From Lucid to TransLucid: Iteration, Dataflow, Intensional and Cartesian Programming

John Plaice Blanca Mancilla
Gabriel Ditu
School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{plaice,mancilla,gabd}@cse.unsw.edu.au

2008

Abstract

We present the development of the Lucid language from the Original Lucid of the mid-1970s to the TransLucid of today. Each successive version of the language has been a generalisation of previous languages, but with a further understanding of the problems at hand.

The Original Lucid (1976), originally designed for purposes of formal verification, was used to formalise the iteration in `while`-loop programs. The pLucid language (1982) was used to describe dataflow networks. Indexical Lucid (1987) was introduced for intensional programming, in which the semantics of a variable was understood as a function from a universe of possible worlds to ordinary values. With TransLucid, and the use of contexts as first-class values, programming can be understood in a Cartesian framework.

1 Introduction

This paper presents the development of the Lucid programming language, from 1974 to the present, with a particular focus on the seminal ideas of William (Bill) Wadge. These include the use of infinite data structures, the importance of iteration, the use of multidimensionality, the rise of intensional programming, the importance of demand-driven computation, eduction as a computational model, and the necessity of replacing the von Neumann architecture with more evolved computational machines.

Many of these ideas were explicit right from the beginning, others implicit, while still others were developed through a series of implementations and expansions of Lucid. Finally, some had to wait until the design and implementation of the most recent version, TransLucid, the result of many years of research.

The relevance of Wadge's ideas is increasingly timely. Let us consider the very last topic, with respect to computer architecture. Since 2003, single processor speedup has not kept pace with Moore's Law. The law remains valid, with chip transistor density doubling approximately every 24 months [1, 2]. However, a corresponding annual 52% single processor speedup, starting in 1986, ceased to be true in 2003, dropping to 20% [3]. To compensate, vendors have moved towards multicore processors, and researchers are talking about manycore processors, each capable of managing very large numbers of threads.

The problem with these hardware developments is that the mainstream programming languages are not well suited to these new architectures. It is difficult to transform a program written in C or some other imperative language to take advantage of parallelism available in a new architecture, let alone to take advantage of varying amounts and forms of parallelism, as successive architectures are brought onto the market every few months.

This kind of scenario was predicted by Wadge and Ed Ashcroft, in the introduction to their 1985 book, *Lucid, the Dataflow Programming Language* [4]. In the introduction, they spoofed the different researchers working in programming language design, semantics and implementation, categorising them into Cowboys, Boffins, Wizards, Preachers and Handymen, according to their various preferences. The point of this tongue-in-cheek description was not meant to insult anyone — although a few feathers did get rustled — but, rather, to point out that most of these approaches implicitly assumed that the von Neumann architecture was going to remain with us forever, and that "Researchers who try to avoid the fundamental controversies in their subject risk seeing their lifetime's work prove in the end to be irrelevant."

The key insight of Bill Wadge is that significant advances in programming language design and semantics cannot be made independently of the underlying computational models. Efficiency is not a mere implementation detail, allowing a programmer to simply provide some unexecutable specification. As a result, existing programming practices, although possibly limited, cannot be ignored. Crucially, the most important practice is that computers iterate, i.e., they are good at doing things over and over again, and do not recurse.

Focusing on efficiency, one must be careful to analyse the underlying assumptions that are being made in any given computational model. For example, one of the criticisms made towards Lucid and its implementations is that the demand-driven implementations are inefficient. Although it is true that demand-driven mechanisms do carry an overhead, it is rarely acknowledged that *any* computational model using memory is itself demand-driven. The infamous Von Neumann memory bottleneck is a bottleneck *precisely* because the memory is accessed in a demand-driven manner: one gives the index of a cell and makes a demand for the value therein; depending on the structure of the memory hierarchy, this demand will be treated in different ways.

The point, therefore, is not whether one should use demand-driven or data-driven mechanisms but, rather, exactly what kind of demand-driven mechanisms are most suitable? Or, given appropriate architectures, to what extent can demand-driven mechanisms be translated into data-driven systems? The first question is completely compatible with the current trends in computer architecture, with multiple cores each running multiple threads; if a demand in one thread blocks, it may well be the case that a previously blocked demand in another thread has been resolved. The second question deals with the development of innovative architectures.

In all of the variants of Lucid, infinite data structures are defined using mutually recursive systems of equations. The recursion is uniquely for definitional purposes, it is not a computational phenomenon. One iterates towards a result.

In this article, we examine the successive versions of Lucid and examine, through the use of common examples, the different interpretations and ideas associated with these different versions. The general tendency is to move from sequential forms of computing to indexical forms of computing, leading ultimately to Cartesian programming with TransLucid.

2 Iteration: Original Lucid

The Lucid language was first conceived in 1974 by Bill Wadge and Ed Ashcroft when the two were academics at the University of Waterloo. Two major papers were published, one in 1976 in *SIAM Journal of Computing* [5], the other in 1977 in *Communications of the ACM* [6]. As we shall see below, in the (Original) Lucid they presented in these papers, Wadge and Ashcroft introduced infinite data structures, iteration and multidimensionality as means to formally describe computation.

At the time, discussions around structured programming were standard. In 1968, Edsger Dijkstra had penned his famous "Go to Statement Considered Harmful" article [7], making the computer science community realise that programming was not simply something that had to be done but, rather, something that could be done with elegance and grace. One of the main ideas, associated with Tony Hoare, was that a block should have a single entry point and a single exit point. These ideas were well presented in the books *Structured Programming* by Dahl, Dijkstra and Hoare [8], and *A Discipline of Programming* by Dijkstra [9]. In the latter, Dijkstra's presentation

led naturally to the vision that computer programs could be formally verified if they had a proper mathematical description.

It is in this context that Wadge and Ashcroft began the work leading to Lucid [10]. Wadge's PhD work at Berkeley was in descriptive set theory, leading to the development of *Wadge games*, described in [11]. Given this experience, he was habituated in thinking in terms of infinite sets.

