Multidimensional Infinite Data in the Language Lucid

Jarryd P. Beck[1] jarrydb@cse.unsw.edu.au
John Plaice[1,2] plaice@cse.unsw.edu.au
William W. Wadge[3] wwadge@cs.uvic.ca

[1]School of Computer Science and Engineering, UNSW, Australia
[2]Department of Computer Science and Software Engineering, Concordia University, Canada
[3]Department of Computer Science, University of Victoria, Canada

Technical Report
UNSW-CSE-TR-201306
February 2013

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Although the language Lucid was not originally intended to support computing with infinite data structures, the notion of (infinite) sequence quickly came to the fore, together with a demand-driven computation model in which demands are propagated for the values of particular values at particular index points. This naturally generalized to sequences of multiple dimensions so that a programmer could, for example, write a program that could be understood as a (nonterminating) loop in which one of the loop variables is an infinite vector.

Programmers inevitably found use for more and more dimensions, which led to a problem which is fully solved for the first time in this paper. The problem is that the implementation's cache requires some estimate of the dimensions actually used to compute a value being fetched. This estimate can be difficult or (if dimensions are passed as parameters) impossible to obtain, and the demand-driven evaluation model for Lucid breaks down.

We outline the evolution of Lucid which gave rise to this problem, and outline the solution, as used for the implementation of TransLucid, the latest descendant of Lucid.

1 Introduction

In this paper, we present an efficient cache-based implementation of the evaluation of arbitrary-dimensional arrays of arbitrary extent, as defined in the programming language Lucid. This problem has remained open since the mid-1980s, when multiple dimensions were first added to Lucid. The presentation in this paper is given in a historical style, by explaining the insights, at the pragmatic, semantic and implementation levels, that led from the original Lucid of 1975 to the current version of Lucid, called TransLucid.

In Lucid, a system of mutually recursive equations is used to define a number of variables, whose values are infinite data structures. The latter cannot be evaluated on a computer, so subsets thereof must be demanded by the programmer, and the interpreter or compiler must then satisfy those demands. Demands are of the form $(\mathit{variable}, \mathit{context})$, where the *context* specifies an index in a multidimensional space, i.e., a set of dimension–ordinate pairs. During the evaluation of a demand (x, κ), other (x_i, κ_i) pairs will have to be demanded. Depending on the program, it is possible that the same demand will be made several times, and so a cache-based implementation, corresponding to memoization in the implementation of functional programming languages, becomes useful.

However, during the evaluation of a demand (x, κ), it may be that only part of κ, i.e., only some of the dimension–ordinate pairs in κ, are needed to compute that demand. The solution provided in this paper corresponds to building a decision tree in the cache, where the root, for each variable x, corresponds to evaluating x with an empty context. Suppose (x, κ) is to be evaluated. Then an iterative process takes place: at iteration i, the cache is asked for the contents of (x, κ_i), where $\kappa_0 = \emptyset$, the empty context; should a normal value v be returned, that is the answer; should a set of dimensions Δ_i be returned, then $\kappa_{i+1} = \kappa_i \cup (\kappa \lhd \Delta_i)$, i.e., the context is augmented to include those dimensions in Δ_i. Should there be no entry, then a computation needs to be initiated.

The paper begins with a presentation of the origins of the problem, with the creation of Lucid in the mid-1970s. Initially, Lucid variables defined single-dimensional infinite streams. Nevertheless, it did not take long before the utility of a cache-based implementation became apparent. The presentation explains the need for ever more dimensions, ultimately leading to the current TransLucid language, the only Lucid descendant with higher-order functions, in which any ground value can be used as dimension, and dimensions are first-class values. The plethora of dimensions in TransLucid, both for programming and implementation purposes, forced the solution to this longstanding problem.

2 Lucid's birth

The language Lucid, originated by author Wadge with E. A. Ashcroft, first appeared in 1975 [2]. It was not at first intended to be a language for infinite data or even (as it later became) a dataflow language. Instead, the goal was to create a relatively conventional Pascal-like language in which iteration (as well as recursion) could be expressed in a mathematically 'pure' form. The idea was that program verification could proceed directly from the statements of a program, without the need to convert the program into recursion or attach assertions at selected points.

The strategy was to specify iterative calculations with temporal operators, initially *first* and *next*. Thus, where a Pascal-like program would have two separate assignments for a simple counter

```
I := 1;
...
while ...
  I := I+1;
  ...
```

the Lucid programmer would write two separate but consistent assertions:

$$\begin{aligned} &\mathit{first}\ I = 0; \\ &\mathit{next}\ I = I + 1; \end{aligned}$$

These two assertions could be treated as axioms from which properties of the program could be derived using rules of inference in an appropriate temporal logic; one with rules of inference such as the temporal induction rule

$$\frac{\mathit{first}\ P \qquad P \to \mathit{next}\ P}{P}$$

The idea was simple enough, but great care had to be taken in developing the temporal logic (as described in [3]).

3 The infinite is unavoidable

To make this logic practical, we[1] had to assume a nonterminating model of time; one in which the set of time points is the natural numbers with the usual order. This more or less forced us to think of program variables (such as I above) as denoting infinite sequences of data values (so that I stands for $\langle 0, 1, 2, 3, \ldots \rangle$). The assertions of a Lucid program then become equations between infinite sequences and the temporal operators like *first* and *next* are revealed as operators on infinite sequences:

$$\begin{aligned} \mathit{first}\ \langle x_0, x_1, x_2, \ldots \rangle &= \langle x_0, x_0, x_0, \ldots \rangle \\ \mathit{next}\ \langle x_0, x_1, x_2, \ldots \rangle &= \langle x_1, x_2, x_3, \ldots \rangle \end{aligned}$$

In fact even the modest addition operation is infinitary:

$$\langle x_0, x_1, x_2, \ldots \rangle + \langle y_0, y_1, y_2, \ldots \rangle = \langle x_0 + y_0, x_1 + y_1, x_2 + y_2, \ldots \rangle$$

In fact it was not the infinite but the finite that gave rise to technical problems: how do you write a loop that terminates?

For example, consider the Pascal-like program to calculate the square of 10:

```
I := 0;
S := 0;
while I < 10
  I := I + 1;
  S := S + 2*I + 1
```

When the loop terminates, `S` has the desired value. We considered incorporating finite streams, but the technical complications were nightmarish. Instead, we introduced a new operator *asa* (short for "as soon as") that extracts from a sequence the value the first argument has when the second argument first becomes true:

$$\langle x_0, x_1, x_2, \ldots \rangle\ \mathit{asa}\ \langle p_0, p_1, p_2, \ldots \rangle = \langle x_k, x_k, x_k, \ldots \rangle$$

where k is the index of the first true value of p.

The Lucid analog of the square program then becomes

$$\begin{aligned} &\mathit{first}\ I = 0; \\ &\mathit{next}\ I = I + 1; \\ &\mathit{first}\ J = 0; \\ &\mathit{next}\ J = J + 2 * I + 1; \\ &\qquad S = J\ \mathit{asa}\ I\ \mathit{eq}\ 10; \end{aligned}$$

[1] In Sections 3–9, the pronoun "we" refers to author Wadge with E. A. Ashcroft.

