cc` in which `k` was "grabbed". Then, computation proceeds as though it has just returned from the `call/cc`.

Here is a simple example which demonstrates the escape mechanism of `call/cc`. The product of the numbers in the input list `'(1 2 3 4 0 5 6 7 8)` is returned.

```
(let ([f (lambda (k)
           (letrec ([multiply (lambda (l)
                                (if (zero? (car l))
                                    (k 0)
                                    (* (car l) (multiply (cdr l)))))])
             (multiply '(1 2 3 4 0 5 6 7 8))))])
  (call/cc f))
```

When 0 is encountered, the continuation `k` is invoked and the value 0 is returned to `call/cc`. Then computation stops. Therefore, no multiplications actually take place (because nested procedures never unwind) and the numbers 5,6,7, and 8 are never even considered. In other words, computation escapes from the procedure `multiply` when a 0 is encountered in the input list.

Here is a simple example which demonstrates the ability to simulate coroutines using `call/cc`.

```
(let ([f (lambda (x) x)] [q (lambda (x) x)] [r 0])
  (let ([proc1 (lambda (k) (set! r 1) (set! q k))]
        [proc2 (lambda (k') (set! f k') (q 3) 3)])
    (begin (call/cc proc1) (f 5))
    (set! r (call/cc proc2))
    r))
```

When the first `call/cc` is invoked, `r` is set to 1 and the continuation `k` is saved in the global variable `q`. Then `f` is invoked with the argument 5. Since `f` is the identity procedure, nothing noteworthy happens. Then `(set! r (call/cc proc2))` is attempted. This invokes `proc2` which assigns the continuation `k'` to `f` and invokes `q` which holds continuation `k`. This suspends the evaluation of the expression whose value is to be assigned to `r` and resumes computation at the first `call/cc` assuming it had just returned a value of 3 (the argument of `q` in `(q 3)`). Next, `(f 5)` is evaluated. But, since `f` is now the continuation `k'`, evaluation of the `begin` expression is suspended and computation resumes at `(call/cc proc2)` assuming the result of that invocation of `call/cc` had been 5 (the argument of `(f 5)`). Thus, computation proceeds as though it were originally `(set! r 5)` and the global variable `r` is set to 5 by the code of the example. This may easily be checked.

Stream computations are essentially unbounded loops which are suspended when the next demanded output token has been computed, and are resumed when the next output token is demanded. At the point of suspension, the output token is passed to a consuming procedure. This process is a combination of escape and coroutine functions and can therefore be modeled using `call/cc`. The example below illustrates the metaphor.

```
(define cont '())

(define step-and-swap
  (lambda (value)
    (let ([f (lambda (k)
               (let ([old cont])
                 (set! cont k)
                 (old value)))])
      (call/cc f))))

(define looper
  (lambda ()
    (letrec ([w (lambda (x) (step-and-swap x) (w (add1 x)))])
      (step-and-swap '())
      (w 1))))

(define create
  (lambda ()
    (let ([f (lambda (k) (set! cont k) (looper))])
      (call/cc f))))

(define demand
  (lambda ()
    (step-and-swap '())))
```

The procedure `looper` appears to be an infinite loop which bumps the value of `x` by 1 on every iteration. However, every time through the loop, the procedure `step-and-swap` suspends `looper` and returns the value of `x` to `demand` which soon terminates. Each time `demand` is invoked `looper` is resumed for one iteration and the cycle repeats.

Each procedure must know about the continuation of the other procedure in order for computation to proceed unbroken. The "other" continuation is kept in the variable `cont`. When `looper` is resumed, `cont` holds the continuation of `demand` and when `demand` is resumed `cont` holds the continuation of `looper`.

The continuation exchanging using `cont` is accomplished by `step-and-swap`. `step-and-swap` exchanges the continuation held in `cont` with its own continuation. Then it invokes the previously held continuation with the value of its argument. The result is that `step-and-swap` (and therefore the procedure in which it resides) is suspended and the procedure associated with the previous continuation is resumed. Eventually, a consumer invokes the continuation of `step-and-swap` (from `cont`) with some value V and computation proceeds as though `step-and-swap` has just returned with the value V. That is, the process containing `step-and-swap` is awakened.

The only question remaining is how did `cont` get its first continuation? This is the job of `create` which puts its own continuation in `cont` and starts `looper`. Immediately, `looper` swaps its own continuation for that of `create` thereby priming `cont`.