Wadge was examining programs such as the following one:

```
I = 0
J = 0
while (...)
   J = J + 2*I + 1
   I = I + 1
   PRINT J
end while
```

which gives the output:

```
1 4 9 16 ...
```

In this program, it is easy to understand and to prove that after the assignment:

```
J = J + 2*I + 1
```

that $\texttt{J} = (\texttt{I} + 1)^2$ and that after the assignment:

```
I = I + 1
```

that $\texttt{J} = \texttt{I}^2$. However, in programs of the form:

```
while (...)
   J = ...
   P = ...
   J = ...
   P = ...
end while
```

it is much more difficult to understand the meaning of a program, because of the reassignments of `J` and `P`. This study led to Wadge's insight of "(Re)Assignment Considered Harmful". By letting variables define infinite sequences, he could rewrite the above program as:

```
first I = 0;
next  I = I + 1;
first J = 0;
next  J = J + 2*I + 1;
```

Hence:

$$\begin{aligned} \texttt{I} &= \langle 0, 1, 2, 3, \ldots \rangle \\ \texttt{next I} &= \langle 1, 2, 3, 4, \ldots \rangle \\ \texttt{J} &= \langle 0, 1, 4, 9, \ldots \rangle \\ \texttt{next J} &= \langle 1, 4, 9, 16, \ldots \rangle \end{aligned}$$

By introducing the operators `first` and `next`, one could define the entire history of a variable using just two lines. Formally, if the variables `X` and `Y` are defined by:

$$\texttt{X} = \langle x_0, x_1, x_2, \ldots, x_i, \ldots \rangle \quad (1)$$

$$\texttt{Y} = \langle y_0, y_1, y_2, \ldots, y_i, \ldots \rangle \quad (2)$$

then:

$$\texttt{first X} \;=\; \langle x_0, x_0, \ldots, x_0, \ldots\rangle \tag{3}$$
$$\texttt{next X} \;=\; \langle x_1, x_2, \ldots, x_{i+1}, \ldots\rangle \tag{4}$$

When a constant `C` appears in a program, it corresponds to the infinite sequence:

$$\texttt{C} \;=\; \langle c, c, c, \ldots, c, \ldots\rangle \tag{5}$$

Finally, when a data operator `op` appears in a program, it is applied pointwise to its arguments:

$$\texttt{X op Y} \;=\; \langle x_0 \text{ op } y_0,\ x_1 \text{ op } y_1,\ \ldots,\ x_i \text{ op } y_i,\ \ldots\rangle \tag{6}$$

Finally, to end an iteration, the `asa` operator is used:

$$\texttt{X asa Y} \;=\; \langle x_j, x_j, x_j, \ldots, x_j, \ldots\rangle, \qquad y_j \wedge \forall(i<j)\,\neg\, y_i \tag{7}$$

Here, it is assumed that the values of sequence `Y` must be convertible to Boolean values.

As can be seen from the above discussion, Wadge and Ashcroft privileged *iteration*. They did not redefine what a computer was doing but, rather, presented a formal framework in which one could state exactly what a computer was doing. The language they had introduced had a perfectly clear mathematical semantics, yet represented programming as it really existed.

To handle nested loops, the "time" index was extended to include "multidimensional time". A variable `F` of n dimensions, instead of being a mapping from $\mathbb{N}$ to values, becomes a mapping from $\mathbb{N}^n$ to values. The notation

$$\texttt{F}_{t_1\, t_2 \cdots t_n}$$

denotes the element where the outermost time dimension has value t_n, and the innermost time dimension has value t_1.

The `latest` operator was used to freeze the value of the current stream representing the outer loop, while reaching into the inner loop to manipulate the relevant stream, so that one could come back to the outer loop with the result. Here are the definitions.

$$(\texttt{first F})_{t_1\, t_2 \cdots t_n} \;=\; \texttt{F}_{0\, t_2 \cdots t_n} \tag{8}$$
$$(\texttt{next F})_{t_1\, t_2 \cdots t_n} \;=\; \texttt{F}_{(t_1+1)\, t_2 \cdots t_n} \tag{9}$$
$$(\texttt{F asa G})_{t_1\, t_2 \cdots t_n} \;=\; \texttt{F}_{j\, t_2 \cdots t_n}, \qquad \texttt{G}_{j\, t_2 \cdots t_n} \wedge \forall(i<j)\neg\texttt{G}_{i\, t_2 \cdots t_n} \tag{10}$$
$$(\texttt{latest F})_{t_0\, t_1\, t_2 \cdots t_n} \;=\; \texttt{F}_{t_1 t_2 \cdots t_n} \tag{11}$$

Below is an example of the use of two time dimensions, with the `latest` operator being used to access the values from within the nested loop. The program determines if `n`, the first entry in the `input`, is prime:

```
n = first input;
first i = 2;
  first j = latest i * latest i;
  next j = j + latest i;
  latest idivn = (j eq latest n) asa (j >= latest n);
next i = i+1;
output = (not idivn) asa (idivn or i*i >= n);
```

So we have that:

$$\texttt{i} \;=\; \langle 2, 3, 4, 5, \ldots\rangle$$
$$\texttt{j} \;=\; \big\langle \langle 4, 6, 8, 10, \ldots\rangle, \langle 9, 12, 15, 18, \ldots\rangle, \ldots, \langle i_k^2, i_k^2 + i_k, i_k^2 + 2i_k, \ldots\rangle, \ldots\big\rangle$$

and so on.

When Bill Wadge moved to the University of Warwick in the UK, he met David May, who suggested that the Lucid streams could be understood using dataflow networks, and that the `first` and `next` operators could be combined using a binary operator called `fby` ("followed by"):

$$(\texttt{F fby G})_{t_0\, t_1\, t_2 \cdots} = \begin{cases} \texttt{F}_{0\, t_1\, t_2 \cdots}, & t_0 = 0 \\ \texttt{G}_{(t_0-1)\, t_1\, t_2 \cdots}, & t_0 > 0 \end{cases} \tag{12}$$

The above example then becomes:

```
n = first input;
i = 2 fby i+1;
  j = (latest i * latest i) fby (j + latest i);
  latest idivn = (j eq latest n) asa (j >= latest n);
next i = i+1;
output = (not idivn) asa (idivn or i*i >= n);
```

With the repeated use of `latest`, Original Lucid variables could in theory become infinite entities of arbitrary dimensionality. Although much of the discussion around the early versions of Lucid, both here and elsewhere, has focused on Lucid variables as sequences, effectively privileging an implicit dimension called "time", Lucid variables have always been multidimensional. However, with the initial set of primitives, only one dimension could be manipulated at a time, and the elements of a sequence were seen to be evaluated in order.

The "Extensions" section of [12] included a clear research agenda on user-defined functions, recursive functions, non-pointwise functions, the `whenever` operator, and so on. It also drew an analogy between Lucid's assumed dataflow execution model and the dataflow networks of Kahn and MacQueen [13, 14], both at the semantic and the operational levels. As we can see, the initial Lucid papers contained far more than was apparent on the surface.

3 Lucid, the Dataflow Programming Language

The pLucid language is the version of Lucid presented in *Lucid, the Dataflow Programming Language*, by Wadge and Ashcroft [4]. In pLucid, the implicit nesting of iteration by indentation is replaced by the use of `where` clauses, corresponding to the `whererec` of ISWIM [15]. In addition, new operators and extra syntax for structuring programs are added.

Unlike in the original ISWIM, pLucid is a first-order language, disallowing higher-order functions. pLucid became the "Dataflow Programming Language" because it operates on infinite streams, and dataflow streams could now be naturally expressed. For the authors, these sequences were meant to be "histories of dynamic activity".

With pLucid's focus on dataflow, it became possible to recursively define filters over streams, using the primitive operators `first`, `next` and `fby`. We give below the definitions for the predefined operators `wvr`, `upon` and `asa`.

