Demand-driven real-time computing*

Jean-Raymond Gagné           John Plaice

May 19, 1999

Abstract

We present the RLUCID language, which is Lucid with a synchronous time semantics, along with an extra operator, before, to allow for reactivity. We translate the index-based semantics into a timestamp-based semantics, and, by using demand-driven techniques, show how the language can be implemented efficiently.

1 Introduction

The RLUCID language is a timestamped extension to Ed Ashcroft and Bill Wadge's Lucid. RLUCID was designed by John Plaice [2], following initial work undertaken by Sarma Vempari for his M.Sc. thesis [3], and discussion with Bill Wadge. The RLUCID language has never been implemented, and we show in this paper that it can be translated into a timestamp-based formalism, where the timestamps are in the domain $T = R \times Z$.

Simply stated, RLUCID is unidimensional Lucid with a timestamped semantics, along with a single new operator, before. This binary Boolean operator incorporates the essence of reactive systems: the ability to make choices according to the order of arrival of inputs.

Like in Lucid, RLUCID streams are indexed by the natural numbers. In addition, each daton is timestamped by a real number. Hence, two dataflows $X$ and $Y$ can be considered to be streams of pairs $(value, timestamp)$:

$$X = ([x_0, s_0], [x_1, s_1], \ldots, [x_i, s_i], \ldots)$$

and

$$Y = ([y_0, t_0], [y_1, t_1], \ldots, [y_i, t_i], \ldots).$$

To ease the reading, the notation $[x, s]$ will be used instead of the more usual $(x, s)$.

*Presented to the International Symposium on Languages for Intensional Programming, Demokritos Institute, Athens, Greece, 28–30 June 1999. Current addresses: Jean-Raymond Gagné, Département d'informatique, Université Laval, Québec (Québec) Canada G1K 7P4. John Plaice, School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia. Email: gagne@ift.ulaval.ca, plaice@cse.unsw.edu.au.
As in Lucid, data operations are applied pointwise (over the indices) to their arguments. Since RLUCID is a synchronous language, the application of, say, + to two streams is applied as soon as the operands are available, as in:

\[ X + Y = \left( [x_0 + y_0, \max(s_0, t_0)], \ldots, [x_i + y_i, \max(s_i, t_i)], \ldots \right). \]

In other words, in RLUCID, if a daton shows up at a binary computation node and there is no matching daton on the other line, then the first daton is queued until its partner shows up.

Constants are considered to be infinite streams that are available at "the beginning of time", say 0. So, constant \( k \) actually means

\[ ([k, 0], [k, 0], \ldots, [k, 0], \ldots). \]

Lucid is essentially ISWIM with two additional operators, next and fby ("followed-by"), that allow manipulation of streams. Suppose that \( X \) and \( Y \) are as above. Then

\[ \text{next } X = ([x_1, s_1], [x_2, s_2], \ldots, [x_{i+1}, s_{i+1}], \ldots) \]

and

\[ X \text{ fby } Y = ([x_0, s_0], [y_0, t_0], \ldots, [y_{i-1}, t_{i-1}], \ldots). \]

RLUCID's new operator is the Boolean before: If \( X \) and \( Y \) are as above, then

\[ X \text{ before } Y = \left( [s_0 \leq t_0, \min(s_0, t_0)], \ldots, [s_i \leq t_i, \min(s_i, t_i)], \ldots \right). \]

Section 2 will show that by using this operator in recursive functions, many interesting operators can be defined.

A couple of notes on the semantics should be made. First, the above presentation allows situations where timestamps are not totally ordered. Although there are situations where this approach is acceptable, the presentation in this paper will disallow it, as our streams have a real-time interpretation: we want to ensure that timestamps are fully ordered.

Second, it is possible to define streams that have infinite bursts of datons, all with the same timestamp. This is clearly the case for constants, but also for expressions such as:

\[ X = 0 \text{ fby } X + 1 \]

which defines the flow

\[ ([0, 0], [1, 0], \ldots, [i, 0], \ldots). \]

We will see below how the fby operator can be redefined to get rid of these problems.
2 RLUCID examples

There are several RLUCID constructs which are heavily used which can be derived from the base language. Many of these constructs are defined as recursive functions.