Here is a short transcript which demonstrates the interaction of these programs.

```
>(create)
()
>(demand)
1
>(demand)
2
>(demand)
3
...
```

4 The Need For Better Stream Manipulation

Consider the problem of splitting a stream of tokens into n disjoint streams based on some given switch function which maps stream tokens to stream names. There are two obvious ways this problem might be attacked functionally. One could write separate filter procedures for each output stream or one could write a procedure which assigns each input token to an output stream until demand for tokens is satisfied.

Let us simplify our discussion by supposing the input stream is an arbitrary stream of integers and the output streams are the even and odd integers of the input stream. Typical code for the first solution looks like this:

```
(define filter-odds
  (lambda (int-stream)
    (if (even? (car int-stream))
        (filter-odds (cdr int-stream))
        (cons (car int-stream) (filter-odds (cdr int-stream))))))

(define filter-evens
  (lambda (int-stream)
    (if (odd? (car int-stream))
        (filter-evens (cdr int-stream))
        (cons (car int-stream) (filter-evens (cdr int-stream))))))
```

where `cons`, `cdr`, and `car` are lazy. Unfortunately, each integer is tested twice: once by filter-odds and once by filter-evens. In the general case the switch function would be applied to each input token n times. This could be serious if the test were expensive.

The second method takes care of this problem by synchronizing all output streams using special padding symbols called *hiatons* [13]. Typical code looks like this:

```
(define filter-odd-even
  (lambda (int-stream)
    (let ([v (car int-stream)])
      (if (even? v)
          (cons (cons v '#) (filter-odd-even (cdr int-stream)))
          (cons (cons '# v) (filter-odd-even (cdr int-stream)))))))


(define filter-odds
  (lambda (pair-stream)
    (let ([v (cdr (car pair-stream))])
      (if (eqv? v '#)
          (filter-odds (cdr pair-stream))
          (cons v (filter-odds (cdr pair-stream)))))))


(define filter-evens ... )

(define solution
  (lambda (int-stream)
    (let ([result (filter-odd-even int-stream)])
      (cons (filter-odds result) (filter-evens result)))))
```

where # symbols are hiatons. This solution trades switch tests of int-stream tokens for tests of hiatons and space to store them. Since hiaton tests are trivial the total cost of testing can be close to optimal. However, the increase in space required for the hiatons could be prohibitive.

A syncronizer solution without hiatons is possible. For example,

```
(define filter
  (lambda (str)
    (if (even? (first str))
        (let ([new-stream (pair (first str) (first (filter (rest str))))])
          (pair new-stream (rest (filter (rest str)))))
        (let ([new-stream (pair (first str) (rest (filter (rest str))))])
          (pair (first (filter (rest str))) new-stream)))))
```

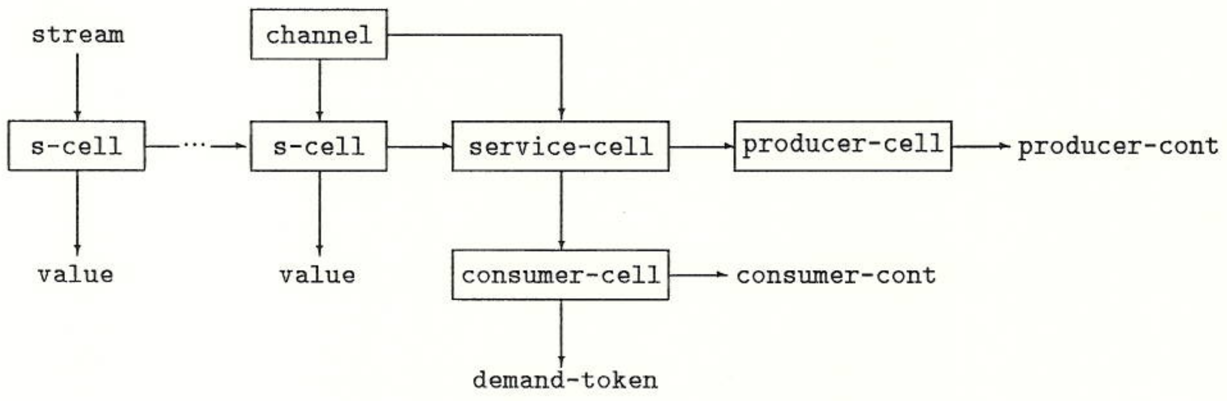
where `first` and `rest` are call-by-need variants of `car` and `cdr` and `pair` is a fully lazy cons. However, the cost of this solution is the same due to destructuring.