The `wvr` operator accepts two streams and outputs the value of the first stream whenever the second one is true. In other words, certain values of the first stream are suppressed, depending on the values of the second stream:

```
X wvr Y = if first Y
          then X fby next X wvr next Y
          else next X wvr next Y
          fi;
```

Hence if:

$$\texttt{Y} = \langle \texttt{false}, \texttt{false}, \texttt{true}, \texttt{true}, \texttt{false}, \texttt{true}, \ldots \rangle \tag{13}$$

then:

$$\texttt{X wvr Y} = \langle x_2, x_3, x_5, \ldots \rangle \tag{14}$$

The `upon` operator accepts two streams and continually outputs the first value of the first stream until the second stream is next true. The next value of the first stream is then accepted and the process begins all over again. In other words, the first stream is *advanced upon* the second stream taking true values:

```
X upon Y = X fby if first Y then next X upon next Y
                 else X upon next Y
                 fi;
```

Hence, if `Y` is as defined in Equation (13) then:

$$\texttt{X upon Y} \quad = \quad \langle x_0, x_0, x_0, x_1, x_2, x_2, x_3, \ldots \rangle \tag{15}$$

The `asa` operator, described in §2, returns the value of the first stream that corresponds to the first true value in the second stream. In other words, `asa` is capable of selecting a single value from a stream. It can be used for simulating the halting clause of a loop or for terminating a program once it has computed the desired result.

```
X asa Y = first (X wvr Y)
```

Hence, if `Y` is as defined in Equation (13) then:

$$\texttt{X asa Y} \quad = \quad \langle x_2, x_2, x_2, \ldots \rangle \tag{16}$$

Like in the Original Lucid, programs in pLucid work with a sort of multidimensional time, but the dimensions are not explicit. However, the `latest` operator is replaced with the `is current` operator to freeze values for iteration in the next dimension. The example from the previous section becomes:

```
not idivn asa idivn or i * i >= N
where
  N is current n;
  i = 2 fby i + 1;
  idivn = j eq N asa j >= N;
  where
    I is current i;
    j = I * I fby j + I;
  end where;
end where;
```

Wadge and Ashcroft do note that a similar effect to using `is current` could be achieved by adding extra dimensions [4, p. 106].

With the recursive definitions, it becomes possible to use `next` so that the order of calculations does not correspond to the stream order. For example, given the definition:

```
howfar
where
  howfar = if X == 0 then 0 else 1 + next howfar;
end where;
```

and variable `X`:

$$\texttt{X} \quad = \quad \langle 1, 6, 4, 0, 9, 2, 1, 4, 3, 0, 6, \ldots \rangle$$

then here is `howfar`:

$$\texttt{howfar} \quad = \quad \langle 3, 2, 1, 0, 5, 4, 3, 2, 1, 0, \ldots \rangle$$

We will see in the next section how the implementation moves from simple iteration to *eduction*.

4 Implementing Lucid: Eduction

Interpreters were written for the Original Lucid by Tom Cargill and David May. Despite the intuition that they were dealing with dataflow streams, they in fact implemented a demand-driven approach. A request is made for a (*variable*, *tag*) pair. Should the evaluation of this variable at that tag require the evaluation of other (*variable*, *tag*) pairs, then requests are made for these too. This technique, known as *eduction*,[1] has become standard in interpreters of all versions of Lucid.

The use of eduction is crucial to deal with two major problems. First, out-of-order computation may be needed, to access streams at any point. Second, some of the operators acting like filters need to discard some of their input. With eduction, there are no wasted calculations.

The original interpreters were replaced by a more refined version developed by Calvin Ostrum [16]. However one problem still remained, and this was that the interpreter suffered from poor performance, since certain values continually needed to be recalculated. The solution was the implementation of a *warehouse* to cache calculated values and accelerate future computations.

The housekeeping of the warehouse posed its own problems, as each tagged value had to be unique. And because of tagging, the size of the warehouse could increase quite rapidly. Tony Faustini extended Ostrum's interpreter and implemented a warehouse with a garbage collector [4] as follows: values in the warehouse have a retirement age; they grow older after successive garbage collections and they are deleted if not accessed by the time they reach their retirement age. The retirement threshold is determined dynamically as the warehouse is used. Although efficiency is not guaranteed, the scheme has been proven in practice. It is important to note that the performance of the warehouse affects the runtime performance of the program but not its correctness.

From this implementation it was clear that there was a tension between the denotational semantics of infinite sequences and the operational semantics of iteration: the objects being implemented were not pipeline data, as described in the semantics. Rather, they were objects being accessed randomly. Data could be added to an object for any given tag, and any given tag could be reached at any time. The language could also handle several of these objects at the same time. Basically the semantics and the implementation did not coincide. And they did not do so, because they could not talk of multidimensionality, even though many of the tools had already been developed.

Two choices were possible. First was to restrict Lucid so that it could become a pure dataflow language; this choice was taken with the design of LUSTRE, described in §5. Second was to find a better model for understanding Lucid; this choice was taken with the introduction of *intensional programming*, described in §7.

In fact, the two choices can be distinguished from a denotational point of view. In Lucid, for a one-dimensional stream, the order on streams is the *Scott* order, in which, for example:

$$\langle \bot, 1, \bot, 3, \bot, \bot, \bot, \ldots \rangle \quad \sqsubseteq_S \quad \langle \bot, 1, \bot, 3, \bot, 5, \bot, \ldots \rangle$$

Ordinary dataflow, on the other hand, uses the *prefix* order, for example:

$$\langle 0, 1, 2, 3 \rangle \quad \sqsubseteq \quad \langle 0, 1, 2, 3, 4, 5 \rangle$$

5 Synchronous Programming: LUSTRE

LUSTRE (Synchronous Real-Time Lucid) [17, 18] is a simplification of the Lucid language specifically designed for the programming of reactive systems in automatic control. LUSTRE was designed by Paul Caspi and Nicolas Halbwachs, and the first compiler and semantics were written by author Plaice.

LUSTRE uses the *synchronous* approach to programming such systems, in which it is assumed that the reaction to an input event always takes less time than the minimum delay between two successive input events. As a result, it can be assumed that the output generated from an input event is simultaneous to that input event.

[1] To *educe* means to *draw out*.

A LUSTRE stream is a pipeline dataflow, using the prefix order over streams. It is assumed that all elements of a stream will be generated, in the same order as their indices. Different streams may have different clocks, either a global base clock or a Boolean LUSTRE dataflow.

The two main operators of LUSTRE are `->` and `pre`. The equation:

```
X = 0 -> pre X + 1
```

is in some sense equivalent to the Lucid equation:

```
X = 0 fby X + 1
```

However, the Lucid equation has no sense of timing, while the LUSTRE equation is timed. Furthermore, the `pre` corresponds to a delay in automatic control. With the LUSTRE primitives, it is impossible to refer to the future, nor to more than a finite amount of the past.

When several dataflows share the same clock, then the i-th element of each stream sharing that clock must be calculated within the i-th instant of that clock. This approach, a radical simplification of Lucid, is very powerful for representing timed systems.