Another useful operator we quickly discovered is *fby* (short for "followed by"). The output of the *fby* operator is the first component of its first input followed by its entire second input, with the index increased by one:

$$\langle x_0, x_1, x_2, \ldots\rangle \; \mathit{fby} \; \langle y_0, y_1, y_2, \ldots\rangle = \langle x_0, y_0, y_1, y_2, \ldots\rangle$$

The *fby* operator allowed us to dispense with pairs of equations defining a single variable; for example, the square program above shrinks to

$$\begin{aligned} &I = 0 \; \mathit{fby} \; I + 1; \\ &J = 0 \; \mathit{fby} \; J + 2 * I + 1; \\ &S = J \; \mathit{asa} \; I \; \mathit{eq} \; 10; \end{aligned}$$

Merging the pairs of equations had an important consequence: every variable is defined by a single equation and thus we can invoke a least-fixed-point theorem, provided we have a domain of sequences over which the operators are monotonic and continuous. Which domain? The choice would turn out to have important consequences.

4 The dataflow model emerges

At the Arc-et-Senans conference at which Lucid was first presented, we met Gilles Kahn and learned of his elegant dataflow model [8]. We quickly realized that our programs (at least some of them) could be interpreted operationally in terms of this model. For example, *fby* can be thought of as a processing station that accepts and passes on the first data token that arrives on its left-hand input stream; thereafter it simply accepts tokens arriving on its right input stream and passes them on unchanged.

Even addition corresponds to a Kahn style processing station (or "filter", to use UNIX terminology). The + filter repeatedly awaits a pair of tokens from each input stream, adds them as numbers, and sends the sum on the output stream.

The dataflow model suggested other filters as well. For example, *wvr* (short for "whenever"), which passes on only those tokens arriving on the right-hand stream that correspond to tokens representing the value true on the left input stream. For example,

$$\langle \mathit{tt}, \mathit{ff}, \mathit{tt}, \mathit{tt}, \mathit{ff}, \mathit{tt}, \ldots\rangle \; \mathit{wvr} \; \langle y_0, y_1, y_2, \ldots\rangle = \langle y_0, y_2, y_3, y_5, \ldots\rangle$$

Kahn had described a domain of streams—all finite as well as infinite sequences— and this meant that arbitrary programs are meaningful. (We take a program to be a set of equations defining program variables, one equation per variable.) By the usual techniques, these domains yield function spaces, and this in turn meant that every recursive function definition is a meaningful definition of a filter. This allowed some ingenious programs, such as the following one, which uses *wvr* to generate the (infinite) stream of all primes:

$$\begin{aligned} &\mathit{sieve}(N) \; \texttt{where} \\ &\qquad N = 2 \; \mathit{fby} \; N + 1; \\ &\qquad \mathit{sieve}(X) = \mathit{first} \; X \; \mathit{fby} \; \mathit{sieve}\big(X \; \mathit{wvr} \; (X \; \mathit{mod} \; \mathit{first} \; X) \; \mathit{ne} \; 0\big) \end{aligned}$$

In operational terms, this program corresponds to a Kahn network that grows dynamically as the computation proceeds.

We soon stopped describing Lucid as "Pascal-like" and began thinking of it as a declarative/functional dataflow language. We, together with interested colleagues, began working on implementations.

5 Limitations of Kahn Dataflow

One of these implementers was David May (who later became a major contributor to Occam and the Transputer). May (and other implementers) discovered a serious problem—that the Kahn domain semantics for Lucid did not validate the rules of inference we had published [3, 4] for the language.

The heart of the matter was the disparity between the operational and denotational semantics for `if`–`then`–`else`. The Lucid papers assumed that `if`–`then`–`else` worked pointwise (like +), so that, for example

$$\texttt{if } \langle tt, ff, tt, ff, \ldots\rangle \texttt{ then } \langle x_0, x_1, x_2, \ldots\rangle \texttt{ else } \langle y_0, y_1, y_2, \ldots\rangle = \langle x_0, y_1, x_2, y_3, \ldots\rangle$$

But suppose that the x stream has only one component—that after x_0, it deadlocks and nothing more arrives. The (Kahn) operational semantics produces $\langle x_0, y_1\rangle$. The value y_3 never sees the light of day because the `if`–`then`–`else` filter deadlocks waiting for x_2, which never arrives.

The denotational semantics from the Lucid paper makes no sense for this example because it refers to a non-existent x_2 value. We can salvage something by using the value $\perp$ ("bottom") to represent the 'value' of a computation that does not terminate. The denotational semantics then specifies the output to be

$$\langle x_0, y_1, \perp, y_2, \ldots\rangle$$

We called streams with $\perp$ components "intermittent streams", but they make no sense in terms of Kahn dataflow. The published denotational semantics of Lucid corresponds not to the Kahn domain of streams, but to the (much larger) domain of intermittent streams (with pointwise approximation).

David May and (simultaneously) other implementers found an ingenious solution to this discrepancy. They abandoned the Kahn model and introduced a demand-driven system, where the i-th component of a stream is computed only on receipt of an explicit demand for it. The program begins by demanding the index-0 value of the output. These demands propagate backwards through the defining expressions and generate demands for other values at possibly different indices. This demand-driven model, which we later called *eduction*, does not treat sequence indices as time. It supports intermittent streams and validates the published denotational semantics.

The eduction model requires a cache to be practical. When the i-th value of variable V is calculated, the value is stored in a cache, labeled with i and V. When a demand is generated, the implementation first looks in the cache to see if it is already available. Unfortunately the cache has to be periodically thinned out or it will grow rapidly. We eventually discovered a very effective thinning heuristic, called the "retirement plan", that in practice kept the cache at a reasonable size and almost never threw out anything later demanded [5].

6 Multidimensionality

Abandoning the interpretation of stream index as time meant giving up the Kahn dataflow model. (Not completely: it could still be used as a heuristic guide for programmers.)

It opened up, however, an important new possibility: 'streams' that depend on more than one dimension.

In the simplest case, we can introduce a new 'space' dimension s, so that in general the value of a variable depends on both the natural number index t and also the natural number index s. We let V_t^s denote the value of variable V at space point s and time point t.

We add spatial analogs of *first*, *next*, and *fby* called *init* ("initial"), *succ* ("successor") and *sby* ("succeeded by"). The operators act only on the space dimension and ignore the time dimension, so that, for example

$$succ(V)_t^s = V_t^{s+1}$$

Variables that depend only on the space dimension can be thought of as infinite vectors; the equation

$$N = 0 \; sby \; N + 1;$$

defines the vector of all natural numbers.

Variables that depend on both parameters can be pictured in at least three different ways: as streams of (infinite) vectors, as vectors of (infinite) streams, or as two-dimensional matrices.

