

From abstract time to real time*

John Plaice[†] Ridha Khédri[†] René Lalement[‡]

Abstract

In computer science, the term ‘time’ is overloaded. Through a succession of languages, it is shown that there exist at least three different notions of time, each of which corresponds to a different semantic concept. First is the order of occurrence in a single stream, as in Lucid. Second is the relative order of occurrence of events in different streams, as in RLUCID. Third is the actual time that values are calculated in a given implementation; this last concept, ‘real time’, is found in RTLucid. The three different approaches to time are complementary, allowing RTLucid to be strictly more expressive than RLUCID, RLUCID to be strictly more expressive than Lucid, and Lucid to be more expressive than ISWIM.

1 Introduction

There has been much research in computer science with respect to reactive systems and real-time systems. Some of the more successful work has used temporal logic, either descriptively or prescriptively. Despite much of the success of this work, there are still many troubling questions about the nature of time. These are not esoteric, philosophical questions, but grounded, technical problems in computer science.

In fact, the term ‘time’ can have at least three meanings. The purpose of this discussion is to show that in fact these meanings correspond to different semantic concepts, and that they can all be integrated into a single programming language. In fact, they can be introduced one at a time, thereby giving a succession of languages.

*Presented to the Sixth International Symposium on Lucid and Intensional Programming, Université Laval, Canada, 26–27 April 1993. This research was partially funded by the National Sciences and Engineering Research Council (Canada) and the Fonds pour la Formation de Chercheurs et l’Aide à la Recherche (Québec).

[†]{John Plaice, Ridha Khédri}, Département d’informatique, Université Laval, Ste-Foy, Québec, Canada G1K 7P4. e-mail: {plaice,khedri}@ift.ulaval.ca .

[‡]René Lalement, CERMICS, École Nationale des Ponts et Chaussées, La Courtine, F-93167 Noisy-le-Grand, France. e-mail: rl@enpc.fr .

$E ::=$	k	
	$ $	id
	$ $	$op(E, \dots, E)$
	$ $	$id(E, \dots, E)$
	$ $	$if\ E\ then\ E\ else\ E$
	$ $	$E\ where\ Q$
	$ $	$next\ E$
	$ $	$E\ fby\ E$
	$ $	$E\ precedes\ E$
	$ $	$E\ before\ E$
		} ISWIM
		} Lucid
		} RLUCID
		} RTLUCID
$Q ::=$	$id = E$	
	$ $	$id(id, \dots, id) = E$

Figure 1: Abstract syntax for RTLUCID.

The discussion below will only consider functional languages, although many of the results should be translatable to other programming paradigms, such as logic or imperative programming.

The outline of this paper is as follows. A brief reminder of functional programming is given, using Landin's ISWIM [3]. Streams are introduced by adding the Lucid operators `next` and `fby` [8]. Reactive systems become programmable when decisions can be made according to the order of arrival of input, using RLUCID's `precedes` [5, 6]. Finally, hard real-time constraints can be expressed using RTLUCID's `before` [6]. Since each language includes its predecessor, to simplify the presentation, only the syntax of RTLUCID is given, as in Figure 1.

It is assumed that a basic algebra of constants and data operators exists. The constants are designated by k and the data operators by `op`. In general, these basic operators will include the standard arithmetic and boolean operators.

2 Functional programming

The language to which notions of time will be added is Landin's ISWIM [3]. This language is normally considered to be the base of any functional programming language. ISWIM programs are simply expressions where the local variables and functions are defined using sets of equations. The order of occurrence of equations is of no importance.