Today, LUSTRE is the core language in the Scade Toolkit, distributed by Esterel Technologies, and used for programming control systems in, nuclear reactors, avionics and aerospace systems [19]. At the time of writing, Scade is used for programming part of the flight control or the engine control of the following aircraft:

- Airbus A380, A340-500, A340-600 (EU).
- Sukhoi SuperJet 100 (Russia).
- Eurocopter Écureuil/Astar AS 350 B3 (EU).
- Embraer Phenom 100 (Brazil), with Pratt-Whitney PW617F.
- Cessna Citation Encore+ (USA), with Pratt-Whitney PW535B.
- Dassault Aviation Falcon 7X (France).

6 Explicit Multidimensionality: Ferd Lucid and ILucid

The "Beyond Lucid" chapter in [4] presents a discussion of explicitly making Lucid multidimensional, in order to not have to use operators such as `is current`. Two approaches are studied.

Ferd Lucid Ferd Lucid [4, pp. 217–22] was defined so that one could manipulate infinite arrays varying in one time dimension and an arbitrary number of space dimensions. These arrays were called *ferds* (an obsolete word in the Oxford English Dictionary meaning "warlike arrays"). The notation

$$\mathtt{F}_t^{s_0\, s_1\, \cdots}$$

denotes the element where the time dimension has value t, the first space dimension has value s_0, the second space dimension has value s_1, etc. Here are the operators:

$$(\mathtt{first\ F})_t^{s_0\, s_1\, s_2\, \cdots} = \mathtt{F}_0^{s_0\, s_1\, s_2\, \cdots} \tag{17}$$

$$(\mathtt{next\ F})_t^{s_0\, s_1\, s_2\, \cdots} = \mathtt{F}_{t+1}^{s_0\, s_1\, s_2\, \cdots} \tag{18}$$

$$(\mathtt{F\ fby\ G})_t^{s_0\, s_1\, s_2\, \cdots} = \begin{cases} \mathtt{F}_0^{s_0\, s_1\, s_2\, \cdots}, & t = 0 \\ \mathtt{G}_{t-1}^{s_0\, s_1\, s_2\, \cdots}, & t > 0 \end{cases} \tag{19}$$

$$(\mathtt{initial\ F})_t^{s_0\, s_1\, s_2\, \cdots} = \mathtt{F}_t^{0\, s_0\, s_1\, s_2\, \cdots} \tag{20}$$

$$(\mathtt{rest\ F})_t^{s_0\, s_1\, s_2\, \cdots} = \mathtt{F}_t^{(s_0+1)\, s_1\, s_2\, \cdots} \tag{21}$$

$$(\mathtt{F\ cby\ G})_t^{s_0\, s_1\, s_2\, \cdots} = \begin{cases} \mathtt{F}_t^{s_1\, s_2\, s_3\, \cdots}, & s_0 = 0 \\ \mathtt{G}_t^{(s_0-1)\, s_1\, s_2\, \cdots}, & s_0 > 0 \end{cases} \tag{22}$$

The operators `initial`, `rest` and `cby` are counterparts to the Lucid operators `first`, `next` and `fby`. However, they do not have identical semantics. As can be seen above, the *rank* — or *dimensionality* — of `X cby Y` is one more than the rank of `Y`, while the rank of `rest X` is one less than the rank of `X`. This approach is cumbersome, as the programmer needs to keep track of the rank of each of the objects being manipulated.

There are also operators to transform a space dimension into a time dimension, and vice versa:

$$(\texttt{all F})\,{}^{s_0\,s_1\,s_2\,\cdots}_{t} = \texttt{F}\,{}^{s_1\,s_2\,\cdots}_{s_0} \tag{23}$$

$$(\texttt{elt F})\,{}^{s_0\,s_1\,s_2\,\cdots}_{t} = \texttt{F}\,{}^{t\,s_0\,s_1\,s_2\,\cdots}_{t} \tag{24}$$

We give an example to compute the prime numbers using the sieve of Erasthones:

```
all p
where
  i = 2 fby i+1;
  m = all i fby m wvr m mod p ne 0;
  p = initial m;
end where;
```

The behavior of this program is as follows.

$$\texttt{i} = \langle 2, 3, 4, 5, 6, \ldots \rangle$$

$$\texttt{m} =$$

$\texttt{s}_0\backslash\texttt{t}$	0	1	2	3	4	$\cdots$
0	2	3	5	7	11	$\cdots$
1	3	5	7	11	13	$\cdots$
2	4	7	11	13	17	$\cdots$
3	5	9	13	17	19	$\cdots$
4	6	11	15	19	23	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$$\texttt{p} = \langle 2, 3, 5, 7, 11, \ldots \rangle$$

The result is that of `p`, but varying in space dimension $\texttt{s}_0$.

The main drawback of Ferd Lucid is that the extensional treatment of arrays (i.e., a 3D object is a sequence of 2D objects and a 2D object is a sequence of 1D objects) makes the language too difficult to use in solving multidimensional problems, where the dimensions are not only orthogonal but transposable.

ILucid ILucid [4, pp. 223–7] was defined to manipulate "multidimensional time." The `active` operator decreases the rank of its argument, while `contemp` increases its rank. The notation

$$\texttt{F}_{\,t_0\,t_1\,t_2\,\cdots}$$

denotes the element where the first time dimension has value t_0, the second time dimension has value t_1, the third time dimension has value t_2, etc. Here are the operators:

$$(\texttt{first F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,0\,t_1\,t_2\,\cdots} \tag{25}$$

$$(\texttt{next F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,(t_0+1)\,t_1\,t_2\,\cdots} \tag{26}$$

$$(\texttt{F fby G})_{\,t_0\,t_1\,t_2\,\cdots} = \begin{cases} \texttt{F}_{\,0\,t_1\,t_2\,\cdots}, & t_0 = 0 \\ \texttt{G}_{\,(t_0-1)\,t_1\,t_2\,\cdots}, & t_0 > 0 \end{cases} \tag{27}$$

$$(\texttt{active F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,t_0\,t_2\,t_3\,\cdots} \tag{28}$$

$$(\texttt{contemp F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,t_0\,t_0\,t_1\,t_2\,\cdots} \tag{29}$$

A number of "interesting" operators could be defined. For example:

$$(\texttt{current F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,t_1\,t_1\,t_2\,\cdots} \tag{30}$$

$$(\texttt{remaining F})_{\,t_0\,t_1\,t_2\,\cdots} = \texttt{F}_{\,(t_1+t_0)\,t_1\,t_2\,\cdots} \tag{31}$$

Suppose that:

$$X \quad = \quad \big\langle \langle a_0, a_1, a_2, \ldots \rangle, \langle b_0, b_1, b_2, \ldots \rangle, \langle c_0, c_1, c_2, \ldots \rangle, \ldots \big\rangle$$

then:

$$\texttt{current X} \quad = \quad \big\langle \langle a_0, a_0, a_0, \ldots \rangle, \langle b_1, b_1, b_1, \ldots \rangle, \langle c_2, c_2, c_2, \ldots \rangle, \ldots \big\rangle$$
$$\texttt{remaining X} \quad = \quad \big\langle \langle a_0, a_1, a_2, \ldots \rangle, \langle b_1, b_2, b_3, \ldots \rangle, \langle c_2, c_3, c_4, \ldots \rangle, \ldots \big\rangle$$

For example, the following program:

```
contemp (y asa y > n)
where
  n = current active m;
  y = remaining active x;
end where;
```

finds the first-encountered present or future value of `x` which is greater than the present (current) value of `m`.