- `first` \(X\) generates a constant stream, whose value is the first value of \(X\):

  \[
  \text{first } X = X \ \text{fby first } X
  \]

- \(X\ wvr\ Y\) generates the corresponding value of \(X\) every time that \(Y\) is true:

  \[
  X \ wvr\ Y = \begin{cases} 
  \text{if } Y \\
  \quad \text{then } X \ \text{fby (next } X \ wvr\ \text{next } Y) \\
  \quad \text{else} \\
  \quad \quad \text{(next } X \ wvr\ \text{next } Y)
  \end{cases}
  \]

- `event` \(X\) generates a void (empty type) stream which is synchronous with \(X\). It will be used to define other operations which do not care about the value of \(X\). The `||` is a tuple-building data operation and `select` extracts \(n\)-th component from a tuple.

  \[
  \text{event } X = \text{select}(1, \ \text{void} | | X)
  \]

  Here, `void | | X` builds a tuple, and `select(1, \ void | | X)` extracts the void value (which is the first tuple element) along with \(X\)'s timestamps.

- `A on C` generates a stream whose values are those of \(A\) but whose time-stamps are those of \(C\) if \(A\) is faster than \(C\):

  \[
  A \ on\ C = \text{select}(1, \ A | | C)
  \]

  Suppose we wish to count the occurrences of an event \(A\), exactly when they occur. This would be done as follows:

  \[
  \begin{align*}
  \text{count on (void fby event } A) \\
  \text{where} \\
  \quad \text{count} = 0 \ \text{fby count + 1} \\
  \text{end}
  \end{align*}
  \]

  \(\text{count}\) counts the natural numbers infinitely fast. The \(\text{on}\) slows the counter to the rate of \(A\).

- `choose` \(X_1 : E_1\ X_2 : E_2\ \ldots\ X_n : E_n\ \text{end}\) will execute the \(E_i\) corresponding to the first \(X_i\) which has arrived. It is defined as follows:
choose X1:E1 X2:E2 ... Xn:En end =
    if X1 before X2 and ... and X1 before Xn then E1
    else if X2 before X3 and ... and X2 before Xn then E2
    ... 
    else En

• last(A,C) outputs the last value held by A every time that C occurs. It is defined as follows:

    last(A,C) = current(A, C, first A)
    where
    current(A, B, C) =
    choose
    A: current(next A, B, first A)
    B: C fby current(A, next B, C)
    end
    end

Suppose that we wish to take the count of occurrences of A from above and use it whenever B turns up. The result would be:

    last(occurs, event B)
    where
    occurs = count on (void fby event A)
    count = `0'.fby count + 1
    end

• MergeLeft will generate a single stream from two input streams. Should the two send values at the same time, then the left one takes precedence:

    MergeLeft(A,B) =
    choose
    A: A fby MergeLeft(next A, B)
    B: B fby MergeLeft(A, next B)
    end

• SignalLeftFair will generate a single stream from two input streams. However, the elements will be pairs of Booleans, showing which stream generated the pair. Only one true value will be sent at a time:

    SignalLeftFair(A,B) =
    choose
    A||B: (true||false) fby (false||true) fby SignalLeftFair(next A, next B)
    A: (true||false) fby SignalLeftFair(next A, B)
    B: (false||true) fby SignalLeftFair(A, next B)
    end
• SignalBoth will send two true values at a time, if necessary:

\[
\text{SignalBoth}(A, B) = \\
\begin{align*}
\text{choose} \\
A & | B: (\text{true} | \text{true}) \text{ fby } \text{SignalBoth}(\text{next } A, \text{ next } B) \\
A: (\text{true} | \text{false}) \text{ fby } \text{SignalBoth}(\text{next } A, \text{ B}) \\
B: (\text{false} | \text{true}) \text{ fby } \text{SignalBoth}(A, \text{ next } B)
\end{align*}
\]

3 RLUCID’s indexical semantics

We introduce now RLUCID dataflow definition. To each daton, we associate a value along with a timestamp. **Definition 1:** Let \( D_x \) be a set of values and let \( R' \) be a totally ordered set of timestamps. An RLUCID dataflow \( z \) over \( D_x \) is a pair of functions \((\mathcal{H}_x, \mathcal{F}_x)\) such that \( \mathcal{H}_x : N \to R' \) and \( \mathcal{F}_x : N \to D_x \). We write \( \mathcal{H}(x) \) for \( \mathcal{H}_x \) when it is more legible. Similarly for \( \mathcal{F}(x) \) and \( \mathcal{F}_x \).