The following program declares a variable P defining a two-dimensional variable that represents Pascal's triangle

$$P = (1 \; sby \; 0) \; fby \; sum_2(P);$$
$$sum_2(X) = 1 \; sby \; X + succ(X);$$

Here sum_2 is a spatial filter that takes a vector X as an argument and produces a vector whose first component is one but thereafter is the sum of the corresponding value of X plus its previous value. In other words, using square brackets for vectors

$$sum_2[x_0, x_1, x_2, \ldots] = [1, x_0 + x_1, x_1 + x_2, \ldots]$$

The program defining P given above can be understood as a simple iteration in which sum_2 is repeatedly applied to the initial value $[1, 0, 0, 0, \ldots]$ of P.

It is not hard to see that the $t = 1$ value of P is $[1, 1, 0, 0, \ldots]$, that the $t = 2$ value is $[1, 2, 1, 0, 0, \ldots]$, that the time $t = 2$ value is $[1, 3, 3, 1, 0, \ldots]$, and so on.

If we display these values with the space dimension increasing horizontally and the time dimension vertically,

	space $\rightarrow$						
P	0	1	2	3	4	5	$\cdots$
time $\downarrow$ 0	1	0	0	0	0	0	$\cdots$
1	1	1	0	0	0	0	$\cdots$
2	1	2	1	0	0	0	$\cdots$
3	1	3	3	1	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

we can clearly see Pascal's triangle.

It is important to emphasize that these so-called "space" and "time" dimensions are on an equal footing—neither has priority. We can write another program for P in which space plays the dominant role. If we think of P as a vector of streams, its first component is the stream $\langle 1, 1, 1, \ldots \rangle$, its second component is the stream $\langle 0, 1, 2, 3, \ldots \rangle$, its third the stream $\langle 0, 0, 1, 3, \ldots \rangle$, and so on.

These streams are generated by a simple rule: apart from the first stream, each is the result of calculating a runnning total of the stream to the left. We therefore define

$$tot(X) = S \; \texttt{where} \; S = 0 \; fby \; X + S;$$

It is easy to see that

$$tot\big(\langle x_0, x_1, x_2, x_3, \ldots \rangle\big) = \langle 0, x_0, x_0 + x_1, x_0 + x_1 + x_2, \ldots \rangle$$

and the complete program for P is the definition of tot plus the equation

$$P = 1 \; sby \; tot(P);$$

Finally, we can give a *sieve* program for the stream of primes which is purely iterative, i.e., has no recursively defined filters. We define a loop in which a vector V is progressively thinned

out by removing multiples of its initial component.

$$N = 2 \; sby \; N + 1;$$
$$V = N \; fby \; sieve(V);$$
$$sieve(X) = X \; wherever \; (X \bmod init \; X) \; ne \; 0;$$
$$Primes = init \; V;$$

(The operator *whenever* is the spatial analog of *wherever*).

7 Multidimensional eduction

Space dimensions were mentioned only briefly in [17, pp.217–222], but multidimensionality played a vital role in the further evolution of the language. In this the implementers were in the lead. The pLucid interpreter supported space as well as time—the programs given in the previous section can all be run on the pLucid interpreter.

The pLucid interpreter actually supported a system with *infinitely many* space dimensions, including operators to shift space dimensions and to move coordinates back and forth between time and space, although this feature was experimental and not documented in the pLucid manual, reproduced in [17].

In fact, as should be apparent, it is relatively easy to extend eduction to more than one dimension. A demand consists of a variable plus a set of dimension–coordinate pairs, and values stored in the cache are tagged with multiple dimension values.

The builders of the pLucid interpreter (started by Calvin B. Ostrum and completed by Tony Faustini) even found an ingenious way to implement function calling (including recursion) without resorting to dynamic networks. They introduced an extra user-hidden *place* dimension that encodes a position in the tree of function calls. The function name and its formal parameters become simple variables that vary over the place dimension, with the values of the formal parameters determined by the values of actual parameters. Rondogiannis [16] describes this system and extends it to a wide class of typed higher-order Lucid programs (Lucid was originally first-order). Briefly, Rondogiannis used a separate place dimension for every level in the function type hierarchy.

Notable among the successors to Lucid, as described in [1], was Indexical Lucid. It was used in the GLU system developed by Faustini and Jagannathan to extract parallelism from legacy C software. Indexical Lucid took multidimensionality a step further, using not only (hidden) place dimensions, but allowing user-declared dimensions. (Operators like *fby* were indexed by dimensions.)

The obvious next step was to allow dimensions as first-class variables, but attempts to do so aggravated serious limitations in the eductive treatment of multidimensionality.

8 Accessing a multidimensional cache

The problem already appears in a relatively tame form when we add just one space dimension to the time dimension. In principle, every demand consists of a variable (i.e., the name of the variable, not one of its values), together with coordinates for both s and t; for example, "requesting the value of variable P with $t = 3$ and $s = 2$". Eventually, this calculation returns the value 3 and this fact is entered in the cache:

$$\text{value of } P \text{ at } t = 3 \text{ and } s = 2 \text{ is } 3$$

When the same demand is generated again, the cache entry avoids a recomputation.

All this works well for the first Pascal-triangle program, but a problem arises if we consider the iterative *primes* program. Again, demands for V with values for s and t will be propagated, but since V depends indirectly on N, we may eventually see demands of the form

$$\text{requesting the value of } N \text{ at } t = 4 \text{ and } s = 2$$

The calculation of this value will ignore the value of t and produce the value 3. Ignoring the value of t is not the problem; the problem is that the cache entry

$$\text{value of } N \text{ at } t = 4 \text{ and } s = 2 \text{ is } 3$$

will have the value $t = 4$ as part of its 'tag'. This information is irrelevant and should not be included in the tag. Otherwise, the cache will fill with duplicated entries

$$\begin{aligned}&\text{value of } N \text{ at } t = 0 \text{ and } s = 2 \text{ is } 3\\ &\text{value of } N \text{ at } t = 1 \text{ and } s = 2 \text{ is } 3\\ &\text{value of } N \text{ at } t = 2 \text{ and } s = 2 \text{ is } 3\\ &\ldots\end{aligned}$$

which should all be replaced by the generic entry

$$\text{value of } N \text{ at } s = 2 \text{ is } 3$$

The problem is not in producing suitably generic entries; as a computation proceeds, the interpreter could keep a running tally of those dimensions whose values are actually required, and tag the cache entry with the coordinates of just those dimensions.

The problem finally appears when we have to search the cache. Suppose the current request is for N at $t = 4$ and $s = 2$. When we search the cache for the requested value, what tag do we look for? We need to know which dimensions are actually required in the computation, and since we are trying to avoid performing the computation in the first place, there is no *a priori* reason that we should have this information.

If there are only two dimensions to begin with, the problem is not very serious. We could search first with the tag $t = 4 \wedge s = 2$, then try $t = 4$, then (successfully) try $s = 2$. But suppose there are dozens of dimensions to begin with, or that users can declare their own dimensions, or that there are in fact infinitely many. The cache scheme breaks down.