However, the set of "interesting" operators is unbounded, so more general mechanisms were needed.

7 Intensional Programming: Field Lucid

The aforementioned chasm between denotational and operational issues was recognised by the authors of Lucid. Ashcroft wrote [20]:

> There was no unifying concept that made sense of it all, apart from the mathematical semantics. It was difficult to explain the language operationally. We used statements like "the values of variables are infinite sequences, but don't think of them as infinite sequences — think of them as changing."

This conundrum was solved in a 1986 paper by Faustini and Wadge entitled "Intensional Programming" [21]. In this paper, they made an explicit analogy between Lucid programs and the intensional logic of Richard Montague [22, 23].

The objective of Montague's work was to give a formal semantics to a significant subset of natural language, basing himself on prior work by Rudolph Carnap [24] and Saul Kripke [25]. Carnap had already made the distinction between the *extension* of an utterance — the specific meaning in the exact context of utterance (point in time and space, speaker, listener, etc.) — and its *intension* — the overall meaning for all of the possible contexts of utterance. Kripke had developed a means for using *possible worlds* as indices for giving the semantics of modal logic. Montague's work was to create a new logic, with a rich set of modal operators, using *Kripke structures*.

While developing a semantics for the warehouse, Faustini and Wadge discovered Montague's work, and understood that it is directly applicable to the variables of Lucid. Expressions become intensions mapping possible worlds (multidimensional tags) to extensions (ordinary values). Lucid's operators can be understood as intensional context-switching operators that manipulate the time dimension: `next` moves forward one timepoint and `fby` moves back one timepoint. Simple operations, such as addition, previously treated as pointwise operations on infinite sequences, were simply applied to single values under particular contexts.

The possible worlds semantics of intensional logic significantly clarifies the use of dimensions in Lucid. The dimensions define a coordinate system and each point in multidimensional space is a separate possible world. Lucid's "time" dimension is no longer a conceptual prop. In the words of Ashcroft, "Intensionality clears up the confusion."

Consider Lucid programs using just `next` and `fby`. Then there exists a set of possible worlds, indexed by the natural numbers, $\mathbb{N}$. When the i-th value of stream `X` is being requested, the understanding should be that we are *in* world i and we are simply asking for the value of `X` (in that world). The current possible world can be determined using the `#` primitive, and the `@` primitive can be used to access values in other possible worlds. Both `#` and `@` are *intensional operators*, as are derived operators such as `fby` and `next`. A variable `X` of type $\mathbb{D}$ defines an intension, a mapping $\mathbb{N} \to \mathbb{D}$. The value of `X` in a single possible world is an *extension*.

Once Lucid was understood as an intensional language, further developments consisted of creating more complex, multidimensional, universes of possible worlds, along with the appropriate syntactic adjustments. The first version of Lucid taking this approach was Field Lucid. Its multidimensional data structures can vary independently in multiple orthogonal dimensions. The Lucid operators are expanded to an arbitrary number of dimensions: for every $i \geq 0$, the operators $\texttt{initial}_i$, $\texttt{succ}_i$, $\texttt{sby}_i$, are equivalent to `first`, `next`, and `fby` respectively, but applied to the specific dimension as specified by the suffix. Multidimensional objects can be manipulated, but the dimensions cannot be manipulated, exchanged, transposed or even less created on the fly or passed as dimensional arguments to functions. Here are the operators:

$$(\texttt{first F})_t^{s_0\, s_1\, s_2 \cdots s_i \cdots} = \texttt{F}_0^{s_0\, s_1\, s_2 \cdots s_i \cdots} \tag{32}$$

$$(\texttt{next F})_t^{s_0\, s_1\, s_2 \cdots} = \texttt{F}_{t+1}^{s_0\, s_1\, s_2 \cdots s_i \cdots} \tag{33}$$

$$(\texttt{F fby G})_t^{s_0\, s_1\, s_2 \cdots} = \begin{cases} \texttt{F}_0^{s_0\, s_1\, s_2 \cdots s_i \cdots}, & t = 0 \\ \texttt{G}_{t-1}^{s_0\, s_1\, s_2 \cdots s_i \cdots}, & t > 0 \end{cases} \tag{34}$$

$$(\texttt{initial}_i\ \texttt{F})_t^{s_0\, s_1\, s_2 \cdots s_i \cdots} = \texttt{F}_t^{s_0\, s_1\, s_2 \cdots 0 \cdots} \tag{35}$$

$$(\texttt{succ}_i\ \texttt{F})_t^{s_0\, s_1\, s_2 \cdots s_i \cdots} = \texttt{F}_t^{s_0\, s_1\, s_2 \cdots (s_i+1) \cdots} \tag{36}$$

$$(\texttt{F sby}_i\ \texttt{G})_t^{s_0\, s_1\, s_2 \cdots s_i \cdots} = \begin{cases} \texttt{F}_t^{s_0\, s_1\, s_2 \cdots 0 \cdots}, & s_i = 0 \\ \texttt{G}_t^{s_0\, s_1\, s_2 \cdots (s_i-1) \cdots}, & s_i > 0 \end{cases} \tag{37}$$

The main drawback of Field Lucid is that it adopts an "absolute" view of the multidimensionality; the names of the multiple (orthogonal) dimensions are preordained. Thus, it is not possible to apply a function that expects its arguments to be defined over space dimension 0 to arguments defined over space dimension 1.

8 Dimensional Abstraction: Indexical Lucid

In 1991, Faustini and Jagannathan introduced the language Indexical Lucid [26, 27], which is Lucid with dimensional abstraction. In so doing, all of the multidimensional ideas being worked on were radically simplified.

In Indexical Lucid, new indices can be explicitly created using the `index` declaration within an *indexical where clause*:

```
where
   index a, b;
   ...
end where;
```

The new indices are `a` and `b` and variables may be defined to vary in those indices in addition to the time dimension, the latter always implicit. As predicted in [4], the `is current` declaration is no longer required. The implicit, temporary dimension created by this operation can now be explicitly declared by the programmer.

The standard Lucid operators are available, together with a dimension name suffixing scheme. For example, to access some dimension `a`, one can use any of `first.a, next.a, fby.a, wvr.a, upon.a, asa.a` and `@.a`. The time dimension can be accessed via the standard Lucid operators, or by suffixing `.time` to them. It should be noted that Indexical Lucid dimensions vary over the integers, just as does the time dimension. Also, the terms dimension and index are used

interchangeably here, but at the time only the word index applied. Later, with the explicit references to multidimensional programming, the indices were renamed dimensions.

