Using delayed streams to discern changing conditions in complex environments: Monitors in Apex 3.0

Will Fitzgerald
NASA Ames Research Center
william.a.fitzgerald@nasa.gov

Michael Freed
NASA Ames Research Center
michael.a.freed@nasa.gov

ABSTRACT

Apex is a NASA project that provides a number of components for creating and modeling intelligent agents, used for applications from human simulation to controlling an autonomous rotorcraft. Much of the core technology is written in Common Lisp. The most recent version of Apex, version 3.0, provides new capabilities for monitoring changing conditions in complex environments. In this paper, we describe these capabilities, focusing especially on the use of delayed streams for efficient computation of conditions based on state variable histories.

1. INTRODUCTION

The Apex System provides a range of components for modeling, simulating, generating and analyzing intelligent behavior. Apex is being used for a number of NASA projects, including the Autonomous Rotorcraft Project, an Army/NASA collaborative effort to produce an intelligent airborne observation platform and to develop autonomous planning and control capabilities useful for multiple heterogeneous platforms; and Human-Computer Interaction analyses, automating a complex but powerful and economically valuable model for predicting human performance at human computer interaction tasks.

Apex's Action Selection Architecture (ASA) integrates AI techniques such as hierarchical planning and online-scheduling useful for creating agents with human-level ability. By building capabilities into the architecture and providing a high-level language for behavior representation, Apex has been a powerful tool both for modeling human performance and for building soft real time execution systems.

Apex is written in Common Lisp. A high-level application development and debugging tool, Sherpa, written in Java, provides a graphical user interface. Application programming interfaces to other languages and systems are available. The Apex team at NASA's Ames Research Center is readying the release of the 3.0 version of the Apex System. Among the most important new features of the 3.0 release are new capabilities for monitoring for changing conditions in complex environments. Developers and modelers now have a much richer model of conditions to use for creating intelligent, reactive applications, such as:

- Temporal relations on events. For example, a valve closure command sent followed within 2 seconds by valve-closed signal received, or new location commands sent by an operator and not cancelled within 5 seconds.
- Temporal relations on intervals. For example, High turbulence interval overlaps communication drop-off interval.
- Coordination between control and monitoring. For example, the distance to car in front of me is less than a distance the value of which depends on my desired speed.
- Time series data analysis. For example, the temperature of instrument A has been monotonically increasing since the heater was turned on.
- Unification of querying and monitoring. For example, the temperature of instrument A has (at any point) fallen or falls below some temperature T.
- Explicit constraints on data quality. For example, the temperature of A has held steady with measurements arriving at least 1/second.
- Uniform representation based on constraints, attributes, and intervals. Apex 3.0 provides an ontology of condition types. Conditions are based on explicit or inferred measurements on state variables, or object attribute values that change over time. Essentially, the Apex 3.0 monitoring system allows the application developer to describe constraints on conditions, which (if met) allow the system to react. The overall condition ontology in Apex 3.0 includes:
- Measurement constraints: constraints on the state variable itself, the value of the state variable, or the timing of the measurement. For example, the altitude of the aircraft is greater than 34,000 between now and 1 minute from now. This includes internal Apex measurements, such as the state of a task being terminated.
- Estimation constraints: Inferred measurements based on data regression or data persistence. For example, the color of object-7 is red between now and 1 minute from now, assuming that an objects color will persist for at least five minutes or the altitude of the aircraft will be greater than 34,000 based on a linear regression of data from the last 30 seconds.

- Simple episode constraints: Abstractions over the measurement history of a state variable, including statistical, rate, step, value, timing and quality constraints. For example, the variance in the measured altitude of the aircraft exceeds some value or the measured altitude of the aircraft is monotonically increasing at some scale.

- Atomic episode constraints: External events that come into the Apex system without specifics as to the underlying state variable model. For example, a mouse click occurred.