The solutions presented in this section are not satisfactory because their computational cost depends on the input stream, modulo the switch function, or on the number of ways the input stream should be partitioned. Clearly, there is a solution which is independent of these factors. We will show that imperative streams provide general solutions to similar problems which are functional at the user level and have a computational cost which is at worst a constant times optimal. We will also show how imperative streams can simplify solutions where multiple processes feed the same output stream. We save discussion of this until we present our solution to the problem of topologically sorting a partial order in Section 7.

5 Imperative Streams

The language of imperative streams is based on the following simple architecture: special stream-producing procedures, or *producers*, connect to simple stream-consuming statements, or *consumers*, within procedures (possibly other producers). The connection, or *channel*, is two-way: consumers demand tokens of the producers and producers supply the needed tokens to consumers. Channels are created as needed, are "opened" in a producer, and are "connected" in a consumer before tokens and demand can be exchanged. If the producer has no more tokens to generate over a channel, that channel is closed but not terminated. Channels are terminated outside the scope of their creation. The lexical scoping attribute of Scheme makes special network configuration directives, as in [8], unnecessary.

The code which implements imperative streams and a brief description is given in Appendix A. The code is based on the following data structure.



Boxed entities are cons cells, unboxed entities pointed at by arrows are values, and `stream` is a pointer. Arrows from the right side of a boxed entity represent cdr fields and downward arrows represent car fields. Values that are continuations have a `cont` suffix. `s-cells` represent a list of computed stream values; they may be referenced by one or more `stream` pointers. The last `s-cell` references a `service-cell`. A producer continuation, a consumer continuation, and a `demand-token` are reachable from the `service-cell` and all these entities combined are a *channel*. A channel is referenced by a `channel` cell. The same `channel` cell references the *tail* of the list of `s-cells`.

A consumer walks through the list of `s-cells` taking values until a channel is reached; access is by means of a `stream` pointer. Then it sets the `demand-token` and runs the producer continuation. The producer eventually notices and satisfies the demand and resets the `demand-token`; access is by means of a `channel` cell. A new `service-cell` is created to replace the existing one and the old `service-cell` becomes an `s-cell`. The `channel` pointers are advanced accordingly. The value referenced by the new `s-cell` is the value returned by the producer. None of the locations of existing cells are changed in the process, although the value of the producer continuation is changed to reflect a more advanced computation. This allows streams to share the same channel and different channels to be connected to the same producer. This multi-way property appears to have been omitted from [8] although it has not been explicitly ruled out. None of the examples of [8] are multi-way.

Because a consumer accesses `s-cells` from a `stream` pointer and a producer accesses a channel, all consumer operations have `streams` as arguments and all producer operations have `channels` as arguments. It is the responsibility of the user to correctly associate a stream with its channel.

A channel must be created before it is used. This allows producers generating tokens for two or more channels to be defined prior to the streams they connect. The procedure which does this is `create-channel`. The purpose of this procedure is analogous to the purpose of `(define cont '())` in the example of Section 3.

A channel is made active by two procedures which are applied at opposite ends of the intended connection. The procedure `open!` is placed at the beginning of a producer. It takes a `channel` as argument and returns a copy of the associated stream which may be used locally. The procedure `connect!` is placed at the consumer end of the connection. It takes two arguments: a channel and a procedure call that produces a stream. It returns another copy of the stream which may be used by the consumer. The purpose of `connect!` and `open!` is analogous to the purpose of `create` and the line `(step-and-swap '())`, respectively, in Section 3. After a channel becomes active, control is passed to a consumer. This results in full laziness.

The procedure `share-connect!` allows several channels to share the same producer connection. Actually, it is the producer continuation that is shared. This allows computation within the producer to always resume from the last point of suspension regardless of which consumer is demanding a token.

There are two producer operations: `put-to-channel!` and `close!`. `put-to-channel!` takes two arguments, a value and a channel. The result is to append the value to the list represented by the stream associated with the specified channel. If the `demand-token` associated with the channel is set then control is returned to the consumer. Otherwise the producer keeps going. This action eliminates the need for hiatons mentioned earlier. `close!` works like `put-to-channel!` except that a null is placed in the cdr of the list and the connection is terminated. However, the channel remains in existence within its scope and can be used to open another stream. The purpose of `put-to-channel!` is analogous to the purpose of `(step-and-swap x)` of Section 3.

There are three consumer operations: `car-s`, `cdr-s`, and `null-s?`. `car-s` and `null-s?` are capable of generating demand for the next stream token and work as though their stream arguments are lists. `cdr-s` is a normal cdr in the presence of two or more `s-cells` and otherwise returns the `service-cell` which may be used to generate the next stream value. All consumer operations work the same way if their arguments are lists so "closed" streams may also be handled. `car-s` and `cdr-s` generate error messages if they attempt to access elements that do not exist. There is no direct analog to these consumer operations in Section 3 but the procedure **demand** has a similar purpose.