The pLucid interpreter never solved this problem, which is one of the reasons the multidimensionality was undocumented. Fortunately, many of the programs people write in practice did not cause severe duplication of cache entries. But this was not always the case, and would have been too much to hope for in the presence of, say, dimensions as function parameters.

9 Dimensionality analysis

One approach, used by Indexical Lucid, was to perform a static analysis of the program to compute upper bounds on the dimensionality of program variables. For example, in analyzing the *primes* program, the procedure might conclude that V in general requires values of both s and t, but that N needs at most s. Then the general rule would be that cache entries for V are tagged with an s and a t value, but that values for N are tagged only with an s value. A simple abstract-interpretation approach gives good results for simple 0-order (variables only) programs like that for *primes*. But the presence of user-defined filters greatly complicates the calculation. Suppose

$$foo(X, Y) = init\ X + (Y\ fby\ 3)$$

then *foo* (roughly speaking) has the effect of passing on the dimensionalities of its arguments, with the s dimension of X supressed, but with t unconditionally added. And of course we have to deal with recursive filter definitions. We need a complex domain of filter dimensionalities which explodes if we move to higher orders. And it all breaks down anyway if dimensions can be passed as parameters.

Indexical Lucid's dimensionality analysis was fairly crude; for example, it unioned the dimensionalities of the actual parameters of a filter and used a worst-case analysis to estimate the dimensionality of the the result. It worked well enough to handle a large range of practical programs, but by no means constituted a definitive solution of multidimensional eduction.

10 The proposed solution—lazy tags

Author Wadge with Tony Faustini proposed a possible solution in the late 1980s [6]. The idea is that during eduction and caching, even the tags are generated using a demand-driven scheme. It is this solution which we will retain, when implementing TransLucid (see Section 14).

As indicated above, when the value of a variable is computed, a tally is kept and only referenced dimensions are included in the tag. The key idea is that searching a value in the cache proceeds as a sort of dialogue, in which the searcher gradually assembles the tag associated with the value sought.

Suppose that we want the value of a variable X in the presence of a large number of dimensions, most of which will turn out to be irrelevant.

The first step is to ask (optimistically) for the value of X with an empty tag. The hope here is that X is a constant.

If the cache provides a value—say, 42—then X is indeed the constant 42 and the search concludes successfully.

If the cache finds no entry at all with the empty tag, then we conclude that nothing about X has yet been stored in the cache and we must proceed with the calculation.

The third possibility is that the cache returns a dimension (i.e., a name, not a coordinate). Suppose that the name returned is "z". This means that we must provide the value of z before we can retrieve any more information about X from the cache.

Suppose that the current value of z is 8. We ask for the cache for information about X tagged with $z = 8$. Once again, there are three possibilities.

The first possibility is that the cache returns a value, say, 64. This means that X has the value 64 whenever z has the value 8, and our search is successful.

The second possibility is that there is no entry for X with tag $z = 8$. This means the value we need is yet to be computed, and we must proceed with the computation (adding appropriate entries to the cache as we do so).

The third possibility is that the cache returns another dimension—say, u. This means that when $z = 8$ we also need the value of u to get a value of X.

So we look up the value of u—say, 14—and consult the cache for information about X with tag $z = 8 \wedge u = 14$. Once again we either get a value, or no result and proceed to the computation, or else get the name of another dimension whose value will be required.

In this way we eventually either (1) build up the tag that retrieves the required value or (2) learn that we will have to compute it.

11 Dimensions as first-class values

At the programming-language design level, the natural next step was to introduce dimensions as first-class values (see reference [13] for a full history of this topic.) This challenge was taken up by Joey Paquet and author Plaice in Tensor Lucid [9, 10], developed to write tensor equations naturally. Tensor Lucid allowed declared dimensions to be used as ordinary values, making accessible to the programmer the total dimensionality of an object. However, all dimensions still had to be created lexically, and could not be created on demand, during execution.

The counterpart of dimensions as values, i.e., values as dimensions, was the next step. This was done by author Plaice in Multidimensional Lucid [11], in which any ground value could be used as a dimension. Ultimately, the difference with Indexical Lucid was that the dimensions need not always be identifiers, hence could be created on the fly, opening up many new possibilities.

Further discussion even led to discussion of contexts as first-class values. Nevertheless, 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 these languages would have been of limited utility.

12 The TransLucid project

The TransLucid project (`translucid.web.cse.unsw.edu.au`) was initiated after a meeting between authors Plaice and Wadge in late 2005. We discussed the design and implementation of an extension of Lucid that would allow dimensions to be treated as first-class values, and in which any ground value, such as a string or an integer, could be used as a dimension.

To illustrate this idea, consider the Ackermann function, defined below as a two-dimensional table *ack*, varying in dimensions 0 and 1:

$$\begin{aligned} ack = {} & \texttt{if \#}.1 \equiv 0 \texttt{ then \#}.0 + 1 \\ & \texttt{elsif \#}.0 \equiv 1 \texttt{ then } ack \texttt{ @ } [1 \leftarrow \texttt{\#}.1 - 1, 0 \leftarrow 1] \\ & \texttt{else } ack \texttt{ @ } \left[1 \leftarrow \texttt{\#}.1 - 1, 0 \leftarrow ack \texttt{ @ } [0 \leftarrow \texttt{\#}.0 - 1]\right] \texttt{ fi} \end{aligned}$$

	dim 0 $\rightarrow$						
ack	0	1	2	3	4	5	$\cdots$
dim 1 $\downarrow$ 0	1	2	3	4	5	6	$\cdots$
1	2	3	4	5	6	7	$\cdots$
2	3	5	7	9	11	13	$\cdots$
3	5	13	29	61	125	253	$\cdots$
4	13	65533	$\cdots$				
5	65533	$\cdots$					
$\vdots$	$\ddots$						

Since *ack* is an infinite table, its evaluation can only be partial: this is done with respect to a context—written `#` in the definition—which is a function mapping *dimensions* to *ordinates*; the expression '`#`.1' means the application of the context `#` to dimension 1, in order to return the current 1-ordinate.

The context of evaluation can be changed using the `@` operator by specifying the new ordinates for some dimensions. In the definition of *ack*, the `@` operator is used three times. In expression '$0 \leftarrow ack$ `@` $[1 \leftarrow \texttt{\#}.1 - 1, 0 \leftarrow 1]$', the change of context is specified in a *relative* manner, specifying that the 1-ordinate is to be decremented, and the 0-ordinate set to 1. In expression '$0 \leftarrow ack$ `@` $[0 \leftarrow \texttt{\#}.0 - 1]$', the change of context is specified *partially*, giving only a change for the 0-ordinate, implying that the 1-ordinate remains unchanged.

The original TransLucid allowed the definition of a system of mutually recursive equations defining arbitrary-dimensional arrays of arbitrary extent, where the dimensions were all ground values. Using this simple language, a number of experiments at building cache-based evaluators along the lines presented in the previous section were undertaken. The first implementation is described in [14]: evaluation of an expression is undertaken in a leftmost, depth-first manner. Another approach is presented in [15], where different subexpressions are evaluated separately in different threads, and evaluation of one thread blocks on the cache access should it need a result currently being computed by another thread.