The advantages of functional programming have been vaunted in detail many times; for example, reference [2] contains a full discussion. However, certain details that might be remembered are that 'pure' functional languages, such as the one presented here, are referentially transparent. In a given scope, a

$$\begin{array}{c}
\overline{\rho \vdash k : k} \\
\\
\overline{\rho \vdash id : \rho(id)} \\
\\
\frac{\rho \vdash E_i : e_i \quad i=1, \dots, p}{\rho \vdash op(E_1, \dots, E_p) : op(e_1, \dots, e_p)} \\
\\
\frac{\rho \vdash E_i : e_i \quad i=1, \dots, p}{\rho \vdash id(E_1, \dots, E_p) : \rho(id)(e_1, \dots, e_p)} \\
\\
\frac{\rho \vdash E_1 : true \quad \rho \vdash E_2 : e_2}{\rho \vdash if \ E_1 \ then \ E_2 \ else \ E_3 : e_2} \\
\\
\frac{\rho \vdash E_1 : false \quad \rho \vdash E_3 : e_3}{\rho \vdash if \ E_1 \ then \ E_2 \ else \ E_3 : e_3} \\
\\
\frac{\rho \vdash E_2 : e_2 \quad \rho[id \leftarrow e_2] \vdash E_1 : e_1}{\rho \vdash E_1 \ where \ id = E_2 : e_1} \\
\\
\frac{\rho \setminus \{id_1, \dots, id_p\} \vdash E_2 : e_2 \quad \rho[id \leftarrow \lambda(id_1, \dots, id_p).e_2] \vdash E_1 : e_1}{\rho \vdash E_1 \ where \ id(id_1, \dots, id_p) = E_2 : e_1}
\end{array}$$

Figure 2: The semantics of ISWIM

variable will have only one meaning, in all contexts, and it can be replaced by its defining expression wherever it occurs in an expression.

ISWIM's operational semantics are given in Figure 2. The judgment

$$\rho \vdash E : e$$

means that in environment ρ , the expression E has value e . The environment maps identifiers to values.

As an example, the following program generates a list, every element of which is the double of the corresponding element in the original list:

```

map double [1,2,3]
where double x = x*2
      map f = m
      m x = if x=nil
            then nil
            else (f(hd x)) :: (m(tl x))
end

```

Note that the function `map` is a *higher-order* function: its first argument must be a function.

To avoid using closures in the semantics, liberties have been taken. It is supposed that if there is no value for identifier id in ρ , then $id(\rho)$ just gives id . Thus, in the function declaration, the evaluation of E_2 will not give a value but, rather, a new expression where the only remaining identifiers are the formal parameters. This decision was taken because Lucid implementations do not use closures but, rather, a tagging mechanism which simulates the occurrences in a calling tree. See reference [7] for more details.

3 Adding streams

Streams are natural programming tools that are found in many areas of computer science. Pipes in UNIX allow one to connect the output of one program to the input of the next program.

In their simplest form, Lucid streams are simply functions from natural number contexts to the domain of values. Streams can be manipulated with two operators. The `next` operator generates a new stream where the first element has been removed. The `fbv` operator creates a new stream by putting the first element of the first stream before the second stream.

The operational semantics of Lucid are given in Figure 3. The judgment

$$\rho, C \vdash E : e$$

means that in environment ρ and in natural context C , the expression E has value e . The environment maps identifier-context pairs to values. Other versions of Lucid (see reference [4]) allow contexts to be arbitrary tuples of integers, or even arbitrary values.

The simplest derived function is the `first` operator:

```
first x = x fby first x
```

This function generates a constant stream from the first value in a stream.

Streams can be considered to be a first approximation to time. If a system has one input channel and one output channel, then the order in which occurs the input can 'define' time. For example, the running total of the input can be written as follows:

```
out where out = in fby (out + next in)
```

Clearly, the $n + 1$ -th occurrence of `out` occurs after the n -th occurrence of `out`.

However, it is possible to write Lucid programs which are not pipeline dataflow. For example, the pathological program

```
f n where f x = if x=0 then 5 else next (f x)
```

gives, for each index i , the value 5 should there exist an index $j > i$ such that the j -th value of `x` is 0. However, an arbitrary amount of computation may