- Complex episode constraints: Logical (and, or, and not) and temporal relations (in order, before, after, etc.) over other types of monitors, including other complex episodes. For example, step two and step three of the current procedure both must be in a terminated state and the measured state of the landing gear must be down before the altitude of the aircraft is below some level.

In this paper we will discuss some areas of interest to Common Lisp programmers, especially with regard to creating larger, complex applications in Lisp. In particular, we will focus on the use of delayed streams, which have been crucial to the effective development and use for monitors.

2. PDL AND MONITORS

Fig. 1 shows a typical procedure written in Apex's Procedure Description Language (PDL). It declares a procedure named `(look for object ?color)`. It consists of three steps. It sends a user notification when a color measurement for any object becomes available, and then sends a notification when the location measurement for the object seen becomes available. It then terminates (returning the object location).

The `waitfor` clause in step `s1` describes a simple monitor–it describes the enablement condition for starting the first notify task. The condition is simply the sensing of a measurement of any object's color. The `waitfor` clause in step `s2` is more complex–two conditions have to be true: step `s1` has to terminate and a location measurement for the object bound in step `s1` to have a location measurement sensed. Finally, the `waitfor` in step `s3` is monitoring for the termination of step `s2`–note that `?s2` alone in a monitor clause is a syntactic shorthand for `(task-state ?s2 = terminated)`. Furthermore, the variables `?s1` and `?s2` are bound to the instantiated tasks with the same name.

```
(procedure
 (index (look for object ?color))
 (step s1 (notify (i saw ?color ?object1))
   (waitfor (color ?object1 = ?color)))
 (step s2 (notify (i saw ?object1 at ?loc))
   (waitfor
    (:and (task-state ?s1 = terminated)
          (location ?object1 = ?loc)))
 (step s3 (terminate >> ?loc)
       (waitfor ?s2))))
```

Figure 1: "Typical" PDL procedure

3. STATE VARIABLES

In the Apex system, a *state variable* is an attribute, of some object, such as the *altitude* of a particular aircraft, say *aircraft-1*, the *temperature* of an *engine*, the *position* of some *button*, etc. A particularly important family of state variables is the state of the tasks being executed by the Apex system (as we saw by example in Fig 1); for example, the *state* of *task-23* might move from *pending* to *engaged* to *terminated*. Fig. 2 shows a state transition diagram for task states.

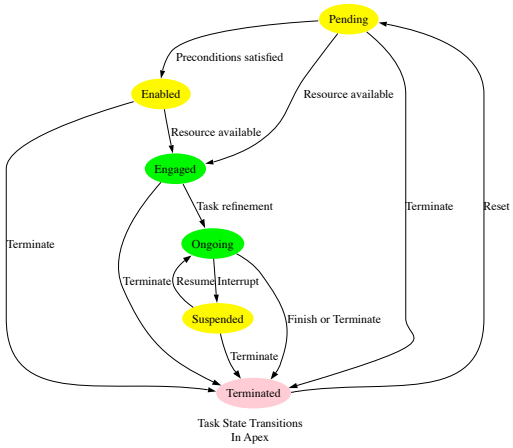

Figure 2: Task Transitions in Apex 3.0

The value of state variables can vary over time; the Apex system allows for the recording of state variable measurements along with their timestamps: for example, the altitude of an aircraft might be 10,000 at time t, 11,000 at time $t+1$, 12,000 at time $t+2$, *etc*. Apex treats these measurements lightly, as it were; that is, very little is assumed about the measurements except that they are temporally ordered and are associated with a particular state variable. [1] Formally, a measurement, m, is an unordered triple, $[s, t, v]$, where s, t, and v are the state variable, timestamp, and value, respectively, associated with m. Define $s(m), t(m), v(m)$ as functions on m giving a measurement's state variable, timestamp, and value, respectively.