The experiments demonstrated that the proposed implementation technique was effective. However, the original TransLucid was unusable, as it had no functions, nor could one introduce dimension identifiers.

Resolving these problems was non-trivial, and required developments at all levels: programming language design, denotational semantics and implementation. This work was undertaken by authors Plaice and Beck and is summarized in reference [12]. The language described therein corresponds to the implementation available at `translucid.web.cse.unsw.edu.au`.

13 Abstractions and `where` clauses

The three standard Lucid functions, *first*, *next* and *fby*, become in TransLucid:

$$\begin{aligned}
&\texttt{fun } \mathit{first}.d\ X = X\ \texttt{@}\ [d \leftarrow 0]\\
&\texttt{fun } \mathit{next}.d\ X = X\ \texttt{@}\ [d \leftarrow \texttt{\#}.d + 1]\\
&\texttt{fun } \mathit{fby}.d\ X\ Y = \texttt{if } \texttt{\#}.d \equiv 0\ \texttt{then } X\ \texttt{else } Y\ \texttt{@}\ [d \leftarrow \texttt{\#}.d - 1]\ \texttt{fi}
\end{aligned}$$

They could also have been defined as ($\rightarrow$ is right-associative):

$$\begin{aligned}
&\texttt{var } \mathit{first} = \lambda^{\mathrm{b}} d \rightarrow \lambda^{\mathrm{n}} X \rightarrow X\ \texttt{@}\ [d \leftarrow 0]\\
&\texttt{var } \mathit{next} = \lambda^{\mathrm{b}} d \rightarrow \lambda^{\mathrm{n}} X \rightarrow X\ \texttt{@}\ [d \leftarrow \texttt{\#}.d + 1]\\
&\texttt{var } \mathit{fby} = \lambda^{\mathrm{b}} d \rightarrow \lambda^{\mathrm{n}} X \rightarrow \lambda^{\mathrm{n}} Y \rightarrow \texttt{if } \texttt{\#}.d \equiv 0\ \texttt{then } X\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } Y\ \texttt{@}\ [d \leftarrow \texttt{\#}.d - 1]\ \texttt{fi}
\end{aligned}$$

which makes it clear that these are higher-order, Curried functions, like functions in most existing functional languages. Note that *fby* in TransLucid is written in prefix style, not infix.

The function *first* takes two parameters, a *base parameter* d, and a *named parameter* X. The latter is assumed to vary in the dimension d, and the body is evaluated in the context in which the function is applied, thereby pulling the zeroth element of X in dimension d.

The base parameter is passed by value, and the body of the corresponding abstraction is *not* evaluated with respect to the application context. The named parameter, on the other hand, is passed by name, and the body of the corresponding abstraction *is* evaluated with respect to the application context.

The function *next* also takes a base parameter d and a named parameter X. It shifts all of X one step "to the left".

The function *fby* takes three parameters, one base parameter d, and two named parameters, X and Y. It shifts Y one slot "to the right" and inserts the zeroth element of X.

If $A = \langle a_0, a_1, a_2, \ldots\rangle$ and $B = \langle b_0, b_1, b_2, \ldots\rangle$ are s-streams, then

	dim s $\rightarrow$						
	0	1	2	3	4	5	$\cdots$
first.s A	a_0	a_0	a_0	a_0	a_0	a_0	$\cdots$
next.s A	a_1	a_2	a_3	a_4	a_5	a_5	$\cdots$
fby.s A B	a_0	b_0	b_1	b_2	b_3	b_4	$\cdots$

There is a third kind of parameter, the *value parameter*, which is passed by value, but whose corresponding abstraction *is* evaluated with respect to the application context. For example,

$$\texttt{fun } \mathit{index}!d = \texttt{\#}.d + 1$$

also writable as:

$$\texttt{var } \mathit{index} = \lambda^{\mathrm{v}} d \rightarrow \texttt{\#}.d + 1$$

gives one plus the current d-ordinate, whatever d might be. Clearly, for this function to be useful, the body '$\texttt{\#}.d + 1$' must be evaluated in the application context, not in the abstraction context.

	dim s $\rightarrow$						
	0	1	2	3	4	5	$\cdots$
index!s	1	2	3	4	5	6	$\cdots$

We illustrate the use of the above functions to define Ackermann as a function taking two base parameters:

$$\begin{aligned}
&\texttt{fun } \mathit{ack}.m.n = A\\
&\texttt{where}\\
&\quad \texttt{dim } d_m \leftarrow m\\
&\quad \texttt{dim } d_n \leftarrow n\\
&\quad \texttt{var } A = \mathit{fby}.d_m\ (\mathit{index}!d_n)\\
&\qquad\qquad\qquad \big(\mathit{fby}.d_n\ (\mathit{next}.d_n\ A)\ (A\ \texttt{@}\ [d_n \leftarrow \mathit{next}.d_m\ A])\big)\\
&\texttt{end}
\end{aligned}$$

The *ack* function has been defined using a *dimensionally abstract* `where` *clause*, in which two dimension identifiers, d_m and d_n, are introduced. When the function is applied, two unused dimensions are allocated to identifiers d_m and d_n, and the context of evaluation of A takes place with the ordinate of d_m's dimension set to m and the ordinate of d_n's dimension set to n. Note the replacement of all but one explicit dimension manipulation by the use of functions *index*, *next* and *fby*.

There are also situations in which the body needs to be evaluated with respect to part of the context in which an application is created. Consider, for example, the function *pow*, which takes argument n and returns the n-th–power function, i.e., $pow.n.m$ yields the value m^n:

$$
\begin{array}{l}
\texttt{fun}\ pow.n = P \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ P = fby.d\ (\lambda^{\mathrm{b}}\ m \to 1)\ (\lambda^{\mathrm{b}}\ \{d\}\ m \to m \times P.m) \\
\texttt{end}
\end{array}
$$

The variable P is a d-stream of power functions:

	dim d $\to$			
P	0	1	2	$\cdots$
	$\lambda^{\mathrm{b}}\ m \to m^0$	$\lambda^{\mathrm{b}}\ m \to m^1$	$\lambda^{\mathrm{b}}\ m \to m^2$	$\cdots$

The explicit '$\{d\}$' in the second λ^{b} abstraction in the definition of P ensures that the d-ordinate needed to evaluate P within the abstraction is frozen at the time of creation of the abstraction.

Finally, there is a fourth kind of abstraction, the *intension abstraction*, which allows the ordinates of certain dimensions to be frozen in an expression to be evaluated later. For example, the variable

$$\texttt{var}\ tempAtLocation = \uparrow\{location\}\ temperature$$

will provide the temperature for a fixed location, once it is evaluated in a context defining dimension *location*. For example

$$tempAtLocation\ \texttt{@}\ [location \leftarrow Paris]$$

is the temperature in Paris; note that this expression still varies in all other dimensions, such as, for example, *date* and *time*.