Let \( x = (\mathcal{H}_x, \mathcal{F}_x) \) be a dataflow over \( D_x \), \( y = (\mathcal{H}_y, \mathcal{F}_y) \) be a dataflow over \( D_y \) and, for each \( i, x_i = (\mathcal{H}_{x_i}, \mathcal{F}_{x_i}) \) be a dataflow over \( D_{x_i} \). Furthermore, let \( f : D_{x_1} \times \cdots \times D_{x_m} \to D_z \) be a mapping. Then Table 1 gives the semantics for the RLUCID operators. It turns out that the skip operator is easier to use than the next operator for the subsequent semantic work; see Gagné’s thesis for more details.

**Table 1:** The original semantics for RLUCID

<table>
<thead>
<tr>
<th>RLUCID operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = k )</td>
<td>( \mathcal{H}_z n = 0 ) ( \mathcal{F}_z n = k )</td>
</tr>
<tr>
<td>( z = \text{op}(x_1, \ldots, x_m) )</td>
<td>( \mathcal{H}<em>z n = \max(\mathcal{H}</em>{x_1} n, \ldots, \mathcal{H}<em>{x_m} n) ) ( \mathcal{F}<em>z n = \text{op}(\mathcal{F}</em>{x_1} n, \ldots, \mathcal{F}</em>{x_m} n) )</td>
</tr>
<tr>
<td>( z = \text{next } x )</td>
<td>( \mathcal{H}_z n = \mathcal{H}_x(n + 1) ) ( \mathcal{F}_z n = \mathcal{F}_x(n + 1) )</td>
</tr>
<tr>
<td>( z = x \text{ fby } y )</td>
<td>( \mathcal{H}_z n = \begin{cases} \mathcal{H}_y 0 &amp; \text{if } n = 0 \ \mathcal{H}_y(n - 1) &amp; \text{otherwise} \end{cases} ) ( \mathcal{F}_z n = \begin{cases} \mathcal{F}_y 0 &amp; \text{if } n = 0 \ \mathcal{F}_y(n - 1) &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>( z = x \text{ skip } s )</td>
<td>( \mathcal{H}_z n = \max(\mathcal{H}_x n, \mathcal{H}<em>z(\sum</em>{i=0} n \mathcal{F}_i i)) ) ( \mathcal{F}_z n = \mathcal{F}<em>x(\sum</em>{i=0} n \mathcal{F}_i i) )</td>
</tr>
<tr>
<td>( z = x \text{ before } y )</td>
<td>( \mathcal{H}_z n = \min(\mathcal{H}_x n, \mathcal{H}_y n) ) ( \mathcal{F}_z n = \begin{cases} \text{true} &amp; \text{if } \mathcal{H}_z n \leq \mathcal{H}_y n \ \text{false} &amp; \text{otherwise} \end{cases} )</td>
</tr>
</tbody>
</table>
4 Bursts in RLUCID

Before RLUCID programs can be translated into equivalent timestamped programs, some decisions must be taken with regards to the semantics of some of RLUCID’s operators.

First, we wish to be able to deal with real timestamps, i.e. we want the timestamps of datons to correspond to real time. We must therefore introduce the constraint that the timestamps of a dataflow be ordered, i.e. \( \mathcal{H}_\tau n \leq \mathcal{H}_\tau (n + 1) \).

Second, we have not addressed the problem of bursts, where more than one daton, possibly an infinite set, can appear at any given instant \( t \). Bursts are one of the reasons that no implementation of RLUCID have ever been made.

Bursts are produced by constants and RLUCID expressions such as

\[
X = 0 \ fby \ X + 1,
\]

in this case an infinite set of integers, all sharing the same timestamp. The problem with bursts appears when they are combined with the RLUCID before operator.