The Apex system allows for the tracking of *state variable histories*, that is, the temporally ordered values of state variable measurements. A state variable history, H, is a totally ordered set of measurements such that if $m_i, m_j \in H$ and $i < j$, then $t(m_i) \leq t(m_j)$. We notate the ith members of H as H_i, and the size of H as $|H|$. Let an *interval*, i, be an ordered pair of timestamps, $\langle t_1, t_2 \rangle$ such that $t_1 \leq t_2$. We can define the interval of a non-empty state variable history, $\mathrm{interval}(H)$, as $\langle t(H_1), t(H_{|H|}) \rangle$. We can also define (possibly empty) subsets of a state variable history on a given interval. A state variable history H' is such a subset given interval $\langle i, j \rangle$ if, if $m \in H'$, then $m \in H$ and $i \leq t(m) \leq j$.

Monitoring in Apex is concerned with whether a condition occurs within an interval. In the simplest case, this is whether a predicate

[1] In particular, it is not assumed that state variable measurements necessarily persist (or not persist) over time. That is, for example, knowing that the altitude of an aircraft is 10,000 at time t and 11,000 at time $t + 10$ does not allow one to assume what the value is between $t = 0$ and $t = 10$. Similarly, knowing that a button is in the *down* position at time t does not allow one to assume what the value is at time $t+1$. Apex does provide mechanisms, as mentioned previously, to estimate such values.

is true of some state variable measurement within the interval–to repeat an example from above, a measured temperature has fallen below a setpoint value within the interval. Often, the interval is the interval implied by being "in the future,"' that is, from the point in time monitoring begins until "the end of time." The code snippet in Fig. 3 shows a step, `s1` which is initiated when a particular state variable (the temperature of the room) falls below a setpoint.

```
(step s1
  (put-on-coat)
    (waitfor
     (:measurement (temp room < 0))))
(step s2
   (put-on-shoes)
    (waitfor
    (:measurement (temp room > 0)
      :timestamp
       (> (start-of +this-task+))))
```

Figure 3: Act when temperature change.

Although, in some sense, the normal case is to monitor for conditions in the agent's future, Apex is not limited to this–monitors can check for past conditions as well, as in the monitor for step `s2` in Fig. 3[2]. Again, in the typical case, the first instance of a measurement meeting the condition is the relevant measurement; but there might be many values in an interval that meet the condition. It would be good to use data structures and algorithms that optimize for the normal, simple cases, but allow for complex checking as well. The Apex system uses delayed streams extensively in the monitoring subsystem.

4. DELAYED STREAMS

Delayed streams, sometimes called *pipes* [4], *generated lists* [2], lazy lists, or just *streams* [1], are a special kind of sequence in which all elements but the first element are calculated as needed. Among other things, they allow the ability to represent countably infinite lists (such as the integers). For our purposes however, delayed streams are just what we need: in the simple case, the first value is what is required, and the remaining values are not. The following is a quick overview of delayed streams; consult [1] or [4]for more details.

The first requirement is to represent a delayed computation. A delayed computation, called a *promise* in Scheme [3], is a function/value pair: when used, if the function is present, it should be called to produce the value (and then the function is removed); if no function is present, the value is used. To *force* the promise is just to use the promise. Fig 4 shows the simple code needed to create a promise[3].

Since delayed streams are sequences, we'll use the delayed stream equivalent of `cons`, `car` and `cdr` to create and access delayed streams. Note that `stream-cons` must be a macro–it inserts a promise where the tail is, delaying the computation of the tail until it is forced. The function `stream-force` forces the promises in a delayed stream, perhaps up to a certain number. See Fig. 5.

[2]That enablement monitors are introduced with the clause `waitfor` is thus a bit of a misnomer, but it does describe the usual case.

[3]The code presented here is based on an almagram of the code in [1] and [4].

```
(defstruct promise
  (value nil) (function nil))

(defun force (promise)
  (if (not (promise-p promise))
   promise
   (progn
   ;; calculate if needed
   (when (promise-function promise)
     (setf (promise-value promise)
       (funcall (promise-function promise)))
     (setf (promise-function promise) nil))
   ;; always return value
   (promise-value promise))))

(defmacro promise (&body body)
 `(make-promise
    :function (lambda () ,@body)))
```

Figure 4: Promise/force implementation. Quote in `promise` is a backquote.