This new feature allows functions, and even variables, to be defined with formal dimension parameters. One or more dimensions can be added to a definition, using a suffixing scheme similar to that used for the operators. The dimension names used are not actual dimensions, but placeholders for dimensions supplied when the function is called. Today, we might call them dimensional templates.

An additional ternary indexical operator, `to`, copies one dimension to another for a specific expression: `a to b x.a`. The expression `x` varies now in both dimension `b` and dimension `a`. The same result could be achieved by writing `x @.b #.a`.

As mentioned before, the `#` and `@` operators become `#.`*d* and `@.`*d*, where `#.`*d* allows one to query about *part* of the set of dimensions rather than the *entire* set. Similarly, `@.`*d* allows one to change some of the dimensions at a time without having to access any of the others, or the whole set at the same time.

The basic operators thus become:

```
first.d X  = X @.d 0;
next.d X   = X @.d (#.d+1);
prev.d X   = X @.d (#.d-1);
X fby.d Y  = if #.d<=0 then X else Y @.d (#.d-1);
X wvr.d Y  = X @.d T
             where
               T = U fby.d U @.d (T+1);
               U = if Y then #.d else next.d U;
             end where;
X asa.d Y  = first.d (X wvr.d Y);
X upon.d Y = X @.d W
             where
               W = 0 fby.d if Y then (W+1) else W;
             end where;
```

The prime number tester becomes in Indexical Lucid:

```
not idivn asa.a idivn or i * i >= n
where
  index a, b;
  i = 2 fby.a i + 1;
  j = i * i fby.b j >= n;
  idivn = j eq n asa.b j + i;
end where;
```

The eductive model of computation is still used successfully to run Indexical Lucid. The demand-driven interpreter coupled with the warehouse needs little modification in order to handle the multiple dimensions. From the point of view of the eductor, the dimensional tags just become more complex.

To summarize, dimensions can now be manipulated, exchanged, transposed, etc., but the *rank* of an object — the set of dimensions in which it varies — cannot be accessed. For this, it is necessary to have dimensions as first-class values (see §13).

9 Going Parallel: Granular Lucid

The first attempt at a production version of the full Lucid language took place with the creation of Granular Lucid (GLU) in the early 1990's by Jagannathan and Faustini. GLU was a hybrid parallel-programming system that used Indexical Lucid as coordination language to ensure the parallel execution of coarse-grain tasks written in C [28].

In GLU programs, an intensional core (Indexical Lucid) specifies the parallel structure of the application and imperative functions—written in C—perform the calculations. This high-level approach was designed to take advantage of coarse-grain data parallelism present in many applications, for example, matrix multiplication, ray-tracing, video encoding, CT-scan reconstruction, etc.

The eduction algorithm was adapted to distribute these tasks across a shared-memory multiprocessor or a network of distributed workstations. Because of the coarse-grain nature of the problems addressed, Indexical Lucid became an efficient programming environment, since the overhead of the eductive algorithm became inconsequential. Furthermore, the runtime system could be compiled for a variety of system configurations.

To deal with C operations that might involve side-effects, three binary operators were added to Indexical Lucid [29]: "`,`", "`!`" and "`?`". Expression `(x,y)` returns the value of `y` but only after `x` has been evaluated. Expression `(x!y)` returns the value of `y`, after having launched the calculation of `x`. The difference between these two operators is that "`,`" is total while "`!`" is partial, i.e., it will produce a result even if `x` is undefined. Expression `(x?y)` evaluates both `x` and `y` and returns the first value computed, corresponding to a parallel merge.

The development of GLU showed that intensional programming can be naturally applied to real-world problems. Intensional programming, coupled with eduction, provided an elegant, straightforward solution for transforming existing imperative programs with latent parallelism into parallel programs, simply by breaking up the imperative tasks into manageable chunks whose recombination could be described in Lucid and whose distribution could be managed by the runtime system. The GLU language and runtime system were used for the second Lucid book, entitled *Multidimensional Programming* [30], published in 1995 by Ashcroft, Faustini, Jagannathan and Wadge.

The GLU system, although interesting from an academic point of view, was not widely used, for one very important reason; the reworking of existing imperative applications required digging through existing C code to determine how best to distribute tasks, and this was a non-trivial activity.

10 Possible Worlds Versioning: The Context Comes to the Fore

At the same time as the work on Indexical Lucid and GLU was taking place, author Plaice and Bill Wadge began to work on the problem of *possible-worlds versioning*, in which the very *structure* of a computer program varies with a multidimensional context. Although it was not understood at the time, this work would have tremendous influence on the future developments of Lucid.

The concept of possible-worlds versioning was first presented in [31]. A universe of possible worlds was defined *in an algebraic manner*, with a partial order defined over that universe. The structure of a program, or of any other hierarchically defined entity, was an intension. Building a specific extension in a specific possible world simply meant using the most relevant versions of each component, including of build files used to define how components are to be assembled together. The version tag for the resulting built component was the least upper bound of the version tags of all the chosen source components.

In each possible world, a computer program's structure can be different, and each component of the program can be different. When a component is requested, then the *most relevant version* of that component is chosen, and building of the system continues with lower level components.

A detailed discussion of possible-worlds versioning, complete with a presentation of a number of experiments in versioned electronic documents, file systems, Web pages and other electronic media, is given in "Possible Worlds Versioning", by the first two authors, also found in this volume [32].

For the purposes of the presentation below, it became clear, as discussions took place between the various researchers involved, that the context is an active entity that permeates both a program's structure and its behaviour. It would also become clear that the values held by all of the dimensions in a Lucid program formed such a context, and that it should become a first-class

entity (see §14).

11 Versioned Definitions: Plane Lucid

While the general problem of possible-worlds versioning was being studied, so was the more specific problem of adding versioned definitions to Lucid. In particular, Du and Wadge developed Plane Lucid [33, 34], in which multiple definitions can be given for the same variable. When a (*identifier*, *context*) is requested, then the *most relevant* definition for the identifier, with respect to the current context, must be chosen before that definition may be evaluated.

Du and Wadge used Plane Lucid to define a three-dimensional *intensional spreadsheet*. Just as in a regular spreadsheet, the intensional spreadsheet uses two spatial dimensions, but adds a third, temporal dimension. Each cell is indexed by a triple (h, v, t). The spreadsheet is viewed as a single entity whose value varies according to the context. The value in a specific context may be defined in terms of values in other contexts.

Plane Lucid, a language similar to Field Lucid, with additional intensional operators for navigating the space and time dimensions. Each of the three dimensions receives five context-switching operators. These have simple Indexical Lucid equivalents. For example, Table 1 lists the operators for the horizontal dimension together with their Indexical Lucid counterparts.