As an example, consider stream \( Y \) in the expression:

\[
Y = X \ merge \ X
\]

where \( X \ merge \ Y = \begin{cases} \text{if } X \ before \ Y \\ \text{then } X \ fby \ ((\text{next } X) \ merge \ Y) \\ \text{else } Y \ fby \ (X \ merge \ (\text{next } Y)) \end{cases}
\]

The intuition for \( Y \) is that it merges — according to the timestamps — two copies of stream \( X \), i.e. a fair semantics for merge would result in the stream \( (0, 0, 1, 1, 2, 2, 3, 3, \ldots) \) for \( Y \): datons would be alternately taken from each of the two copies of \( X \). However, since in RLUCID there is no way to distinguish two datons in a burst through their timestamps, stream \( Y \) is in fact the sequence \( (0, 1, 2, 3, \ldots) \). In the timestamped translation given below, we will give a subtly different semantics to RLUCID that will ensure a fair semantics for merge.

The semantics for RLUCID’s \( fby \) operator must also be addressed. In the expression \( Z = X \ fby \ Y \), the first daton of \( Z \) is the first daton of \( X \), and the subsequent datons are those of \( Y \) shifted from the present to the future by one place in \( Y \)'s timestamp sequence. A question arises: What happens to the \( Y \) datons that occur before the occurrence of \( X \)'s first daton?

There are two possibilities: either they are lost (and do not appear in the resulting stream) or they are memorized until they are output in a burst. Given the semantics of RLUCID, it would be inappropriate to lose the datons, so some method must be found to deal with the potential resulting bursts. This could be handled by building queues corresponding to sets of unused datons on a given line, however this would lead to unbounded memory situations, which we want to avoid.

A solution to the burst problem appears if we redefine the notion of constant as well as the semantics of the \( fby \) operator, which makes use of timestamps defined over \( T_1 \).

Table 2 gives the new RLUCID semantics. It is still based on indices, using the real-time timestamp interpretation.
<table>
<thead>
<tr>
<th>RLUCID operator</th>
<th>Semantics</th>
</tr>
</thead>
</table>
| $z = k$ | $\mathcal{H}_z n = (0, n)$  
$\mathcal{F}_z n = k$ |
| $z = \text{op}(x_1, \ldots, x_m)$ | $\mathcal{H}_z n = \max(\mathcal{H}_{x_1} n, \ldots, \mathcal{H}_{x_m} n)$  
$\mathcal{F}_z n = \text{op}(\mathcal{F}_{x_1} n, \ldots, \mathcal{F}_{x_m} n)$ |
| $z = \text{skip } s$ | $\mathcal{H}_z n = \max(\mathcal{H}_z n, \mathcal{H}_z (\sum_{i=0}^{n} \mathcal{F}_z i))$  
$\mathcal{F}_z n = \mathcal{F}_z (\sum_{i=0}^{n} \mathcal{F}_z i)$ |
| $z = x \text{ fby } y$ | $\mathcal{H}_z n = \begin{cases} \mathcal{H}_z 0 & \text{if } n = 0 \\ \max(\mathcal{H}_y(n-1), \mathcal{H}_z(n-1) + (0, 1)) & \text{otherwise} \end{cases}$  
$\mathcal{F}_z n = \begin{cases} \mathcal{F}_z 0 & \text{if } n = 0 \\ \mathcal{F}_y(n-1) & \text{otherwise} \end{cases}$ |
| $z = x \text{ before } y$ | $\mathcal{H}_z n = \min(\mathcal{H}_x n, \mathcal{H}_y n)$  
$\mathcal{F}_z n = \begin{cases} \text{true} & \text{if } \mathcal{H}_z n \leq \mathcal{H}_y n \\ \text{false} & \text{otherwise} \end{cases}$ |

Constants generate datons with timestamps from successive micro-instants within the same macro-instant.

The operator $\text{fby}$ can no longer generate bursts, as it takes into account the timestamp of the previously generated daton $\mathcal{H}_z(n-1) + (0, 1)$ while yielding a new one.

We have replaced the $\text{next}$ operator by the $\text{skip}$ operator.

5 \hspace{1cm} RLUCID's timestamp-based semantics

We present here a semantics for RLUCID using only timestamps and prove the equivalence with the previous semantics.