The implementation of functions and of `where` clauses takes place using additional dimensions, hidden from the user. Each base parameter, value parameter and dimension identifier is implemented with a separate dimension. As for named-parameter abstractions, they are not primitive, and are implemented using intension and value-parameter abstractions.

14 Caching TransLucid

Because of the plethora of dimensions used in TransLucid, both for programming and implementation purposes, it was essential to develop a cache-based interpreter. The implementation we describe in this section is for a maximally parallel interpreter with a centralized cache. The rules given below correspond closely to the behavior of the actual interpreter.

14.1 Assumptions

The cache-based implementation does not fully implement the TransLucid defined in the denotational semantics. Rather, it is assumed that a static semantics is applied to TransLucid programs, and only those passing are implementable using a cache:

1. The denotational semantics for TransLucid allows the context to be passed around explicitly, simply by writing `#`. In the cache-based implementation, the context must always appear in an expression of the form `#`.E, i.e., the context may only appear in a context-query situation; an arbitrary context may not be passed from one part of the program to another. In addition to simplifying the implementation, this choice can also be defended from a security point of view: the context can only reveal the ordinates of dimensions computed explicitly within the program.

2. (In the abstract syntax defined below, we will see that `where` clauses are split into `wheredim` and `wherevar` clauses, in order to facilitate the separate manipulation of dimension and variable identifiers.) The denotational semantics generates different dimensions for a given local dimension identifier each time that the local `wheredim` clause in which it is declared is entered. In the cache-based implementation, each local dimension identifier is mapped to the same dimension upon each entry, i.e., the `wheredim` clause must be defined in such a way that the different entries into the clause must be such that a single dimension is suitable. For this choice to work, a static semantics must ensure that the corresponding `wheredim` clause satisfies the following constraints:

   - The evaluation of a `wheredim` clause in one context cannot depend on the evaluation of the same `wheredim` clause in another context. To ensure that this is the case, the transitive closure of the dependency graph for the nodes in the abstract syntax tree must not contain any loops for `wheredim` clauses.
   - No `wheredim` clause can return an abstraction that varies in a local dimension identifier defined in that `wheredim` clause. To ensure that this is the case, if a local dimension identifier appears in the rank of the body of an abstraction, then that local dimension identifier must appear in the list of frozen dimensions for that abstraction.

   In addition to simplifying the implementation, this choice is consistent with the fact that we have yet to find a useful program in which the same dimension identifier gets mapped to multiple dimensions.

With respect to the presentation, the rules given below will only be for a restricted subset of TransLucid, in which there are no user-defined abstractions and the only applications are for base-parameter abstractions. Figure 1 gives the abstract syntax we work with.

$$
\begin{array}{rll}
E ::= & x & \textit{identifier} \\
| & {}^{m}c & \textit{m-ary constant symbol, } m \in \mathbb{N} \\
| & [E \leftarrow E, \ldots] & \textit{tuple builder} \\
| & \texttt{\#}.E & \textit{context query} \\
| & E\,.\,(E, \ldots) & \textit{base application} \\
| & \texttt{if } E \texttt{ then } E \texttt{ else } E \texttt{ fi} & \textit{conditional} \\
| & E \texttt{ @ } E & \textit{context perturbation} \\
| & E \texttt{ wheredim } \phi_x \leftarrow E, \ldots \texttt{ end} & \textit{local dimensions} \\
| & E \texttt{ wherevar } x = E, \ldots \texttt{ end} & \textit{local variables}
\end{array}
$$

Figure 1: Syntax of TL expressions

The constant symbols correspond to the constants and functions provided by host language. Their meaning will be provided by an interpretation ι in the rules below. Note that in the `wheredim` clause, the dimension identifier has been replaced by a fixed dimension ϕ_x.

14.2 Threads

Given the increasing availability of parallelism in the computing infrastructure, at both the fine-grain and the coarse-grain levels, the rules provided below are to designed to encourage maximal parallelism.

Suppose we have an expression E with n subexpressions E_i, $i = 1..n$. Then the calculation of expression E will take place using a thread w, and the calculation of each of the subexpressions E_i will take place using a thread w_i. The threads are labeled with finite lists such that the initial thread is the empty list, and $w_i = w \cdot i$ consists of appending i to the list w. We write $w' \leq w$ to mean that thread w' is an ancestor to thread w, i.e., that w' is a prefix of w.

14.3 Clocks

Each thread and the cache will have its own clock, which is simply a natural number. Each time that a thread uses the cache, the clocks of both are set to the fastest clock. Each time that a thread uses the results from other threads, its clock is advanced to the fastest clock. All clocks start from zero.

14.4 The cache

A cache β is a synchronous, reactive machine with four internal variables:

- β.ck $\in \mathbb{N}$ is a clock (a counter), initially 0;
- β.age $\in \mathbb{N}$ is the *global retirement age*, initially 2, for the garbage collector;
- β.data contains the *nodes* of the cache;
- β.limit contains the *maximum number of nodes* in the cache.

The rules are defined in such a way that variable β.data holds a decision tree for each variable, keeping track of the dimensions that are needed to access a value. For example, below there are n intermediate decisions to reach a particular value v, where $\Delta_1, \ldots, \Delta_n$ are mutually exclusive sets of dimensions:

$$\begin{aligned}
\beta.\mathrm{data}(x, \emptyset) &= \Delta_1 \\
\beta.\mathrm{data}(x, \kappa \lhd \Delta_1) &= \Delta_2 \\
\beta.\mathrm{data}\big(x, \kappa \lhd (\Delta_1 \cup \Delta_2)\big) &= \Delta_3 \\
&\ldots \\
\beta.\mathrm{data}\big(x, \kappa \lhd (\Delta_1 \cup \cdots \cup \Delta_{n-1})\big) &= \Delta_n \\
\beta.\mathrm{data}\big(x, \kappa \lhd (\Delta_1 \cup \cdots \cup \Delta_n)\big) &= v
\end{aligned}$$

Each node γ_j, $j \in \mathbb{N}$, in β.data is either a leaf node storing a value v or a value $\texttt{calc}\langle w \rangle$, meaning that thread w is currently responsible for computing this entry; or an internal node storing a pair consisting of a set Δ of dimensions and sets of ordinates for those dimensions that have entries.

The cache uses the retirement plan mentioned in Section 5. Each node γ_j has an age, γ_j.age, initialized to zero when the node is created. Every time that a node is retrieved, its age is reset to zero. The garbage collector keeps a global retirement age, and every garbage collection run, the global retirement age is decreased by one. If, at any point, a node whose age is greater than the global retirement age is retrieved, the global retirement age is increased to the age of the node before that node is set back to zero. Every garbage collection run, the age of every node is increased by one. Only leaf nodes can be collected. Should, during any one garbage collection run, an internal node have all of its children collected, then it can be collected immediately if its age is greater than the retirement age.

The variable β.data must be initialized with the entries for all of the inputs (free variables) of the expression to be evaluated.