A notable feature of this implementation is that a stream receives demand from and supplies tokens to only one consumer. Thus, there is no confusion as to the order tokens are generated on a channel regardless of the number of streams sharing the same channel. In [8], care must be taken to prevent tokens from getting lost or directed to the wrong channel after reconfiguring.

6 Simple Examples

The following program computes a stream of Fibonacci numbers.

```
(define fibonacci
  (lambda (channel)
    (let ([fib (open! channel)])
      (letrec ([f (lambda (s1 s2)
                    (put-to-channel! (+ (car-s s1) (car-s s2)) channel)
                    (f (cdr-s s1) (cdr-s s2)))])
        (put-to-channel! 1 channel)
        (put-to-channel! 1 channel)
        (f fib (cdr-s fib))))))

(define fib-stream
  (let ([f-channel (create-channel)])
    (connect! f-channel (lambda () (fibonacci f-channel)))))
```

Consider the following test procedure which drives `fibonacci` and produces the n^{th} Fibonacci number.

```
(define test-fib
  (lambda (n)
    (letrec ([next
               (lambda (x s)
                 (cond [(= x n) (car-s s)]
                       [else (next (add1 x) (cdr-s s))]))])
      (let ([channel (create-channel)])
        (next 1 (connect! channel (lambda () (fibonacci channel))))))))
```

This example shows the basic relationship between producers and consumers. `test-fib` terminates without the channel to `fibonacci` ever being closed. If `test-fib` is run a second time, a new "copy" of `fibonacci` is created and the same result as the first run will be obtained (provided the same argument is used) because a new channel is created. However, if the channel were defined globally, the second run would continue from where the first run left off and the result would be different.

Early references to the stream of Fibonacci numbers are lost. In fact, there are only three references active at any time. Hence, `test-fib` does not consume much space.

Here is a solution to the famous Tower of Hanoi puzzle where `n`, the number of disks, is set to four and the output is a list of pairs which specify a peg from which to take a disk and a peg to move the disk to.

```
(define hanoi
  (lambda (channel n)
    (open! channel)
    (letrec ([han (lambda (x y z m)
                    (cond [(= m 1) (put-to-channel! (cons x y) channel)]
                          [else
                           (han x z y (sub1 m))
                           (put-to-channel! (cons x y) channel)
                           (han z y x (sub1 m))]))])
      (han 'A 'B 'C n))
    (close! channel)))
```

A stream may be constructed as follows:

```
(define hanoi-stream
  (let ([h-channel (create-channel)])
    (connect! h-channel (lambda () (hanoi h-channel 4)))))
```

This may be called, for example, by

```
   (car-s (cdr-s ...  (cdr-s hanoi-stream) ...  )).
```

It is possible to create a "copy" of `hanoi` by means of another `connect!` statement using another `channel`. Then the original stream and the copy function independently. If a stream is created by means of another `connect!` statement and (the same) `h-channel`, the new stream is the *tail* of the original stream at the time the connection is made.

Finally, we present a procedure for merging two (possibly finite) streams `stream-1` and `stream-2` of increasing integers. `(merge-s ch stream-1 stream-2)` produces a stream of increasing integers containing all the integers in both streams.

```
(define merge-s
  (lambda (ch stream-1 stream-2)
    (open! ch)
    (letrec ([merg (lambda (s1 s2)
                     (cond
                       [(null-s? s1) s2]
                       [(null-s? s2) s1]
                       [(< (car-s s2) (car-s s1))
                        (put-to-channel! (car-s s2) ch)
                        (merg s1 (cdr-s s2))]
                       [else
                        (put-to-channel! (car-s s1) ch)
                        (merg (cdr-s s1) s2)]))])
      (merg stream-1 stream-2))))
```

As a result of the `null-s?` tests, `merge-s` terminates with a value that is a stream. Such termination is equivalent to a loop of `put-to-channel!` statements, one for each integer in the stream, followed by a `close!`. However, the implementation given in Appendix A operates in a more efficient manner: the output stream is simply tacked onto the stream being consumed at the other end of `ch`.

All of the solutions given in this section can be written just as easily and with approximately the same asymptotic computational cost using conventional streams. In the next section we show two examples where imperative streams have a clear advantage.

7 Multiprocesses and Streams

In this section we present another solution to the stream splitting problem. The solution requires minimal space, has optimal complexity under the conditions of the next section, and yet is functional at the user level. This example illustrates the use of imperative streams in applications involving networks of communicating processes.