**Definition 2:** For each RLUCID dataflow $x_R = (\mathcal{H}_x, \mathcal{F}_x)$, we define an equivalent timestamped dataflow $x = (\mathcal{C}_x, \mathcal{V}_x)$, such that:

$$
\mathcal{C}_x = \{ \mathcal{H}_x n \mid n \in \mathbb{N} \}
$$

$$
\mathcal{V}_x t = \mathcal{F}_x(\mathcal{I}_x t)
$$

where

$$
\mathcal{I}_x t = \#\{ t' \in \mathcal{C}_x \mid t' \leq t \}.
$$

The index ($\mathcal{I}$) translates an implicit timestamp $t$ into an index $n$.

Applying these rules to the semantics of RLUCID yields a new timestamp-based se-
mantics, given in Table 3, where

\[ \mathcal{R}_x t = \sum_{t' \in C_x, t' \leq t} \nu_x t'. \]

The running sum (\(\mathcal{R}\)) computes the sum of the data whose timestamps are less than the argument \(t\).

**Proposition 1** The new semantics for RLUCID is consistent with the original semantics.

Table 3: Timestamped semantics for RLUCID

<table>
<thead>
<tr>
<th>RLUCID operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(z = k)</td>
<td>(C_z = {0} \times \mathbb{N})</td>
</tr>
<tr>
<td>(\nu_x t = k)</td>
<td>(\nu_x t = \mathcal{F}_z(I_x t))</td>
</tr>
<tr>
<td>(z = op(x_1, \ldots, x_m))</td>
<td>(C_z = {t \mid \exists i \leq m : t \in C_{x_i} \land \forall j \leq m : I_{x_j} t \leq I_{x_j} t})</td>
</tr>
<tr>
<td></td>
<td>(\nu_x t = op(\nu_{x_1}(C_{x_1}(I_{x_1} t)), \ldots, \nu_{x_m}(C_{x_m}(I_{x_m} t))))</td>
</tr>
<tr>
<td>(z = x) skip (s)</td>
<td>(C_z = {t \mid t \in C_x \land \mathcal{R}_x t \leq I_x t})</td>
</tr>
<tr>
<td></td>
<td>(\nu_x t = \nu_x(\mathcal{R}_x t))</td>
</tr>
<tr>
<td>(z = x) fby (y)</td>
<td>(C_z = {t \mid t \in C_y \land t &gt; C_x 0} \land \pi_r(t) = \pi_r(C_x 0) \land I_y t \geq I_x t \geq C_x 0)</td>
</tr>
<tr>
<td></td>
<td>(\nu_x t = \nu_{x_1}(C_x(I_x t)))</td>
</tr>
<tr>
<td>(z = x) before (y)</td>
<td>(C_z = {t \mid t \in C_y \land I_x t \geq I_y t})</td>
</tr>
<tr>
<td></td>
<td>(\nu_x t = \nu_y(C_y(I_x t)))</td>
</tr>
</tbody>
</table>

**Proof** We show that each new semantics respects the given rules of equivalence. We write \(C_x n\) to extract the \(n\)-th timestamp of the ordered set \(C_x\).

- \(z = k\)

\[
C_z = \{(0, n) \mid n \in \mathbb{N}\} = \{0\} \times \mathbb{N}
\]

\[
\nu_x t = \mathcal{F}_z(I_x t) = k
\]
\[ z = \text{op}(x_1, \ldots, x_m) \]

\[ C_z = \{ \max(\mathcal{H}_{x_1}n, \ldots, \mathcal{H}_{x_m}n) \mid n \in \mathbb{N} \} \]

\[ = \{ \max(C_{x_1}n, \ldots, C_{x_m}n) \mid n \in \mathbb{N} \} \]

\[ = \{ t \in T \mid \exists n \in \mathbb{N}, \exists i \leq m : t = C_{x_i}n \land \forall j \leq m : n \leq \#\{ t' \in C_{x_j} \mid t' \leq t \} \} \]

\[ = \{ t \mid \exists i \leq m : t \in C_{x_i} \land \forall j \leq m : \#\{ t' \in C_{x_j} \mid t' \leq t \} \leq \#\{ t' \in C_{x_j} \mid t' \leq t \} \} \]

\[ = \{ t \mid \exists i \leq m : t \in C_{x_i} \land \forall j \leq m : I_{x_i}t \leq I_{x_j}t \} \]

\[ \nu_{z:t} = \mathcal{F}_z(I_{z:t}) \]