Table 1: Indexical Lucid equivalents to horizontal Plane Lucid operators

Plane Lucid operators	**Indexical Lucid equivalents**
`side A`	`A @.h 0`
`right A`	`A @.h (#.h + 1)`
`left A`	`A @.h (#.h - 1)`
`A hsby B`	`if #.h > 0 then B @.h (#.h - 1) else A fi`
`A hbf B`	`if #.h < 0 then A @.h (#.h + 1) else B fi`

Since not every cell has the same definition, Du and Wadge specified four definition levels for values of cells. They are, in decreasing order of priority: local, dimensional, planar and global. As might be expected, they represent progressive generalisations. For example, in spreadsheet `S`:

```
S[h <- 3, v <- 4, t <- 5] = 3;
S[v <- 4, t <- 5] = 2;
S[t <- 5] = 1;
S = 0;
```

cell $(3, 4, 5)$ is locally defined to take the value 3; cells $(?, 4, 5)$ are dimensionally defined to take the value 2; cells $(?, ?, 5)$ are planarly defined to take the value 1; and all other cells are given a default value of 0.

12 List Dimensions: Attributes and Functions

In Indexical Lucid, a dimension can only take integers for values. This is natural, as the objects being manipulated by Indexical Lucid are assumed to be multidimensional arrays, and the integer tuples can be used to index into these arrays. However, there are other kinds of data structure that require different kinds of index. In particular, a finite tree can naturally be understood as a mapping from tuples — finite lists — to values.

This intuition was taken up by Senhua Tao [35], under Wadge's supervision, to treat attribute grammars — used for defining the syntax and semantics of programming languages — intensionally, in an effort to properly support circular attribute definitions.

Tao created TLucid, an extension of Lucid with a list-valued dimension, encoding the position inside a syntax tree, in addition to the standard time dimension. The length of the finite list

corresponds to the length of the path to the corresponding node in the tree. Context-switching operators are added to TLucid for traversing this new dimension. These operators are given in Table 2, along with equivalents in an "Indexical Lucid" that would allow non-integer dimensions.

Table 2: "Indexical Lucid" equivalents to TLucid operators

TLucid operators	**"Indexical Lucid" equivalents**
`index`	`#.i`
`root N`	`N @.i nil`
`parent N`	`N @.i (tl #.i)`
`nextsib N`	`N @.i (((hd #.i) + 1):(tl #.i))`
`child(N, k)`	`N @.i (k:#.i)`

The same technique, this time in a multidimensional framework, was used by Panagiotis Rondogiannis and Wadge [36, 37, 38] to implement higher-order functions in Lucid itself. In these articles, they simulate the calling structure of a higher-order program by using lists-valued dimensions indicating which of each of the functions has been called and from where. The advantage of this approach is that no closure operations are required. However, the technique is not generally applicable to partially applied functions.

13 First-class Dimensions: Multidimensional Lucid

The natural next step was to introduce dimensions as first-class values, work undertaken by Joey Paquet and author Plaice in Tensor Lucid [39, 40], developed to write tensor equations naturally. By allowing declared dimensions to be used as ordinary values, the total dimensionality of an object becomes directly accessible. However, all dimensions must still be created lexically, and cannot be created on demand, during execution, and defining values as dimensions was the next step.

Paquet's work was simplified by author Plaice in Multidimensional Lucid [41], in which any ground value may be used as a dimension. Multidimensional Lucid is simply ISWIM with two new primitives, indexical query (`#`E) and context change (E `@`E_1 E_2). In `#`E, expression E is evaluated to a value v and the context is then queried, using v as the dimension. In E `@`E_1 E_2, expression E_1 is evaluated to v_1 and E_2 is evaluated to v_2; then E is evaluated in the current context, modified so that dimension v_1 yields the value v_2. The difference with Indexical Lucid is that the dimensions need not always be identifiers; therefore they can be created on the fly, opening up many new possibilities.

The example program is here shown in Multidimensional Lucid:

```
n = input;
i = 2 fby.0 i + 1;
j = i * i fby.1 j + 1;
idivn = j eq n asa.1 j >= n;
output = not idvin asa.0 idivn or i * i >= n;
```

The semantics of Multidimensional Lucid was much simpler than that of Indexical Lucid. However, since dimensions could be created on the fly, the eductive algorithm used since the first implementation of Original Lucid was no longer applicable, because the potential wastage of memory while caching partial results was essentially unbounded. Until this problem could be resolved, developing an interpreter for Multidimensional Lucid would have been of limited utility.

14 First-class Contexts

The idea of contexts as first-class values in Lucid came from two separate directions. As mentioned in §10, one was the work in possible-worlds versioning, leading to an understanding of a context

as an entity permeating the behaviour and structure of a program or of a set of programs. The second came from Joey Paquet, in the development of the GIPSY project that he is leading in developing a generic infrastructure for the interpretation and compilation of Lucid variants [42].

In 2001, Paquet sent an email to his Concordia University colleague Peter Grogono and to author Plaice, with a proposal for generalising the work on dimensions as values. He suggested that first-class contexts be added to Lucid. The proposal was that the `@` operator should become binary: E' `@` E should mean that E should evaluate to a context, and that E' would then be evaluated, using E as its new context. In that letter, he also proposed that perhaps the E could even evaluate to a *set of contexts*, which would mean that E' would be evaluated in each context within that set, yielding a set of values.

Notwithstanding the possible advantages of adding contexts and sets of contexts to Lucid, the implementation issues of such an addition are daunting, because the problems outlined in the previous section would simply be amplified.

15 Cartesian Programming: TransLucid

The TransLucid project started in late 2005, with a visit to UNSW by Bill Wadge. During his stay, he explained an earlier idea that he and Tony Faustini had had about *lazy eduction*, in which requests to the warehouse caching the partial results would be incremental. When requesting an (*identifier*, *tag*) pair, the execution engine would begin with an empty tag, and the warehouse would either return a result or a request for information about additional dimensions, in which case a new request would need to be made with a more refined tag. Once sufficient information was provided, then the warehouse could provide the answer or provoke a calculation, if necessary.

With this idea, an initial set of rules for a language with contexts as values was developed, as outlined in the previous section. This language, presented in author Ditu's PhD thesis [43], had as goal to be usable as a real programming language, either directly or as the target language for compiling languages in a variety of paradigms. An example problem is the Ackermann function:

```
ack = if #0 == 0 then #1 + 1
      elsif #1 == 1 then ack @ [ 0 <- #0-1, 1 <- 1 ]
      else ack @ [ 0 <- #0-1, 1 <- ack @ [ 1 <- #1-1 ] ] ;;
```

The use of `[ ... ]` allows the specification of a new context relative to the current context, replacing the values for an arbitrary set of dimensions. As in GLU, new types and operators can be added to the language.

With TransLucid, there are two kinds of data structure: the explicit tuple, and the implicit infinite multidimensional array, which we call a *hyperdaton*. When used as a context, the explicit tuple corresponds exactly to a multidimensional coordinate into the hyperdaton. For this reason, we have coined a new term for programming in TransLucid: *Cartesian programming*, with the clear allusion to the Cartesian referential system. In some sense, everything has become simpler. We have reached, in the words of Jean Dhombres [44], « La banalidad del referencial cartesiano » ("The trivialitity of the Cartesian referencial system.")