Here is the code for splitting an integer stream into its even and odd integers. The output is a cons cell with the even stream in the car and the odd stream in the cdr.

```
(define even-odd
  (lambda (input-stream)
    (let*
      ([next-token
         (lambda (e-ch o-ch)
           (open! e-ch)
           (letrec
             ([switch
                (lambda (token)
                  (if (even? token) e-ch o-ch))]
              [split-stream
                (lambda (s)
                  (cond [(null-s? s) (close! e-ch) (close! o-ch)]
                        [else (let ([token (car-s s)])
                                (put-to-channel! token (switch token))
                                (split-stream (cdr-s s)))]))])
             (split-stream input-stream)))]
       [e-chnl (create-channel)]
       [o-chnl (create-channel)]
       [e-stream (connect! e-chnl (lambda () (next-token e-chnl o-chnl)))]
       [o-stream (share-connect! o-chnl e-chnl)])
      (cons e-stream o-stream))))

(define filter-odd-even (even-odd str))
```

where `str` is an input stream created using `connect!` and `open!`. This solution may be generalized to an $n-$way filter. The general solution requires no hiatons and there is exactly one `switch` test per `input-stream` token. Thus, modulo the expense of `call/cc`, this solution represents an improvment over the functional solutions presented earlier. We will discuss the expense of `call/cc` in the next section.

Using `call/cc` as the underlying control mechanism for streams has another advantage: it allows early escapes from procedures when streams are closed. Consider the problem of topologically sorting (lazily) a partial ordering. The following is a macro expansion solution to this problem where `[n (m ...)]` means that all elements of the list `(m ...)` must directly precede element `n` in the partial order.

```
(extend-syntax (topo)
  [(topo ([n (m ...)] ...))
   (lambda ()
     (open! topo-chnl)
     (letrec ([n (lambda ()
                   (set! n (err-fun 'n topo-chnl))
                   (m) ...
                   (put-to-channel! 'n topo-chnl)
                   (set! n (lambda () #t)))]
                   ...)
       (n) ... (close! topo-chnl)))])

(define err-fun
  (lambda (x chnl)
    (lambda () (writeln 'cycle-found-at x) (close! chnl))))
```

In this example we have chosen to make the channel global to demonstrate that global channels are permissable and to make the following sample invocation a bit more readable.

```
(define tot
  (let ([topo-chnl (create-channel)])
    (connect!
      topo-chnl
      (topo ([a (b c e)] [b (d f)] [c (b)] [d ()] [e (b f)] [f (d)])))))
```

This instance has the topological ordering `d,f,b,e,c,a`. The expansion of `topo` with this input is given in Appendix B.

The code for `topo` outputs lazily to the stream associated with `topo-chnl`. One procedure is constructed for each element in the input lists. Each of these procedures feeds the stream with the associated element when all of its predecessors have been output to the stream by their associated procedures (in the line `(m) ...`). No extra control information is apparent in `topo` because control is handled beneath the surface by `call/cc`.

If the input is not a partial ordering, `err-fun` will be invoked at some point. This procedure displays an error message and closes the output stream. Doing so returns control to the top-level procedure which drives the stream and all vestiges of the remainder of the computation within `topo` are erased. Thus, the unwinding of nested calls within `topo` to report the error is eliminated.

8 Execution Cost

We have made several prominent claims for efficiency in this paper. In this section we justify these claims.

First, when we talk of efficient programs we are talking about programs with execution times that increase with input size no faster than a constant times the complexity of an optimal program.

Second, our efficiency claims depend on the assumption that `call/cc` is not implemented as a control stack copy operation [2,3] but rather as a cons cell that points to a portion of the heap-based control stack. We realize that no production version of Scheme currently does this. Implementing `call/cc` by representing the control states in the heap has constant cost, but procedure calls are slightly more expensive (by a constant factor) and this is why `call/cc` implementations are the way they are. However, heap-based systems have certain advantages over stack-based systems: for example, implementation of multiple threads, generation scavenging [12], and improved garbage collection in the presence of large memories [1]. We regard `call/cc` as another argument in favor of heap-based systems and expect appreciation of it to rise as it becomes more understood. Hence, we expect to see a shift to heap-based systems eventually.

Third, there is little space cost since there is only one active continuation per channel and producer.

Fourth, we have made use of the macro expansion facility known as `extend-syntax`. It is possible to implement this facility in such a way that our efficiency claims are justified. But, this is not a major point since efficient and conventional code for topological sorting can be produced with imperative streams. The only reason we have used `extend-syntax` here is that it resulted in concise code for the problem of topologically sorting a partial order.