\[ = \text{op}(\mathcal{F}_{x_1}(I_{z:t}), \ldots, \mathcal{F}_{x_m}(I_{z:t})) \]

\[ = \text{op}(\nu_{x_1}(\mathcal{C}_{x_1}(I_{z:t})), \ldots, \nu_{x_m}(\mathcal{C}_{x_m}(I_{z:t}))) \]

\[ z = x \quad \text{skip} \quad s \]

\[ C_x = \left\{ \max(\mathcal{H}_x n, \mathcal{H}_x (\sum_{i=0}^n \mathcal{F}_{x:i})) \mid n \in \mathbb{N} \right\} \]

\[ = \left\{ t \in T \mid \exists n \in \mathbb{N} : t = C_x n \land (\sum_{i=0}^n \mathcal{F}_{x:i}) \leq \#\{ t' \in C_x \mid t' \leq t \} \right\} \]

\[ = \left\{ t \in T \mid \exists n \in \mathbb{N} : t = C_x (\sum_{i=0}^n \mathcal{F}_{x:i}) \land n \leq \#\{ t' \in C_x \mid t' \leq t \} \right\} \]

\[ = \left\{ t \mid t \in C_x \land R_x t \leq I_{z:t} \right\} \}

\[ \nu_{z:t} = \mathcal{F}_x(I_{z:t}) \]

\[ = \mathcal{F}_x\left(\sum_{i=0}^{I_{z:t}} \mathcal{F}_{x:i}\right) \]

\[ = \mathcal{F}_x(R_x t) \]

\[ = \nu_x(C_x(R_x t)) \]
\[ z = x \text{ fby } y \]

\[ C_z = \{ t : t = \min(C_z) \lor \exists n > 0 : t = \max(\mathcal{H}_y(n-1), \mathcal{H}_z(n-1) + \langle 0, 1 \rangle) \} \]

\[ = \{ t : t = C_z 0 \lor \exists n \geq 0 : t = \max(\mathcal{H}_y n, \mathcal{H}_z n + \langle 0, 1 \rangle) \} \]

\[ = \begin{cases} 
  t = C_z 0 \\
  \forall t \in C_y \land \#\{ t' \in C_y \mid t' \leq t \} \leq \#\{ t' \in (C_z + \epsilon) \mid t' \leq t \} \\
  \forall t \in (C_z + \epsilon) \land \#\{ t' \in (C_z + \epsilon) \mid t' \leq t \} \leq \#\{ t' \in C_y \mid t' \leq t \} 
\end{cases} \]

\[ = \begin{cases} 
  t = C_z 0 \\
  \forall t \in C_y \land \mathcal{I}_y t \leq \#\{ t' \in C_z \mid t' < t \} \\
  \forall t \in (C_z + \epsilon) \land \#\{ t' \in C_z \mid t' < t \} \leq \mathcal{I}_y t 
\end{cases} \]

\[ = \begin{cases} 
  t = C_z 0 \\
  \forall t \in C_y \land t > C_z 0 \\
  \forall t \in (C_z + \epsilon) \land \#\{ t' \in C_z \mid t' < t \} \leq \mathcal{I}_y t 
\end{cases} \]

\[ \nu_z t = \mathcal{F}_z(\mathcal{I}_z t) \]

\[ = \begin{cases} 
  \mathcal{F}_0 & \text{if } \mathcal{I}_z t = 0 \\
  \mathcal{F}_y(\mathcal{I}_z t - 1) & \text{otherwise} 
\end{cases} \]

\[ = \begin{cases} 
  \nu_z t & \text{if } t = C_z 0 \\
  \nu_y(\mathcal{C}_y(\mathcal{I}_z t - 1)) & \text{otherwise} 
\end{cases} \]

\[ z = x \text{ before } y \]

\[ C_z = \{ \min(\mathcal{H}_z n, \mathcal{H}_y n) \mid n \in \mathbb{N} \} \]

\[ = \{ t \in T : \exists n \in \mathbb{N} : t = C_z n \land n \geq \#\{ t' \in C_y \mid t' \leq t \} \lor t = \mathcal{C}_y n \land n \geq \#\{ t' \in C_z \mid t' \leq t \} \} \]

\[ = \{ t : t \in C_z \land \mathcal{I}_z t \geq \mathcal{I}_y t \lor t \in C_y \land \mathcal{I}_y t \geq \mathcal{I}_z t \} \]

