Indexical Lucid

by

A.A. Faustinia
Department of Computer Science
Arizona State University
Tempe
Arizona 85287

R. Jagannathan
Computer Science Laboratory
SRI International
Menlo Park
California 94025

Abstract

Indexical Lucid is a programming language based on Ashcroft and Wadge's Lucid. It differs from Lucid in two ways. Firstly, it is a multi-dimensional language in which the user introduces new dimensions into a program through the use of indexical where clauses. Secondly, Indexical Lucid does not include the Lucid "is current" construct and so cannot be thought of as a true extension to original Lucid. We show that a multi-dimensional language like Indexical Lucid does not need an "is current" construct because the indexical where clauses are sufficient to specify "iterative" subcomputations.

Introduction

Two views of Lucid exist in the computer science community. The oldest view is that Lucid is a language for writing and proving properties about programs. This view is still valid even though the language has evolved considerably since the earliest publications [AsWa76, AsWa77]. The second and more recent view is that Lucid is a stream-oriented (dataflow) language. This view is largely based on the publication of the book "Lucid the dataflow programming language" [WaAs85]. This second view has been taken by many as a consummation or final definition of the language. This is certainly not the case. In this paper we examine some of the more recent advances in the development of Lucid. Other developments such as higher order Lucid are covered by other papers in this symposium.

From a historical perspective the ideas in this paper can be traced back to the chapter entitled "Beyond Lucid" in [WaAs85] and to a paper by E.A. Ashcroft on massive parallelism in Lucid [Ash85]. Our ideas differ from those originally envisioned in early papers and is more closely related to more recent work by Du and Wadge [WaDu88, WaDu90] at the University of Victoria, Canada.

Originally, multi-dimensional Lucid objects were defined and manipulated using the operators "initial", "rest" and "cby" [WaAs85]. This was elegant since it complemented the "first", "next" and "fby" of stream oriented Lucid. Moreover, the proposed operators were information preserving. In other words, a multi-dimensional object could be defined and manipulated without any of the components being lost. This is not true for streams built with the operators "first", "next" and "fby".

Unfortunately, this approach to multi-dimensionality was difficult to use by those trying to write programs. The main reason for the difficulty was that the
programmer had to think of the multi-dimensional object monolithically. The operators preserved information by manipulating data in a nested hyperspace. For example, the first argument of the dyadic operator "cby" might itself be a multi-dimensional object that would be "twisted" by the operator "cby" into a higher dimension to preserve it. The operator "initial" would then perform an "untwisting" operation to extract the data into its original "shape". This approach, may have validity, but for the moment it is too difficult for programmers to use such operators in the solution of simple multi-dimensional problems.

The work described in this paper and that of Du and Wadge is based on a distributed view of multi-dimensionality. In this view, a multi-dimensional space is composed of multiple orthogonal dimensions. In this view each dimension has associated with it operators for defining data on a per dimension basis.

A simple extension to the Lucid stream operators is sufficient to deal with this type of multi-dimensional space. Unfortunately, this means that we need similar but distinct operators for each new dimension. These operators are easily defined; they are simply definitions similar to "first", "next" and "fby" but for particular dimensions rather than for time. In early work on this approach to multi-dimensionality the operators were absolute and predefined [FaWa87, AFJ91]. That is, the programmer was given a set of operators for manipulating each dimension. Moreover, the dimensions themselves were absolute pre-existing dimensions often called space dimension 0, 1, 2, etc. Associated with each dimension was a corresponding set of appropriately named Lucid operators for the dimension. The operators "initial0", "succ0" and "sby0" were used to define objects varying in the first dimension, "initial1", "succ1" and "sby1" the second dimension and so on. This is almost like having a programming language in which a programmer is only allowed to use the identifiers "var0", "var1" and so on. It is possible to program in such a language but the lack of meaningful names makes such a language difficult to understand. Indexical Lucid enables the programmer to specify dimension names so that multi-dimensional programs are themselves more easily read. This is important syntactic sugar.

**Indexical Programming**

Lucid is a simple example of an indexical language. A variable in a Lucid program is not associated with a memory location as it is in a procedural programming language. A Lucid variable is associated with a temporal stream. In other words a Lucid variable takes on different values at different points in "time". A time stream in Lucid is not a real-time stream it is a time index. In other words there is the first value of the variable, the second and so on. The following pair of equations define the variable even :-

\[
\begin{align*}
\text{first even} &= 2 \\
\text{next even} &= \text{even}+2
\end{align*}
\]

The first equation defines the first or initial value of the even stream (2) and the second equation the successive values of even in terms of the current value of even.
plus two.\(^1\) We can think of the above equations as equivalent to the following recurrence relation :-

\[
\begin{align*}
t = 0 & \quad \text{[even]}_t = 2 \\
t > 0 & \quad \text{[even]}_{t+1} = \text{[even + 2]}_t
\end{align*}
\]

The \(t\) in the recurrence relation is the implicit index or time context of Lucid. In the Lucid equations there is no explicit mention of \(t\) or time other than the first equation that states that the initial value of \(\text{even}\) is 2 and yet the above Lucid equations define the stream of values \(<2, 4, 6, 8, \ldots>\) without the explicit use of indices (in this sense was can think of Lucid as an intensional language [FaWa87]).