Fifth, we take into account the fact that `call/cc` enables early escapes which can result in significant execution cost savings in certain instances.

We have performed some rudimentary experiments to determine the penalty of using `call/cc` on an existing Scheme system. We compared the computation cost of `test-fib` to the cost of the program below which simulates `test-fib` without using `call/cc`. `test-fib` is `call/cc` intensive.

```
(define dot
  (lambda (n)
    (let ([s1 '(1)] [s2 '(1)])
      (letrec
        ([next
           (lambda (x a b)
             (cond [(= x n) (car b)]
                   [else (next (add1 x) b (list (+ (car a) (car b))))]))])
        (next 2 s1 s2)))))
```

The experiments were performed on a VAX 8800 using Chez Scheme. We observed only a factor of 2 slowdown using `test-fib` to compute the 10000^{th} Fibonacci number. We note, however, that the program was run on an empty system stack.

We also compared `even-odd` to the two functional solutions to stream splitting of Section 4. In one case one level of nesting of `call/cc` was generated and in the other case two levels of nesting were generated. For a random stream of 10000 integers between 0 and 99 the one level solution was slower than maintaining hiatons by a factor of two and four times slower than double testing.

However, the two levels solution was six times slower than hiaton testing and twelve times slower than double testing. The extra overhead is due to greater copying from the stack.

Finally, we compared the generalized n-way version of `even-odd` to the n-testing functional solution. We used one level of nesting of `call/cc` and we subtracted out the time to generate the random numbers. For a random stream of 10000 integers between 0 and 25 with 26-way splitting, the imperative streams solution was over four times faster than the purely functional solution.

These simple experiments show the potential of imperative streams. However, they appear to be viable only on heap based systems unless the fraction of `call/cc` invocations is low.

9 Conclusions

We have presented a mechanism for maintaining streams in an imperative world. The mechanism is based on the Scheme facility `call/cc`. This mechanism can provide reduced programming effort in an environment where multiple streams interact. In such an environment streams are produced by special procedures called producers. Producers are usually regarded as infinite loops that become suspended when they produce the next stream token and are resumed when there is demand for the next stream token. However, producers can also generate finite streams. The important features of imperative streams are

- A producer may send tokens to any number of consumers regardless of their location.
- Early escapes from procedures are automatic.
- A producer may be "copied" and reused any number of times, usually with different parameter settings.
- There is little space penalty.
- The cost of n-way stream splitting can be held to a relatively slowly growing function of n.
- Imperative streams, when fully developed, can be used to build stream networks.

Extensions to the mechanism of imperative streams as presented here are necessary if this facility is to be useful. We reserve discussion of these extensions for future papers.

10 Acknowledgements

We would like to thank Guy Steele for challenging us to remove a write statement in a previous version of `topo`. It was this challenge which focused our attention on this problem.

References

[1] **A. Appel,** "Garbage collection can be faster than stack allocation," *Info. Proc. Lett.,* Vol. **25** (1987), pp. 275-279.

[2] **W. D. Clinger, A. H. Hartheimer, and E. M. Ost,** "Implementation strategies for continuations," *Proc. 1988 ACM Conf. on Lisp and Func. Prog.,* ACM, New York (1988), pp. 124-131.

[3] **O. Danvy,** *Memory Allocation and Higher Order Functions,* Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques (June 1987), pp. 241-252.

[4] **R. K. Dybvig,** *The Scheme Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1987).

[5] **C. T. Haynes, and D. P. Friedman,** "Embedding continuations in procedural objects," *ACM Trans. on Prog. Lang. and Sys.,* Vol. **9**, No. 4 (October 1987), pp. 582-598.

[6] **C. T. Haynes, D. P. Friedman, and M. Wand,** "Obtaining coroutines with continuations," *Comput. Lang.,* Vol. **11**, No. 3/4 (1987), pp. 143-153.

[7] **R. Harper, D. MacQueen, and R. Milner,** "Standard ML," Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh, ECS-LFCS-86-2 (1986).

[8] **Kahn, and MacQueen** "Coroutines and networks of parallel processes," *Information Processing,* B. Gilchrist (ed.), North-Holland (1977), pp. 993-998.

[9] **E. E. Kohlbecker,** *Syntactic Extensions in the Programming Language Lisp,* Ph. D. Thesis, Indiana University (1986).

[10] **J. Rees, and W. C. Clinger,** (eds): "Revised Report on the Algorithmic Language Scheme," *Sigplan Notices,* Vol. **21**, No. 12 (December 1986), pp. 37-79.