\[ \nu_z t = \mathcal{F}_z(\mathcal{I}_z t) \]

\[ = \begin{cases} 
  \text{true} & \text{if } \mathcal{H}_x(\mathcal{I}_z t) \leq \mathcal{H}_y(\mathcal{I}_z t) \\
  \text{false} & \text{otherwise} 
\end{cases} \]

\[ = \begin{cases} 
  \text{true} & \text{if } C_z(\mathcal{I}_z t) \leq C_y(\mathcal{I}_y t) \\
  \text{false} & \text{otherwise} 
\end{cases} \]

\[ = \begin{cases} 
  \text{true} & \text{if } \mathcal{I}_z t \geq \mathcal{I}_y t \\
  \text{false} & \text{otherwise} 
\end{cases} \]

6 Infinite accumulation

The semantics of Table 3 cannot be implemented since
• constants continue to generate an infinite set of datons in the first macro-instant;

• the functions \( I \) and \( R \) are used in expressions such as \( \mathcal{V}_x(C_x(t)) \), thereby accessing a timestamp in a clock \( C_x \), which is in turn used to access a value in \( \mathcal{V}_x \). In general, this cannot be done without having to memorize an unbounded set of datons.

We can simplify the above semantics by using constraints on clocks, in effect restraining them so that datons are generated only when they are needed. In fact, datons may be retained so long as their relative arrival order is kept unchanged, so that the before operator yields the same results. A clock \( C \) may be transformed into a retarded clock \( C' \) if we change the function \( \mathcal{V} \) into a new function \( \mathcal{V}' \) so that:

\[
\forall n \in \mathbb{N} : \mathcal{V}'(C'_n) = \mathcal{V}(C_n)
\]

Table 4 summarizes the new semantics with constraints on clocks. Note that if clock retarding is used as needed, then the same (denotational) dataflow might have to be replicated several times into several (operational) dataflows, for different uses at different rates (on different clocks).

**Proposition 2** The replication of one (denotational) dataflow into several (operational) dataflows need only be done a finitely many times.

**Proof** We need to show how to build the dependency tree of a program and show that this process terminates.

Let \( x \) be a dataflow and let \( x^n \) denotes the instantiation of \( x \) used with rate \( n \). We write \( d(x^n) \) to extract the dataflow \( x \) from \( x^n \).

Let \( P \) be a RLUCID program, then for each \( y \in \text{out}(P) \) with builds a tree, denoted \( \mathcal{T}(y) \), defined as:

\[
\begin{align*}
\mathcal{T}(y) &= [eq(y)](1, \emptyset) \\
[z = k](n, S) &= S \cup \{ z_n = k \} \\
[z = \text{op}(x_1, \ldots, x_m)](n, S) &= \bigcup_{i=1}^{m} [eq(x_i)](n, S \cup \{ z_n = \text{op}(x_{i_1}, \ldots, x_{i_m}) \}) \\
[z = x \text{ skip } s](n, S) &= [eq(x)](n \times s, S \cup \{ z_n = x_n \text{ skip } s_n \}) \cup [eq(s)](n, S \cup \{ z_n = x_n \text{ skip } s_n \}) \\
[z = x \text{ before } y](n, S) &= [eq(x)](n, S \cup \{ z_n = x_n \text{ before } y_n \}) \cup [eq(s)](n, S \cup \{ z_n = x_n \text{ before } y_n \}) \\
[z = x \text{ fby } y](n, S) &= [eq(x)](0, S \cup \{ z_n = x_0 \text{ before } y_n \}) \cup \\
& \begin{cases} \\
\emptyset & \text{if } \exists eq(y_n) \in S \land n = 0 \\
[eq(s)](n, S \cup \{ z_n = x_0 \text{ before } y_n \}) & \text{if } \neg \exists eq(y_m) \in S, d(y_m) = d(y_n) \\
\end{cases}
\end{align*}
\]

Starting with the program outputs, we go through their definitions and attach to them a rate of use. At first this rate of use is 1, but each time we encounter an \( x \text{ skip } s \)
expression, we multiply by \( s \) the already accumulated rates of use and continue to build a subtree from there.