Of course, one needs functions for mapping, filtering and appending See Fig. 6.

Note that the functions in Fig. 6 all use `stream-cons` in non-base cases. This ensures that further computation of streams is delayed. The `integers` function, defined in Fig 7, creates a delayed stream of integers; evaluting `(stream-filter (integers) 'oddp)` results in the delayed stream `(1 .#<promise>)`.

5. USING DELAYED STREAMS IN APEX MONITORS

In Apex, state variable histories themselves are represented as a data structure with three components: a state variable (which is itself a data structure), and a resizable vector containing the the state variable's measurements. A vector, rather than, say, a treap or red-black tree is used because it is almost always the case that new measurements are added at the end of the history. Vectors also allow for faster (binary) search of timepoints using intervals. The function `(first-index-within` *`history interval`*`)` returns, using binary search, the lowest index of the measurements in a state variable history within an interval (or `null` if there are no measurements within the interval). With this function in hand, it is easy to write a function that returns a stream of values from a state variable history within an interval. See Fig .8.

Again, the call to `stream-cons` means a delayed stream is created. So, with a history, H, with values:

$$\{\langle s, 1, 1\rangle, \langle s, 2, 2\rangle, \langle s, 3, 3\rangle, ...\langle s, 1000, 1000\rangle\}$$

a call to `(sv-history-stream` H $\langle 2, 1000\rangle$`)` yields the delayed stream `(`$\langle s, 2, 2\rangle$ `.#<promise>)`.

The condition monitors described in Fig. 3 are thus implementable as delayed stream filters, with `(temp room > 0)`, for example,

```
(defmacro stream-cons (head tail)
  `(cons ,head (promise ,tail)))

(defun stream-car (stream)
  (car stream))

(defun stream-cdr (stream)
  (when (promise-p (cdr stream))
   (setf (cdr stream)
     (force (cdr stream))))
  (cdr stream))

(defun stream-force (stream &optional count)
  (labels
   ((streamf (istream)
     (if (null istream)
       stream
       (streamf (stream-cdr istream))))
    (streamfc (istream cnt)
     (if (or (null istream)
             (<= cnt 0))
         stream
        (streamfc
          (stream-cdr istream) (1- cnt)))))
    (if (realp count)
      (streamfc stream count)
      (streamf stream))))
```

Figure 5: Stream accessor implementation. Quote in `stream-cons` is a backquote.

being transformed into something like[4]:

```
(stream-filter
  (lambda (m) (> (value m) 0))
  (sv-history-stream h
    (make-interval (now) (end-of-time))))
```

where `h` is the state variable history associated with the state variable `(temp room)`, `(value m)` returning $v(m)$, and `now` and `end-of-time` returning timepoints referring to the current time and the "end of time"[5].

6. THE TRICKY CASE OF AND

In addition to the simple condition monitors we have been discussing, Apex also provides for monitoring for complex comditions; that is, various combinations of logical and temporal conditions. As a simple example of a complex monitor, consider the monitor in the code snippet of Fig 9, which enables signaling "Mayday" if both engines have failed[6].

In this case, we have two state variable histories, call them E_1 and E_2, on the status of engines one and two, respectively. If I is the default interval, this monitors for the conjunction of engines one

[4]When we use the expression "something like," this indicates that the code represented isn't exactly the code found in the Apex code base, but has been modified for our presentation in this paper.

[5]In Apex, this is defined as `most-positive-fixnum`.