$$\begin{array}{c}
\overline{\rho, C \vdash k : k} \\
\\
\overline{\rho, C \vdash id : \rho(id, C)} \\
\\
\frac{\rho, C \vdash E_i : e_i \quad i=1, \dots, p}{\rho, C \vdash \text{op}(E_1, \dots, E_p) : \text{op}(e_1, \dots, e_p)} \\
\\
\frac{\rho, C \vdash E_i : e_i \quad i=1, \dots, p}{\rho, C \vdash id(E_1, \dots, E_p) : \rho(id, C)(e_1, \dots, e_p)} \\
\\
\frac{\rho, C \vdash E_1 : \text{true} \quad \rho, C \vdash E_2 : e_2}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : e_2} \\
\\
\frac{\rho, C \vdash E_1 : \text{false} \quad \rho, C \vdash E_3 : e_3}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : e_3} \\
\\
\frac{\rho, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow e_2] \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id = E_2 : e_1} \\
\\
\frac{\rho \setminus \{id_1, \dots, id_p\}, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow \lambda(id_1, \dots, id_p). e_2], C \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id(id_1, \dots, id_p) = E_2 : e_1} \\
\\
\frac{\rho, C+1 \vdash E : e}{\rho, C \vdash \text{next } E : e} \\
\\
\frac{\rho, 0 \vdash E_1 : e}{\rho, 0 \vdash E_1 \text{ fby } E_2 : e} \\
\\
\frac{\rho, C-1 \vdash E_2 : e}{\rho, C \vdash E_1 \text{ fby } E_2 : e}
\end{array}$$

Figure 3: The semantics of Lucid

need to be done before the value 5 is generated. There is therefore *no* notion resembling real time.

In addition, should two streams be added together, as in $A+B$, there is nothing to prevent A and B from being completely out of sync. At some point, A might be much faster than B , but at another point, B might be much faster than A . Lucid is incapable of distinguishing ‘slow’ streams from ‘fast’ streams.

4 Reactive programming

Reactive programs are programs that interact *continually* with their environments. They are found everywhere, and include just about anything from elec-

tronic gadgets to aircraft control and industrial control. The interaction in these programs typically involves a feedback process: the output of the reactive system will affect the environment, which will in turn affect the reactive system.

For example, consider a thermostat. It receives as input the temperature of the room and, as output, generates a signal to a furnace to turn on or to turn off. When the input temperature goes above a certain threshold, the thermostat will signal the furnace to turn itself off; the room temperature will ultimately decrease and the thermostat will then signal the furnace to turn itself back on.

This sort of problem can still be handled in the Lucid framework, as there is but one input, the temperature. However, most reactive systems have several inputs, which do not all arrive at the same time, or even with the same periodicity. Consider, for example, a simple wristwatch. Its inputs are the buttons on the watch and the signal from the quartz; its output is the display. The correct behavior of this watch requires that the program can determine the order of arrival of the input events. Pressing button A and then button B is certainly not the same thing as pressing button B and then button A.

If determining the order of arrival of input events is important, the simplest thing to do is add a primitive, called *precedes*, to Lucid. The Boolean stream *A* *precedes* *B* states which of the corresponding datons of *A* and *B* occurred first.

The resulting language is RLUCID [5]. Its semantics are presented in Figure 4. The judgment

$$\rho, C \vdash E : v, t$$

means that in environment ρ and in natural context C , the expression E has value v and occurs at time t . The possible values for time are all in \overline{R}_0^+ . To simplify the presentation, as well as to show the similarities with the other semantics, e refers to the pair v, t .

The semantics of RLUCID assumes the synchronous hypothesis, i.e. that computations take no time (see Berry [1] for a fuller discussion). It is therefore possible to have a counter that counts, *exactly* when an event *A* occurs, the number of occurrences of *A*:

```
#1{count,A} where count = 1 fby count + 1
```

where the $\{\dots\}$ data operator creates tuples and the #1 data operator selects the first element in a tuple.

An often used derived operator is the wait operator, which is defined as follows:

```
wait X1:E1 X2:E2 ... Xn:En end =
if X1 precedes X2 and X1 precedes ... Xn then E1
else if X2 precedes X3 and X2 precedes ... Xn then E2
...
else En
```


Streams can also be synchronized with any other stream, through the use of the `last` operator. The program

```
last(0, #1{count,A}, B)
where last(X,Y,Z) =
    wait Y : last(Y, next Y, Z)
        Z : X fby last(X, Y, next Z)
    end
```

generates, upon every occurrence of `B`, the number of `A`'s that have occurred up to and including that moment.