The machine responds to two different instructions generated by threads, and one instruction generated internally:

- $(v', t') = \beta.\text{find}(x, \kappa, w, t)$
  - If $t > \beta.\text{ck}$, advance $\beta.\text{ck}$ to t.
  - If $\beta.\text{data}(x, \kappa) = \texttt{calc}\langle w' \rangle$ and $w' \leq w$, hang (do not return).
  - If $\beta.\text{data}(x, \kappa)$ is not defined, let $\beta.\text{data}(x, \kappa)$ be $\texttt{calc}\langle w \rangle$ and advance $\beta.\text{ck}$ by 1. Should β receive more than one $\beta.\text{find}(x, \kappa, w_i, t_i)$ instruction at the same instant $\beta.\text{ck}$, the w is randomly chosen from the w_i.
  - For all the nodes γ_j in the chain which stores $\beta.\text{data}(x, \kappa)$, if $\gamma_j.\text{age} > \beta.\text{age}$, then set $\beta.\text{age} = \gamma_j.\text{age} + 1$, then set $\gamma_j.\text{age} = 0$.
  - If the number of nodes is greater than $\beta.\text{limit}$, then run $\beta.\text{collect}()$.
  - Return $(\beta.\text{data}(x, \kappa), \beta.\text{ck})$.
- $(v', t') = \beta.\text{add}(x, \kappa, w, t, v)$
  - If $t > \beta.\text{ck}$, advance $\beta.\text{ck}$ to t.
  - If $\beta.\text{data}(x, \kappa) \neq \texttt{calc}\langle w \rangle$, hang (do not return).
  - Let $\beta.\text{data}(x, \kappa)$ be v and advance $\beta.\text{ck}$ by 1.
  - Return $(\beta.\text{data}(x, \kappa), \beta.\text{ck})$.
- $\beta.\text{collect}()$
  - $\beta.\text{age} = \beta.\text{age} - 1$
  - For each node γ_j in the cache, in a post-order traversal of the tree:
    - If $\gamma_j.\text{age} \geq \beta.\text{age}$ then remove the node if it has no children, and if it is not holding the value $\texttt{calc}\langle w \rangle$.
    - If γ_j has not been collected, increment $\gamma_j.\text{age}$ by one.

14.5 The rules

The rules on the next two pages are of the form

$$(v', t') = [\![E]\!]\iota\xi\kappa\Delta\beta wt$$

meaning that expression E evaluates in interpretation ι, environment ξ (mapping identifiers to expressions), context κ, known dimensions Δ, cache β, thread w with current clock t to the pair (v', t'), where v' is a value and $t' \geq t$. The value v' can be an ordinary value or a set of dimensions Δ' needed to compute this expression.

Comments on the rules

All of the evaluation rules that involve some computation of subexpressions take into account the fact that any of those subexpressions might evaluate to a demand for one or more dimensions. Should this occur, computation does not continue, and all of these demands are aggregated into a larger demand, which becomes the result.

Rules (1)–(2) correspond to the evaluation of constants. They evaluate to the same value in all contexts and do not interact with the cache in any manner.

Rules (3)–(6) are trivial modifications of their equivalent rules from the denotational semantics. They simply pass on demands for dimensions resulting from the evaluation of subexpressions.

$$
\begin{aligned}
&[\![d]\!]\iota\xi\kappa\Delta\beta wt && (1)\\
=\ &(d,t)
\end{aligned}
$$

$$
\begin{aligned}
&[\![{}^m c]\!]\iota\xi\kappa\Delta\beta wt && (2)\\
=\ &(\iota({}^m c),t)
\end{aligned}
$$

$$
\begin{aligned}
&[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]^\circ\iota\xi\kappa\Delta\beta wt && (3)\\
=\ &\text{let } (d_{ij},t_{ij}) = [\![E_{ij}]\!]\iota\xi\kappa\Delta\beta w_{ij}t\\
&\text{in } \begin{cases} \big(\bigcup_{ij}\Delta_{ij}, \max(t_{ij})\big), & d_{ij} \text{ is of form } \Delta_{ij}\\ \big(\{d_{i0} \mapsto d_{i1}\}, \max(t_{ij})\big), & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \,.\, (E_i)_{i=1..m}]\!]^\circ\iota\xi\kappa\Delta\beta wt && (4)\\
=\ &\text{let } (d_i,t_i) = [\![E_i]\!]\iota\xi\kappa\Delta\beta w_i t\\
&\text{in } \begin{cases} \big(\bigcup_{i}\Delta_{i}, \max(t_{i})\big), & d_i \text{ is of form } \Delta_i\\ d_0(d_i)\Delta\beta w(\max(t_i)), & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\xi\kappa\Delta\beta wt && (5)\\
=\ &\text{let } (d_0,t_0) = [\![E_0]\!]\iota\xi\kappa\Delta\beta wt\\
&\text{in } \begin{cases} (\Delta_0,t_0), & d_0 \text{ is of form } \Delta_0\\ [\![E_1]\!]\iota\xi\kappa\Delta\beta wt_0, & d_0 \equiv \mathit{true}\\ [\![E_2]\!]\iota\xi\kappa\Delta\beta wt_0, & d_0 \equiv \mathit{false}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\xi\kappa\Delta\beta wt && (6)\\
=\ &[\![E_0]\!]\iota(\xi[x_i/E_i])\kappa\Delta\beta wt
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{\#}.E_0]\!]\iota\xi\kappa\Delta\beta wt && (7)\\
=\ &\text{let } (d_0,t_0) = [\![E_0]\!]\iota\xi\kappa\Delta\beta wt\\
&\text{in } \begin{cases} (\Delta_0,t_0), & d_0 \text{ is of form } \Delta_0\\ (\{d_0\},t_0), & d_0 \notin \Delta\\ (\kappa(d_0),t_0), & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \texttt{ @ } E_1]\!]^\circ\iota\xi\kappa\Delta\beta wt && (8)\\
=\ &\text{let } (d_1,t_1) = [\![E_1]\!]\iota\xi\kappa\Delta\beta wt\\
&\text{in } \begin{cases} (\Delta_1,t_1), & d_1 \text{ is of form } \Delta_1\\ [\![E_0]\!]\iota\xi(\kappa \dagger d_1)(\Delta \cup \mathrm{dom}(d_1))\beta wt_1, & \text{otherwise}\end{cases}
\end{aligned}
$$

Rule (7), for context queries, is where demands for dimensions are generated. Any time the context is looked up, the requested dimension must be available in Δ. If it is not in Δ, then a demand is generated for that dimension.