Recursively defined dataflows, as \( x = 0 \ fby \ x + 1 \), are permitted. But not those like \( x = 0 \ fby (x \ skip n) \) with \( n > 0 \) since they use \( x \) at an incompatible rate.

Since no other form of recursion is allowed, it follows by structural induction that this process terminates if the number of program statements is finite.

Let us note that the system’s inputs must be memorized if they are to be used at different speeds. This is a limitation of the current semantics. It can duplicate its internal dataflows but not its inputs. So memorization of the inputs could lead to unbounded memory situations.

### Table 4: Clocked semantics for RLUCID

<table>
<thead>
<tr>
<th>RLUCID operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = k )</td>
<td>( C_z \leq T_3 )</td>
</tr>
<tr>
<td>( V_z t = k )</td>
<td></td>
</tr>
<tr>
<td>( z = op(x_1, \ldots, x_m) )</td>
<td>( C_z = C_{x_1} ) with constraint ( \forall i \in 1..m : C_z = C_{x_i} )</td>
</tr>
<tr>
<td>( V_z t = op(V_{x_1} t, \ldots, V_{x_m} t) )</td>
<td></td>
</tr>
<tr>
<td>( z = x \ skip s )</td>
<td>( C_z = { t</td>
</tr>
<tr>
<td>( V_z t = V_x t )</td>
<td></td>
</tr>
<tr>
<td>( z = x \ fby y )</td>
<td>( C_z = { t</td>
</tr>
<tr>
<td>( V_z t = \begin{cases} V_x t &amp; \text{if } t = C_x 0 \ V_y t &amp; \text{otherwise} \end{cases} )</td>
<td></td>
</tr>
<tr>
<td>( z = x \ before y )</td>
<td>( C_z = { t</td>
</tr>
<tr>
<td>( V_z t = \begin{cases} \text{true} &amp; \text{if } I_z t \geq I_y t \ \text{false} &amp; \text{otherwise} \end{cases} )</td>
<td></td>
</tr>
</tbody>
</table>

Note that we still have to deal with some kind of infinity, since the semantics of the before operator makes use of \( I \) on two different dataflows \( x \) and \( y \), and the difference between \( I_z t \) and \( I_y t \) could potentially be unbounded.

**Proposition 3** The semantics presented in Table 4 is correct.

**Proof** The only non-trivial new derivations of clocks are for \( fby \) and \( skip \):

- For \( z = x \ fby y \)
  
  \[
  \begin{align*}
  &= (t = C_x 0) \lor (C_y 0 > C_x 0 \land (t \in C_y \lor \pi_r(t) = \pi_r(C_x 0) \land I_y t \geq \pi_z (t - C_x 0))) \\
  &= (t = C_x 0) \lor (C_y 0 > C_x 0 \land t \in C_y \lor C_y 0 > C_x 0 \land \pi_r(t) = \pi_r(C_x 0) \land I_y t \geq \pi_z (t - C_x 0) \land I_y t \geq \pi_z (t - C_x 0)) \\
  &= (t = C_x 0) \lor (C_y 0 > C_x 0 \land t \in C_y \lor \text{false}) \\
  &= (t = C_x 0) \lor (C_y 0 > C_x 0 \land t \in C_y)
  \end{align*}
  \]

42
7 Conclusions

Once RLUCID programs have been rewritten so that each dataflow is only used at a single rate, then it is easy to translate RLUCID into a suitable target language. In Gagné's thesis [1], there is a translation into BLIZZARD, a timestamp-based dataflow language invented by the authors.

The first problem to be solved was the presence of bursts. It turned out that this can be resolved by slightly changing RLUCID's semantics. The time domain \( T = \mathbb{R} \times \mathbb{Z} \) gives to each element of a burst a distinct timestamp, corresponding to a different micro-instant within the same macro-instant: it is thus possible to do a finer analysis of a macro-instant. In general, this time domain allows a better understanding of instantaneous interaction.

The second problem was related to the use of the same dataflows at different rates. This leads, in the worst case, to unbounded-memory RLUCID programs. It turned out that this problem can be solved using a demand-driven approach, assuming that inputs are not unsynchronized in an unbounded manner. Each denotational dataflow needs a different instantiation for each different rate of use.

Combined with the Java implementation of BLIZZARD, we now have a working implementation of RLUCID.

References