[11] **G. J. Sussman, and G. L. Steele, Jr.,** *Scheme: an Interpreter for Extended Lambda Calculus,* MIT-AIL, AI Memo No. 349, Cambridge, Massachusetts (December, 1975).

[12] **D. Ungar,** "Generation scavenging: a non-disruptive high performance storage reclamation algorithm," *Sigplan Notices,* Vol. **19**, No. 5 (May 1984), pp. 157-167.

[13] **W. W. Wadge, and E. A. Ashcroft,** *Lucid, the Dataflow Programming Language,* Academic Press (1985).

APPENDIX A
The code for implementing imperative streams

```
(define create-channel
  (lambda ()
    (let ([channel (cons '() (cons '<no-consumer-cell> '<null>))])
      channel)))

(define connect!
  (lambda (channel th)
    (let ([service-cell (cdr channel)]
          [go-there (lambda (x)
                      (let ([val (th)]
                            [consumer-cont (cdar (cdr channel))])
                        (cond [(null? val) (close! channel)]
                              [else
                               (set-car! (cdr channel) (car val))
                               (set-cdr! (cdr channel) (cdr val))
                               (consumer-cont '())])))])
      (set-car! service-cell (cons #f '<no-cont-yet>))
      (swap! go-there 'ignore (car service-cell)))))

(define open!
  (lambda (channel)
    (let* ([service-cell (cdr channel)]
           [consumer-cell (car service-cell)]
           [consumer-cont (cdr consumer-cell)]
           [producer-cell (cons '<no-value-yet> '<no-cont-yet>)])
      (set-cdr! service-cell producer-cell)
      (swap! consumer-cont service-cell producer-cell)
      (if (null? (car channel)) (cdr channel) (car channel)))))

(define share-connect!
  (lambda (share-channel original-channel)
    (let ([service-cell (cdr share-channel)]
          [stream-tail (car share-channel)]
          [producer-cell (cddr original-channel)]
          [stay-here (lambda (x) x)])
      (set-cdr! service-cell producer-cell)
      (set-car! service-cell (cons #f '<no-cont-yet>))
      (swap! stay-here 'ignore (car service-cell))
      (if (null? stream-tail) service-cell stream-tail))))
```

```
(define put-to-channel!
  (lambda (v channel)
    (let* ([service-cell (cdr channel)]
           [consumer-cell (car service-cell)]
           [demand-token (car consumer-cell)]
           [consumer-cont (cdr consumer-cell)]
           [producer-cell (cdr service-cell)]
           [new-service-cell (cons consumer-cell producer-cell)])
      (set-car! service-cell v)
      (set-cdr! service-cell new-service-cell)
      (set-car! channel service-cell)
      (set-cdr! channel new-service-cell)
      (set-car! consumer-cell #f)
      (when (eq? demand-token #t)
            (swap! consumer-cont 'ignore producer-cell)))))

(define close!
  (lambda (channel)
    (let* ([service-cell (cdr channel)]
           [consumer-cell (car service-cell)]
           [demand-token (car consumer-cell)]
           [consumer-cont (cdr consumer-cell)]
           [producer-cell (cdr service-cell)]
           [tail (car channel)])
      (when (not (null? tail)) (set-cdr! tail '()))
      (set-cdr! service-cell '<null>)
      (set-car! channel '())
      (set-car! consumer-cell '())
      (when (eq? demand-token #t)
            (swap! consumer-cont 'ignore producer-cell)))))

(define car-s
  (lambda (s)
    (when (take-step? s)
          (step! s) (when (null? (cdr s)) (car '())))
    (car s)))

(define cdr-s
  (lambda (s)
    (when (take-step? s)
          (step! s) (when (null? (cdr s)) (cdr '())))
    (cdr s)))
```

```
(define null-s?
  (lambda (s)
    (cond [(take-step? s)
           (step! s)
           (or (eqv? (cdr s) '<null>) (procedure? (cddr s)))]
          [(pair? s) (eqv? (cdr s) '<null>)]
          [else (null? s)])))

(define swap!
  (lambda (old value place)
    (let ([f (lambda (new)
               (set-cdr! place new)
               (old value))])
      (call/cc f))))

(define take-step?
  (lambda (s)
    (and (not (null? s)) (pair? (cdr s)) (procedure? (cddr s)))))

(define step!
  (lambda (s)
    (let ([consumer-cell (car s)]
          [producer-cont (cddr s)])
      (set-car! consumer-cell #t)
      (swap! producer-cont 'ignore consumer-cell))))
```

A brief description of the code

The description below is based on the code of this appendix and the figure of section 5.