[6]Incidently, this code snippet also demonstrates that Apex's state variables can be discrete data as well as continuous data.

```
(defun stream-map* (fn streams)
  (if (some 'null streams)
    nil
    (stream-cons
     (apply fn
       (mapcar 'stream-car streams))
     (stream-map*
       fn (mapcar 'stream-cdr streams)))))

(defun stream-map (fn &rest streams)
  (stream-map* fn streams))

(defun stream-filter (predicate stream)
  (if (null stream)
    nil
   (if (funcall predicate
         (stream-car stream))
     (stream-cons
      (stream-car stream)
      (stream-filter
        predicate (stream-cdr stream)))
     (stream-filter
        predicate (stream-cdr stream)))))

(defun stream-append* (streams)
  (if (null streams) nil
      (let ((E1 (stream-car streams)))
    (if (null E1)
      (stream-append* (stream-cdr streams))
      (stream-cons
       (stream-car E1)
       (stream-append*
        (stream-cons (stream-cdr E1)
             (stream-cdr streams))))))))

(defun stream-append (&rest streams)
  (stream-append* streams))

(defun stream-mappend (fn stream)
  (if (null stream) nil
    (let ((x (funcall fn (stream-car stream))))
      (stream-cons
        (stream-car x)
        (stream-append
          (stream-cdr x)
          (stream-mappend
            fn (stream-cdr stream)))))))
```

Figure 6: Functions for mapping, filtering and appending of delayed streams.

```
(defun stream-iota
  (&optional (start 0) (step 1) end)
  (if (or (null end)
   (<= start end))
    (stream-cons
      start
      (stream-iota (+ start step) step end))
   '()))

(defun integers (&optional (start 0) end)
  (stream-iota start 1 end))
```

Figure 7: Delayed streams of numbers.

```
(defun sv-history-stream (history interval)
  (sv-history-stream-from
   history
   (first-index-within history interval)
   (interval-end interval)))

(defun sv-history-stream-from (h index endpoint)
  (if (or (null index)
          (>= index (item-count history)))
   nil
   (if (> (timestamp-at h index) endpoint)
     nil
     (stream-cons
       (value-at h index)
       (sv-history-stream-from
         (1+ index) endpoint)))))
```

Figure 8: State variable history streams.

```
(step (signal-mayday)
      (waitfor
       (and
         (status engine1 = failed)
         (status engine2 = failed))))
```

Figure 9: Signal "Mayday" if both engines fail.

and two failing within I. As before, we can convert an individual condition into a stream filter, for example:

```
(stream-filter
  (lambda (m) (eql (value m) 'failed))
  (sv-history-stream E1
    (make-interval (now) (end-of-time))))
```

where `E1` is the Lisp variable representing E_1; we can write a similar stream filter for E_2. What does it mean for *both* streams to return their values?

Previously, we defined a state variable histories (or subhistories) as an ordered set of measurements. The natural meaning of two state variable histories is the Cartesian cross-product of the two state variable histories E_1 and E_2' *i.e.*, the set of all pairs $\langle m_i m_j \rangle$ where $m_i \in E_1$ and $m_j \in E_2$. The naive version of a Cartesian cross-product for streams in in Fig 10.

```
(defun stream-xp-naive (E1 E2)
  (stream-mappend
   (lambda (i)
     (stream-map
       (lambda (j)
         (list i j))
       E2))
    E1))
```

Figure 10: Naive Cartesian cross-product.

The problem with Fig 10's algorithm is that it doesn't work well with infinite lists. Consider all pairs of integers: `(naive-xp (integers) (integers))`. This will return `((0 0) (0 1) (0 2) ...)`; in other words, it will never produce a pair `(1 0)`, because all the integers have to be consumed before starting over. It must be said that the algorithm in Fig 10 is probably acceptable for Apex monitors; streams used in monitoring are unlikely to be infinite in length. However, by interleaving the steams, we can produce an infinite stream of pairs of numbers that does, in fact, reach `(1 0)` and other values by working diagonally though the streams. [1] have a good discussion of this idea; the cross-product function in Fig 11 is based on their ideas. An informal proof that the algorithm of Fig 11 computes the Cartesian cross-product can be found in the Appendix.