Rule (8), for context perturbations, adds to Δ any dimensions whose ordinate has been changed. This is done because any dimension that is changed cannot possibly affect the value of an identifier being cached further up the tree. The dimension is then added to Δ so that if its value is later requested, a demand for it is not generated.

$$
\begin{aligned}
&[\![x]\!]\iota\xi\kappa\Delta\beta wt \qquad (9)\\
=\ &\text{let } (d_0,t_0) = [\![x]\!]^1\iota\xi\kappa\emptyset\beta wt\\
&\text{in } \beta.\text{find}(x,\kappa \lhd \Delta,t,w)
\end{aligned}
$$

$$
\begin{aligned}
&[\![x]\!]^1\iota\xi\kappa\Delta\beta wt \qquad (10)\\
=\ &\text{let } (d_0,t_0) = [\![x]\!]^2\iota\xi\kappa\Delta\beta wt\\
&\text{in } \begin{cases} [\![x]\!]^1\iota\xi\kappa(\Delta\cup\Delta_0)\beta wt, & d_0 \text{ is of form } \Delta_0 \text{ and } \Delta_0\subseteq\text{dom}(\kappa)\\ (d_0,t_0), & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![x]\!]^2\iota\xi\kappa\Delta\beta wt \qquad (11)\\
=\ &\text{let } (d_0,t_0) = \beta.\text{find}(x,\kappa \lhd \Delta,t,w)\\
&\text{in } \begin{cases} \text{let } (d_1,t_1) = [\![\xi(x)]\!]\iota\xi\kappa\Delta\beta wt_0 & \\ \text{in } \beta.\text{add}(x,\kappa \lhd \Delta,t_1,w,d_1), & d_0 \text{ is of form calc}\langle w'\rangle \text{ and } w'\equiv w\\ [\![x]\!]^2\iota\xi\kappa\Delta\beta w(t_0+1), & d_0 \text{ is of form calc}\langle w'\rangle \text{ and } w'\not\leq w\\ (d_0,t_0), & d_0 \text{ is not of form calc}\langle\cdots\rangle\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0\ \texttt{wheredim}\ \phi_{x_i}\leftarrow E_i\ \texttt{end}_{\,i=1..m}]\!]\iota\xi\kappa\Delta\beta wt \qquad (12)\\
=\ &\text{let } (d_i,t_i) = [\![E_i]\!]\iota\xi\kappa\Delta\beta w_i t\\
&\text{in } \begin{cases} \big(\bigcup_i\Delta_i,\max(t_i)\big), & d_i \text{ is of form } \Delta_i\\ \text{let } (d_0,t_0) = [\![E_0]\!]\iota\xi\big(\kappa\ \dagger\ \{\phi_{x_i}\mapsto d_i\}\big)\Delta\beta w(\max(t_i)) & \\ \text{in } (d_0\setminus\{d_i\},t_0), & \text{otherwise}\end{cases}
\end{aligned}
$$

Rules (9)–(11) are where the interaction with the cache takes place. Rule (10) probes the cache in an iterative manner, starting with an empty Δ as given by rule (9), and adding any dimensions demanded at each iteration. Rule (11) interacts with the cache given a particular Δ. First it finds a value in the cache; there are three possibilities depending on what is returned:

1. if the current thread is asked to compute the value, it computes the value and adds the answer to the cache;
2. if another thread, that is not an ancestor of the current thread, is computing the value, then wait in a busy loop, incrementing the clock by 1;
3. otherwise, simply return the answer.

There is a fourth possibility, not appearing in the rules. Should a thread that is an ancestor of the current thread be computing the value, then there is a loop, and the cache will hang, meaning that the rules do not provide a solution.

Rule (12): Because of the restriction imposed on `wheredim` clauses, as presented in the assumptions, the `wheredim` clause acts like a context perturbation with a unique local dimension.

15 Conclusion

The design of the Lucid programming language in the mid-1970s allowed the clear presentation of programs in a declarative style, without the need for writing complex data constructs. This was understood by such luminaries as Alan Kay, the inventor of Smalltalk [7]:

> One of my favorite old languages is one called Lucid by [Bill Wadge and] Ed Ashcroft. It was a beautiful idea. He said, "Hey, look, we can regard a variable as a stream, as some sort of ordered thing of its values and time, and use Christopher Strachey's idea that everything is wonderful about tail recursion and Lisp, except what it looks like." [...]
>
> Strachey said, "I can write that down like a sequential program, as a bunch of simultaneous assignment statements, and a loop that makes it easier to think of." That's basically what Lucid did—there is no reason that you have to think recursively for things that are basically iteration, and you can make these iterations as clean as a functional language if you have a better theory about what values are.

However, it became clear that only with the full addition of multidimensionality, higher-order functions and of a cache-based interpreter, would Lucid become a viable language. With the results presented in this paper, with the demand-driven cache for TransLucid, we have now achieved all of these goals. We envisage that it is now possible to do large-scale programming in TransLucid, and to move beyond interpreters to compilers generating efficient code.

References

[1] Edward A. Ashcroft, Tony Faustini, Jaggan Jaganannathan, and William W. Wadge. *Multidimensional Declarative Programming*. Oxford University Press, New York, 1995.

[2] Edward A. Ashcroft and William W. Wadge. Program Proving Without Tears. Technical Report CS-75-03, Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1975. Presented to the Symposium on Proving and Improving Programs, G. Huet and G. Kahn (ed.), held in Arc-et-Senans, France, 1–3 July 1975, pp.99–114, IRIA, `http://www.cs.uwaterloo.ca/research/tr/1975/CS-75-03.pdf`.

[3] Edward A. Ashcroft and William W. Wadge. Lucid—A Formal System for Writing and Proving Programs. *SIAM Journal of Computing*, 5(3):336–354, 1976.

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

[5] Antony A. Faustini and William W. Wadge. An eductive interpreter for the Language pLucid. In *Proceedings of the ACM Symposium on Interpreters and Interpretive Techniques*, pages 86–91, 1987.

[6] Antony A. Faustini and William W. Wadge. An eductive interpreter for Lucid. In Richard L. Wexelblat, editor, *PLDI*, pages 86–91. ACM, 1987.

[7] Stuart I. Feldman and Alan C. Kay. A conversation with Alan Kay. *ACM Queue*, 2(9):20–30, 2004.

[8] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

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

[10] Joey Paquet and John Plaice. *The semantics of dimensions as values*, pages 259–273. World-Scientific, Singapore, 2000.

[11] John Plaice. Multidimensional Lucid: Design, semantics and implementation. In Peter G. Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors, *DCW*, volume 1830 of *Lecture Notes in Computer Science*, pages 154–160. Springer, 2000.

[12] John Plaice and Jarryd P. Beck. Higher-order Multidimensional Programming. Report UNSW-CSE-TR-2012-15, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2012. Submitted for publication. `http://plaice.web.cse.unsw.edu.au/archive/JAP/U-CSE-201215.pdf`.

[13] John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Mathematics in Computer Science*, 2(1):37–61, 2008.

[14] John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential Demand-Driven Evaluation of Eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1266–1271, 2008.

[15] Toby Rahilly and John Plaice. A Multithreaded Implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2008, Turku, Finland, July 28 – August 1*, pages 1272–1277. IEEE Computer Society, 2008. 1st IEEE International Workshop on Software Engineering for Context Aware Systems and Applications (SECASA 2008).

[16] Panos Rondogiannis and William W. Wadge. Higher-Order Functional Languages and Intensional Logic. *Journal of Functional Programming*, 9(5):527–564, 1999.

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