Since one can always use more parameters—dimensions—as needed to describe a problem, it is possible to translate other programming paradigms into TransLucid, in order to have a single intermediate language. It is also possible to go the other way, and to add some of the multidimensional features of TransLucid to other languages.

The development of TransLucid is continuing, with many experiments in implementation, which are highly relevant to the original discussion in the introduction, which related the importance of architecture design to the development of programming languages. In his honours thesis [45], Toby Rahilly presented a multithreaded eduction engine, and in so doing, derived the idea of *lazy tuples*, in which the values associated with the dimensions in a context are only calculated if necessary. The implementation led, not simply to faster running programs, but also to a more powerful programming model. With lazy tuples, it will be possible to extend TransLucid to manipulate infinite contexts.

At the semantic level, work needs to be done so that TransLucid can be integrated with other aspects of a computer system, in order for it to be usable with files, networks, databases, stream I/O, memory-mapped I/O, and so on. To do this will require adding sets of contexts, types and expressions as values, along with means to refer to time and concurrency. In time, a TransLucid system, with an evolving set of declarations, definitions and demands, will correspond to an object in an object-oriented environment.

16 Conclusions

The initial Lucid publications go back to some 30 years ago, with the attempts by Bill Wadge to formalize existing computation for the purposes of formal program verification. Since then, the Lucid programming language has undergone many changes, leading to the TransLucid language of today. These changes were not mere technical sleights of hand: most of them required a radical reinterpretation of the very concept of computation, leading each time to a deeper, yet simpler, understanding. At each of these stages, Bill Wadge has been present, with ideas for his students and his colleagues. We are all richer for it, as is the discipline.

The current development of a wide variety of manycore and multicore architectures makes the current research in TransLucid of wide relevance. Successful deployment of declarative programming is of greater importance today, where high-performance computing becomes mainstream, than it ever has been.

References

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[2] Gordon E. Moore. Excerpts from A Conversation with Gordon Moore: Moore's Law. Video Transcript, Intel, 2005.

[3] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 2006.

[4] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.

[5] E. A. Ashcroft and W. W. Wadge. Lucid—A formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3):336–354, September 1976.

[6] E. A. Ashcroft and W. W. Wadge. Lucid, A Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[7] Edsger Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[8] Ole-Johan Dahl, Edsger Wybe Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1972. ISBN 0122005503.

[9] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976. ISBN 013215871X.

[10] Bill Wadge. The Origins of Lucid. Presentation to the Institut für Informatik, Christian-Albrechts-Universität zu Kiel, Germany, 8 February 2007.

[11] Victor Selivanov. Wadge reducibility and infinite computations. *Mathematics in Computer Science*, This volume, 2008.

[12] Edward A. Ashcroft and William W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[13] Gilles Kahn. The semantics of simple languages for parallel programming. In *Information Processing 74*, pages 471–475. North-Holland, 1974.

[14] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998. North-Holland, 1977.

[15] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[16] Calvin B. Ostrum. *The Luthid 1.0 Manual*. Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[17] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. A declarative language for programming synchronous systems. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1987*, pages 178–188, Munich, January 1987.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[19] Esterel Technologies. `http://www.esterel-technologies.com/`. Last visited 8 December 2007.

[20] Edward A. Ashcroft. The Lucid Language: Past, Present, and Future. In *Proceedings of the 5th ISLIP*, pages 1–15, San Francisco, USA, Apr 1992. IEEE Computer Society. `http://islip.web.cse.unsw.edu.au/1992`.

[21] A. A. Faustini and W. W. Wadge. Intensional programming. In J. C. Boudreaux, B. W. Hamil, and R. Jenigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier North-Holland, 1987.

[22] Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[23] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics*. D. Reidel, Dordrecht, Holland, 1981.

[24] Rudolph Carnap. *Meaning and Necessity*. University of Chicago Press, 1956. Seventh Impression 1975.

[25] Saul A. Kripke. *Naming and Necessity*. Harvard University Press, Cambridge, Mass., 1980.

[26] Anthony A. Faustini and Rangaswamy Jagannathan. Indexical Lucid. In *Proceedings of the 4th ISLIP*, pages 19–34, Menlo Park, USA, Apr 1991. SRI International. `http://islip.web.cse.unsw.edu.au/1991`.

[27] Anthony A. Faustini and Rangaswamy Jagannathan. Multidimensional Problem Solving in Lucid. Technical Report SRI-CSL-93-03, Computer Science, SRI International, Menlo Park, USA, 1993.

[28] Rangaswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A High-Level System for Granular Data-Parallel Programming. *Concurrency: Practice and Experience*, 9(1):63–83, Jan 1997.

[29] Rangaswamy Jagannathan and Chris Dodd. GLU Programmer's Guide. Technical report, Computer Science, SRI International, Menlo Park, USA, 1996. Version 1.0.

[30] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming*. Oxford University Press, New York, 1995.

[31] John Plaice and William W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–276, March 1993.

[32] Blanca Mancilla and John Plaice. Possible worlds versioning. *Mathematics in Computer Science*, This volume, 2008.

[33] W. Du and W. W. Wadge. The eductive implementation of a three-dimensional spreadsheet. *Software–Practice and Experience*, 20(11):1097–1114, 1990.

[34] W. Du and W. W. Wadge. A 3D spreadsheet based on intensional logic. *IEEE Software*, 7(3):78–89, 1990.

[35] Senhua Tao. TLucid and Intensional Attribute Grammars. In *Proceedings of the 6th ISLIP*, pages 94–106, Laval University, Québec, Canada, Apr 1993. `http://islip.web.cse.unsw.edu.au/1993/`.

[36] Panos Rondogiannis and William W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

[37] Panos Rondogiannis and William W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, May 1999.

[38] Aggelos Charalambidis, Athanasios Grivas, Nikolaos Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, This volume, 2008.

[39] Joey Paquet. *Intensional Scientific Programming*. PhD thesis, Department of Computer Science, Laval University, Québec, Canada, April 1999.

[40] Joey Paquet and John Plaice. *The semantics of dimensions as values*, volume II, pages 259–273. World-Scientific, Singapore, 2000. Based on the papers at ISLIP'99.

[41] John Plaice. Multidimensional Lucid. In Kropf et al. [46], pages 154–160. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[42] Joey Paquet and Peter G. Kropf. The GIPSY architecture. In Kropf et al. [46], pages 144–153. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.

[43] Gabriel Ditu. *The Programming Language TransLucid*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, March 2007.

[44] Jean Dhombres. La banalidad del referencial cartesiano. In Carlos Álvarez J. and J. Rafael Martinéz Enríquez, editors, *Descartes y la ciencia del siglo XVII*. siglo veintiuno editores, México, 2000.

[45] Toby Rahilly. Experiments in concurrent implementations of TransLucid. Honours Thesis, Computer Science, UNSW, 2007.

[46] Peter Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors. *Distributed Communities on the Web*, volume 1830 of *Lecture Notes in Computer Science*. Springer, Berlin, 2000. Third International Workshop, DCW 2000, Québec, Canada, June 2000, Proceedings.