Whichever algorithm is used (I prefer the algorithm in Fig 11 just to protect against failure using infinite lists), the functional equivalent of Fig 9 is something like:

```
(stream-xp/2
  (stream-filter
    (lambda (m) (eql (value m) 'failed))
    (sv-history-stream E1
      (make-interval (now) (end-of-time))))
  (stream-filter
    (lambda (m) (eql (value m) 'failed))
    (sv-history-stream E2
      (make-interval (now) (end-of-time)))))
```

The general Cartesian cross product function is easy enough to define; it can be found in Fig 12. The basic idea is that one stream

```
(defun stream-interleave (E1 E2)
  (if (null E1)
    E2
    (stream-cons
     (stream-car E1)
     (stream-interleave
      E2
      (stream-cdr E1)))))

(defun stream-xp/2 (E1 E2)
  (if (or (null E1) (null E2))
    nil
    (stream-cons
     (list
      (stream-car E1)
      (stream-car E2))
     (stream-interleave
      (stream-interleave
       (stream-map
        (lambda (x)
         (list(stream-car E1) x))
        (stream-cdr E2))
       (stream-map
        (lambda (x)
         (list x (stream-car E2)))
        (stream-cdr E1)))
      (stream-xp/2
       (stream-cdr E1)
       (stream-cdr E2))))))
```

Figure 11: Cartesian cross-product using interleaving.

```
(defun stream-xp* (stream-of-streams)
  (cond
   ;; no sets? {}
   ((null stream-of-streams) nil)
   ;; one set? list of each element
   ((null (stream-cdr stream-of-streams))
    (stream-map 'list
      (stream-car stream-of-streams)))
   ;; two sets? call stream-xp/2
   ((null
      (stream-cdr
        (stream-cdr stream-of-streams)))
    (stream-xp/2
     (stream-car stream-of-streams)
     (stream-car
        (stream-cdr stream-of-streams))))
   ;; more than two? still call stream-xp/2
   ;; but fix up results that come back
   (t
    (stream-map
     (lambda (pair)
       (cons (car pair) (cadr pair)))
       (stream-xp/2
     (stream-car stream-of-streams)
      (stream-xp*
       (stream-cdr stream-of-streams)))))))

(defun stream-xp (&rest streams)
  (stream-xp* streams))
```

Figure 12: Generalized Cartesian cross-product for streams.

produces a list of each element, two streams call `stream-xp/2`, and three or more streams call `stream-xp/2` on the first stream combined with a recursive call to the rest of the streams. Because `stream-xp/2` always returns a list of pairs, we need to fix up the recursive calls so that a flat list is returned.

Unfortunately, this is not quite what is needed for Apex. Apex monitors allow pattern matching, and variable bindings can be passed to the enclosing task. Consider the following monitor, which monitors for engine temperatures. In this case, the value of the temperatures are to be captured, and the value of the second engine's temperature has to be more than 10 degrees hotter than the value of the first engine's temperature. The `(:measurement ...)` form allows an Apex programmer to define additional constraints on a measurement monitor.

```
(step (warn engine2 hot)
  (waitfor
   (and (temp engine1 = ?x)
        (:measurement
           (temp engine2 = ?y)
           :value (?y > (+ ?x 10))))))
```

Let us consider this in parts; first the simpler conjunct `(temp engine1 = ?x)`. The stream that has to be returned is not, in fact, a stream of measurements, but a stream of binding sets. A binding set is just a set of variable/value pairs. Apex uses a system based on the binding set code in [4], which provides functions such as `make-binding-set`, `extend-bindings`, and `substitute-bindings`, as well as pattern matching code. Thus, the form `(temp engine1 = ?x)` should turn into something like Fig. 13.

```
(stream-map
  (lambda (m)
    (extend-bindings
     '?x
     (value m)
     (make-binding-set)))
  (sv-history-stream E1
    (make-interval (now) (end-of-time))))
```

Figure 13: Mapping to bindings. `E1` is the state variable history associated with the state variable `(temp engine1)`.

The difficult thing is converting the second conjunct. We want to provide something like the code in Fig. 13, but we must filter values first based on the values of the variables `?x`–something like Fig. 14.