Using `wait`, two streams can be merged *deterministically*, according to the order of arrival of their input:

```
MergeLeft(X,Y) =
wait
    X: X fby MergeLeft(next X, Y)
    Y: Y fby MergeLeft(X, next Y)
end
```

5 Real-time programming

The synchronous hypothesis allows programmers to refer to timed activities in an implementation-independent manner. However, computations do take time, and there are clearly situations where if the computations are not done at the right moment, then grave situations can occur.

Real-time programs can be considered to be reactive programs which are subject to hard real-time constraints. These constraints can take on several forms, typically required response times or sampling frequency of input.

What is normally done with synchronous languages is to determine the minimum interval between two input events and then to ensure that in any implementation, the maximum response time is less than that minimum interval. However, none of this reasoning can be done in `RLUCID` itself.

To be able to verify 'real' real-time properties of a program, the actual physical implementation must be taken into account. A given program may well run correctly on a fast machine, but not on a slower machine.

The `RTLUCID` language assumes that each event has *two* times: the logical time, which corresponds to the idealized situation where computers are infinitely fast, as in `RLUCID`, and a real time, which corresponds to the actual moment that the value is computed. Clearly, the real time of an event never occurs before the logical time. To ensure program portability, the real time is not directly accessible from a program. A timing constraint limits the allowed delay between the logical time and the real time.

$$\begin{array}{c}
\overline{\rho, C \vdash k : k, 0} \\
\\
\overline{\rho, C \vdash id : \rho(id, C)} \\
\\
\frac{\rho, C \vdash E_i : v_i, t_i \quad i=1, \dots, p}{\rho, C \vdash \text{op}(E_1, \dots, E_p) : \text{op}(v_1, \dots, v_p), \max t_i} \\
\\
\frac{\rho, C \vdash E_i : e_i \quad i=1, \dots, p}{\rho, C \vdash id(E_1, \dots, E_p) : \rho(id, C)(e_1, \dots, e_p)} \\
\\
\frac{\rho, C \vdash E_1 : \text{true}, t_1 \quad \rho, C \vdash E_2 : v_2, t_2}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : v_2, \max(t_1, t_2)} \\
\\
\frac{\rho, C \vdash E_1 : \text{false}, t_1 \quad \rho, C \vdash E_3 : v_3, t_3}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : v_3, \max(t_1, t_3)} \\
\\
\frac{\rho, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow e_2] \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id = E_2 : e_1} \\
\\
\frac{\rho \setminus \{id_1, \dots, id_p\}, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow \lambda(id_1, \dots, id_p). e_2], C \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id(id_1, \dots, id_p) = E_2 : e_1} \\
\\
\frac{\rho, C+1 \vdash E : e}{\rho, C \vdash \text{next } E : e} \\
\\
\frac{\rho, 0 \vdash E_1 : e}{\rho, 0 \vdash E_1 \text{ fby } E_2 : e} \\
\\
\frac{\rho, C-1 \vdash E_2 : e}{\rho, C \vdash E_1 \text{ fby } E_2 : e} \\
\\
\frac{\rho, C \vdash E_i : v_i, t_i \quad i=1, 2}{\rho, C \vdash E_1 \text{ precedes } E_2 : t_1 \leq t_2, \min(t_1, t_2)}
\end{array}$$

Figure 4: The semantics of RLUCID

The semantics of RTLUCID are given in Figure 5. The judgment

$$\rho, C \vdash E : v, t, r$$

means that in environment ρ and natural context C , the expression E evaluates to value v at logical time t and at physical time r . The physical time will never be earlier than the logical time.