`create-channel` builds a part of a `channel` which includes just a `service-cell` with `'<null>` in the cdr. The car of the `channel` is set to `'()`. This is the representation of a null stream.

The role of `connect!` is similar to `create` in section 3. It adds a `consumer-cell` with the consumer continuation in its cdr to the `channel` data structure. Then it invokes a producer procedure, with arguments. It is expected that the first statement in the producer procedure is an `open!` statement.

`open!` is placed in a producer and adds a `producer-cell`, with the producer's continuation in its cdr, to the `channel` which is its argument. It then returns control to the `connect!` statement which spawned it by invoking the continuation of the `consumer-cell`. The stream associated with an invocation of `open!` is returned to `connect!`. The next time the producer continuation is invoked, control returns to `open!`, a copy of the stream is returned locally, and the producer begins execution.

In `connect!` the procedure called `go-there` has two roles. The first, stated above, is to pass control to a producer so that it may connect its end of the `channel`. The second is to append to all streams sharing the `channel`, any stream that may be returned by the producer when it terminates. This is the purpose of the `set-car!` and `set-cdr!`. The `consumer-cont` must be used after the append in order to return control to the current suspended continuation of the consumer end. We could not simply return from `connect!` after the assignment statements. Doing so would return control to the *beginning* of the computation of the consumer process (i.e., to the first call of `connect!`) and would most probably be accompanied by a strange error message.

`share-connect!` creates a new `channel` and associated data structure which shares the same `producer-cell` as a `channel` specified as the second argument. It returns a stream pointer for the channel specified as the first argument.

`put-to-channel!` adds a cell to the list portion of the specified stream. All cells in the `channel` data structure remain fixed except for the `service-cell` which becomes the tail of the stream. The value `v` is placed in the car of that cell. A new `service-cell` is created and spliced into the `channel` data structure. Finally, if the `demand-token` is set, the consumer continuation is executed and control is returned to a consumer statement. Otherwise, execution continues within the producer.

`close!` closes the specified channel. The result is to place `'()` in the cdr of the stream tail to turn it into a list, and to set the `channel` data structure to an inactive configuration. If the `demand-token` is set, control returns to a consumer. Otherwise, execution continues within the producer.

`car-s` and `cdr-s` take the list car and cdr, respectively, if the head of the specified stream is an `s-cell`. Otherwise, a `step!` occurs and the car or cdr of the stream is returned.

`car-s` and `cdr-s` contain the line `when ... (car '())` and `when ... (cdr '())`, respectively. These lines are included to force an error when trying to take a car or cdr that does not exist.

`null-s?` forces a step if necessary and returns *true* if and only if the stream has the "null" structure or is null.

`swap!` works like `step-and-swap` of section 3. It is responsible for swapping control between producers and consumers.

`take-step?` is *true* if and only if the stream head and tail are identical.

`step!` sets the `demand-token`. Then a producer continuation is invoked, control passes to the producer, the producer encounters a `put-to-channel!` or `close!`, and creates a new stream tail for the stream. Control is passed back to `step!`.

APPENDIX B

The macro expansion of topo given it is invoked as follows:

```
(topo ([a (b c e)] [b (d f)] [c (b)] [d ()] [e (b f)] [f (d)])).
```

```
(lambda ()
  (open! topo-chnl)
  (letrec ([a (lambda ()
                (set! a (err-fun 'a topo-chnl))
                (b) (c) (e)
                (put-to-channel! 'a topo-chnl)
                (set! a (lambda () #t)))]
           [b (lambda ()
                (set! b (err-fun 'b topo-chnl))
                (d) (f)
                (put-to-channel! 'b topo-chnl)
                (set! b (lambda () #t)))]
           [c (lambda ()
                (set! c (err-fun 'c topo-chnl))
                (b)
                (put-to-channel! 'c topo-chnl)
                (set! c (lambda () #t)))]
           [d (lambda ()
                (set! d (err-fun 'd topo-chnl))
                (put-to-channel! 'd topo-chnl)
                (set! d (lambda () #t)))]
           [e (lambda ()
                (set! e (err-fun 'e topo-chnl))
                (b) (f)
                (put-to-channel! 'e topo-chnl)
                (set! e (lambda () #t)))]
           [f (lambda ()
                (set! f (err-fun 'f topo-chnl))
                (d)
                (put-to-channel! 'f topo-chnl)
                (set! f (lambda () #t)))])
   (a) (b) (c) (d) (e) (f)
   (close! topo-chnl)))
```