The question is how to provide a stream of binding sets to the filter in Fig. 14, and how to do this in a general way. The solution is to use the Cartesian product of Fig. 13 and the filtered stream Fig. 14, which we show in Fig. 15.

The resultant binding stream of Fig. 15 will result in either an empty or a non-empty stream. If the stream is empty, the monitor fails (but may succeed as new measurements arrive). If it succeeds, the usual case is that only the first binding set is used. Thus, even

```
(stream-map
  (lambda (m)
    (extend-bindings
     '?y
     (value m)
     B))
  (stream-filter
    (lambda (m)
      (> (value m)
        (+ (substitute-bindings '?x B))))
    (sv-history-stream E2
      (make-interval (now) (end-of-time)))))
```

Figure 14: Mapping and filtering bindings. `E2` is the state variable history associated with the state variable `(temp engine2)`, and `B` is a binding set returned in the stream from Fig 13.

```
(stream-map
  (lambda (m)
    (extend-bindings
     '?y
     (value m)
     B))
   (stream-xp
      (stream-map
    (lambda (m)
      (extend-bindings
      '?x
      (value m)
      (make-binding-set)))
    (sv-history-stream E1
     (make-interval (now) (end-of-time))))
    (stream-filter
     (lambda (m)
      (> (value m)
        (+ (substitute-bindings '?x B))))
      (sv-history-stream E2
        (make-interval (now) (end-of-time))))
```

Figure 15: Passing bindings from one stream to another. `E1` and `E2` are the state variable histories associated with the state variables `(temp engine1)` and `(temp engine2)`, respectively.

though we are taking the Cartesian product of two streams, E_1 and $E_2|$, which has the obvious worst-case computational complexity of $|E_1| \times |E_2|$, we have a best case of (modulo the setup of creating the streams) of looking at just one pair.

Of course, we don't expect users writing applications using Apex to write code like that of Fig. 15. Rather, we want to transform the code from Apex's Procedure Description Language, such as that shown in the Apex code snippets shown above, into the kinds of function calls shown above. We do this in the usual Lisp way, by defining Lisp macros and writing code transformation rules. In fact, the transformations are a bit more complicated than indicated in this paper, in order to support the syntax we want for monitors in PDL.

7. CONCLUSION

Our goals in creating the monitor subsystem for Apex 3.0 included creating a flexible and intuitive language for describing conditions in the world based especially on measured values. In creating this flexibility, it is easy to lose the capacity to create efficient-enough applications. Providing support for conjunctive condition monitoring must, in the worst case, be computationally very expensive, as we have hinted. With careful programming of the monitor subsystem, we can ameliorate this expense, even if we cannot remove it entirely. One strategy we have used is to use delayed streams to represent the combinations of conditions that can occur. Because in the normal case, only one condition result is used, delayed streams allow us to avoid much of the computational overhead. Although delayed streams are a standard enough feature in introductory functional programming texts, we have elaborated on our use of them in Apex as an example of a real application of them, and to encourage others to consider adding delayed streams to their Lisp programming toolbox.

Common Lisp code implementing the delayed streams system described here is available on request to the first author.

Acknowledgments

We would like to thank Khalil Michael Dalal for helpful comments on a previous draft.

8. REFERENCES

[1] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Programs, 2nd edition*. MIT Press, 1996.

[2] CHARNIAK, E., RIESBECK, C., MCDERMOTT, D., AND MEEHAN, J. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1987. second edition.

[3] KELSEY, R., CLINGER, W., AND REES, J. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (Sept. 1998), 26–76.

[4] NORVIG, P. *Paradigms of Artificial Intelligence Programming:Case Studies in Lisp*. Morgan Kaufmann, 1992.

APPENDIX

Cartesian Product Proof

Algorithm 1 is an implementation of `stream-xp/2` given in Fig. 11, written in a general algorithmic style, and showing intermediate results.