The new primitive operation is called **before**. The expression **A before B** simply means **A**, with the constraint that **A** must physically occur before **B** logically occurs.

$$\begin{array}{c}
\frac{}{\rho, C \vdash k : k, 0, 0} \\
\\
\frac{}{\rho, C \vdash id : \rho(id, C)} \\
\\
\frac{\rho, C \vdash E_i : v_i, t_i, r_i \quad r \geq \max r_i \quad i=1, \dots, p}{\rho, C \vdash \text{op}(E_1, \dots, E_p) : \text{op}(v_1, \dots, v_p), \max t_i, r} \\
\\
\frac{\rho, C \vdash E_i : e_i \quad i=1, \dots, p}{\rho, C \vdash id(E_1, \dots, E_p) : \rho(id, C)(e_1, \dots, e_p)} \\
\\
\frac{\rho, C \vdash E_1 : \text{true}, t_1, r_1 \quad \rho, C \vdash E_2 : v_2, t_2, r_2 \quad r \geq \max(r_1, r_2)}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : v_2, \max(t_1, t_2), r} \\
\\
\frac{\rho, C \vdash E_1 : \text{false}, t_1 \quad \rho, C \vdash E_3 : v_3, t_3 \quad r \geq \max(r_1, r_3)}{\rho, C \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : v_3, \max(t_1, t_3), r} \\
\\
\frac{\rho, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow e_2] \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id = E_2 : e_1} \\
\\
\frac{\rho \setminus \{id_1, \dots, id_p\}, C \vdash E_2 : e_2 \quad \rho[(id, C) \leftarrow \lambda(id_1, \dots, id_p).e_2], C \vdash E_1 : e_1}{\rho, C \vdash E_1 \text{ where } id(id_1, \dots, id_p) = E_2 : e_1} \\
\\
\frac{\rho, C+1 \vdash E : e}{\rho, C \vdash \text{next } E : e} \\
\\
\frac{\rho, 0 \vdash E_1 : e}{\rho, 0 \vdash E_1 \text{ fby } E_2 : e} \\
\\
\frac{\rho, C-1 \vdash E_2 : e}{\rho, C \vdash E_1 \text{ fby } E_2 : e} \\
\\
\frac{\rho, C \vdash E_i : v_i, t_i, r_i \quad i=1, 2 \quad M = \text{if } t_1 \leq t_2 \text{ then } 1 \text{ else } 2}{\rho, C \vdash E_1 \text{ precedes } E_2 : t_1 \leq t_2, t_M, r_M} \\
\\
\frac{\rho, C \vdash E_i : v_i, t_i, r_i \quad i=1, 2 \quad r_1 \leq t_2}{\rho, C \vdash E_1 \text{ before } E_2 : v_1, t_1, r_1}
\end{array}$$

Figure 5: The semantics of RTLUCID

Suppose that a program has two input: a value I and a second S. The output is the same as I, but it must be output before two S have appeared. The result would then be

```

I before alternate
where alternate = S fby next next S

```

It is clear the `before` operator constrains the code that a compiler can generate. To ensure that ‘real’ hard-time constraints can be stated, a particular input event, such as a second, must be singled out so that a link can be made with the (abstract) program and the (real) implementation. Note that this is the first time in the entire presentation that a particular event is singled out; in the rest of the presentation, time is multiform.

6 Conclusions and discussion

Through the succession of languages (ISWIM, Lucid, RLUCID, RTLUCID), different concepts of time were shown to correspond to different semantics entities; all of which can be used in the same programming language.

The discussion in this paper dealt only with time, as we went from abstract time (Lucid) to logical time (RLUCID) to real time (RTLUCID). Lucid has been extended several times to allow intensional operators defining passage from any context to another, not just integer contexts. In particular, integer-tuple contexts can be used to have a notion of abstract time and space. We speculate that there are probably equivalent notions of ‘logical space’ and ‘real space’, which would give interesting insight into problems related to parallelism.

References

- [1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 1992.
- [2] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.
- [3] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
- [4] J. Plaice. ML-Lucid, an intensional functional language. In *5th International Symposium on Lucid and Intensional Programming*, pages 54–62, San Francisco (CA), USA, 1992. Workshop of the *IEEE 1992 International Conference on Computer Languages*.
- [5] J. Plaice. RLucid, a general real-time dataflow language. In *School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, the Netherlands, 1992.
- [6] J. Plaice, R. Khédri, and R. Lalement. Adding timing constraints to a real-time dataflow language. In preparation, 1993.

-
- [7] P. Rondogiannis and W. W. Wadge. A dataflow implementation technique for lazy typed functional languages. In *6th International Symposium on Lucid and Intensional Programming*, Québec, Canada, 1993.
 - [8] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.