Algorithm 1 $\mathrm{sxp}(S, T)$: function to return the Cartesian cross product of s and t as a delayed stream. Scar, scdr, *etc.*, are `stream-car`, `stream-cdr`, *etc.*.

Require: S and T are streams.

1: **if** $\mathrm{Null?}(S) \vee \mathrm{Null?}(T)$ **then**
2: return *the empty stream*
3: **else**
4: $a \leftarrow \mathrm{list}(\mathrm{scar}(S), \mathrm{scar}(T))$
5: $r \leftarrow \mathrm{smap}(\lambda x.\mathrm{list}(\mathrm{scar}(S), x), \mathrm{scdr}(T))$
6: $c \leftarrow \mathrm{smap}(\lambda x.\mathrm{list}(x, \mathrm{scar}(T)), \mathrm{scdr}(T))$
7: $rc \leftarrow \mathrm{sinterleave}(r, c)$
8: return $\mathrm{scons}(a, \mathrm{sinterleave}(rc, \mathrm{sxp}(\mathrm{scdr}(S), \mathrm{scdr}(T))))$
9: **end if**

We need to show two things. First, that Algorithm 1 computes the Cartesian cross product of two delayed streams, and, second, that it does so without infinite recursion.

To show that Algorithm 1 computes the Cartesian cross product of two delayed streams, first, note that if either S or T are empty streams, the Cartesian cross product is empty by definition. Lines 1 and 2 check this base case. In the non-base case, the Cartesian cross product is computed in four parts, visually displayed in Table 1. The first part is the upper left cell represented by the pair S_i, T_i. This is computed in line 4. The second is the stream of pairs from S_{i+1}, T_i to $|S|, T_i$–that is, the first row past the initial element. This is calculated in line 5. The third is the stream of pairs from S_i, T_{i+1} to $S_i, |T|$–that is, the first column past the initial element. This is calculated in line 6. Finally, the last part is the Cartesian cross product of the substreams consisting of S_{i+1} to $S_{|S|}$ and T_{i+1} to $T_{|T|}$. This is calculated in the recursive call to Algorithm 1 in line 8. The returned value from Algorithm 1 is the `cons` of the upper left cell (i.e., S_i, T_i), plus the interleaving of the row and column, interleaved with the recursive cross product. By inspection, this is the cross product. Note that if there is either not an element i of either S or T, then at some substream of S or T where one of them "ran out" (i.e., $|S|$ or $|T|$ was less than i), the recursive call to Algorithm 1 would be called with a null stream, and the null check of line 1 would obtain. By similar reasoning, if the length of S is finite, then the cross product will have a finite number of columns; if the length of T is finite, the cross product will have a finite number of rows. And, if both $|S|$ and $|T|$ are finite, the cross product is finite, and the algorithm is guaranteed to terminate.

Table 1: Composition of Cartesian cross product

	S_i	S_{i+1}	S_{i+2}	...	$\|S\|$
T_i	S_i, T_i	S_{i+1}, T_i	S_{i+2}, T_i	...	$\|S\|, T_i$
T_{i+1}	S_i, T_{i+1}	*recur...*			
T_{i+2}	S_i, T_{i+2}				
...	...				
$\|T\|$	$S_i, \|T\|$				

To show that Algorithm 1 does not fall prey to the "infinite recursion" problem described above–that is, on infinite streams, eventually for some finite i all pairs from S_1, T_1 to S_i, T_i will be generated in a finite number of steps, first note that the (possibly) infinite streams generated in lines 5 and 6–that is, the 'row' and 'column' components–are interleaved in line 7. This prevents the "infinite recursion" problem for the row and the column. Furthermore, in line 8, this interleaved stream is interleaved with the (possibly infinite) recursive cross-product of S_{i+1} to $S_{|S|}$ and T_{i+1} to $T_{|T|}$. The prevents the "infinite recursion" problem altogether, since the resulting stream is made the `cdr` of the new stream headed by the pair the pair S_i, T_i.