In original Lucid the implicit index was used simply as the time or iteration index. Consequently, a multi-dimensional problem would have to be solved using time and some form of monolithic data structure such as a list or string [FMY84]. Multi-dimensional problems can be expressed directly in indexical Lucid without the need for monolithic data structures. This has important implications for parallel and distributed implementations. Multi-dimensional problems are found in many applications areas especially in scientific computation and computer graphics.

The rest of this paper deals with the "operational" details of indexical Lucid and how to use indexical constructs to solve problems. A syntax and semantics for Indexical Lucid is included in Appendix A. This rather concise format is based on a technical report written by E.A. Ashcroft [Ash84] that gave the syntax and semantic for original Lucid.

### An Indexical Extension to Lucid

The simplest way of extending the number of indices available to the Lucid programmer is to redefine operators for all possible dimensions. This was explained in the previous section (initial0, initial1, etc.). A more pleasing approach is to allow the user to extend the dimensions explicitly giving appropriate names to the new dimensions as and when they are needed. One approach to providing such a mechanism is to extend the Lucid where-clause\(^2\).

An indexed where clause is one that contains declarations of new indices through use of an index declaration. The scope of the indices introduced by the index declaration is the same as the scope of the variables defined in an ordinary where clause. For example the following indexical where clause introduces two new indices height and width.

---

\(^1\) This syntax comes from the original Lucid papers and is often a better way of expressing the idea of a variable changing with time than the more modern semantically equivalent equation:-

\[
\text{even} = 2 \text{ fby even+2}
\]

The operator \(\text{fby}\) is an infix operator pronounced "followed by" see [FMY84].

\(^2\) The Lucid where-clause is an adaptation of Landin's where-clause [Lan66]
\[ a + b \text{ where} \]
\[
\text{index height, width;}
\]
\[
a = \ldots
\]
\[
b = \ldots
\]
\[
\text{end}
\]

In the above program the variables \(a\) and \(b\) may be defined to vary over two new dimensions \(\text{height}\) and \(\text{width}\). In addition to time that is always implicit in Indexical Lucid. Whether or not \(a\) or \(b\) vary with any of these indices depends on the actual definition of \(a\) or \(b\). For example if \((a = 10;\) then it is a constant, if \((a = 1 \text{ fby a+1;})\) then it varies with time. How then do we make \(a\) or \(b\) vary with \(\text{height}\) or \(\text{width}\)? To do this we need to introduce operators specific to the new dimensions.

All of the regular Lucid operators can be extended to any dimension \(h\) as follows:-

<table>
<thead>
<tr>
<th>Conventional Lucid Operators</th>
<th>Conventional Lucid Operators expanded</th>
<th>Extended Indexical Lucid Operators for Index (h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{first}</td>
<td>\textbf{first.time}</td>
<td>\textbf{first.h}</td>
</tr>
<tr>
<td>\textbf{next}</td>
<td>\textbf{next.time}</td>
<td>\textbf{next.h}</td>
</tr>
<tr>
<td>\textbf{fby}</td>
<td>\textbf{fby.time}</td>
<td>\textbf{fby.h}</td>
</tr>
<tr>
<td>\textbf{upon}</td>
<td>\textbf{upon.time}</td>
<td>\textbf{upon.h}</td>
</tr>
<tr>
<td>\textbf{asa}</td>
<td>\textbf{asa.time}</td>
<td>\textbf{asa.h}</td>
</tr>
<tr>
<td>\textbf{wvr}</td>
<td>\textbf{wvr.time}</td>
<td>\textbf{wvr.h}</td>
</tr>
<tr>
<td>\textbf{@}</td>
<td>\textbf{@.time}</td>
<td>\textbf{@.h}</td>
</tr>
</tbody>
</table>

Note that it is not necessary to write \((\text{first})\) as \((\text{first.time})\) it is included for completeness. Roughly speaking, each of the extended Lucid operators works on its particular dimension in the same way that the ordinary Lucid operators worked on the time dimension. This makes the new operators very easy for the Lucid programmer to use.

Very important from a programming standpoint are the "indexical constants". These are variables with the same names as those used for index names. The variable \(\text{time}\) can be used in the expression part of a program and the value it takes depends on when it is used (an intensional variable). In general the value of \(\text{time} \) at \(\text{time i}\) is \(i\). In other words indexical constants act as locators they tell you where you are in the indexical space. Note that all indices in indexical Lucid vary over the natural numbers.

<table>
<thead>
<tr>
<th>Conventional Lucid Constants</th>
<th>New Constant for Index names (h), size</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{time}</td>
<td>(h), (\text{size})</td>
</tr>
</tbody>
</table>
The following is an example of a simple Indexial Lucid that defines a unit matrix and uses most of the new concepts introduced:-

```lucid
... where
    index i,j;
    unitMatrix = if i == j then 1 else 0 fi;
end;
```

Additional Useful Indexical Operators

Let $h$ and $v$ be two distinct index names and $<expression> . h$ be an arbitrary expression varying in the $h$ dimension.

$h \to v <expression> . h$

This expression enables an expression that varies in the $h$ dimension to vary in the same way but in the $v$ dimension. This can also be expressed as a user defined function in the following way:-

"h to v"(x) = x @ .v h

Binary trees computations (see the paper on tournament computations in these same proceedings) can be mapped onto a two dimensional space using the functions `rightChild` and `leftChild`. To do this we need two dimensions ($h$ and $v$). We can the implement a binary tree computation by iterating along the $v$ dimension computing with pairs of successive values in the $h$ dimension. We will do this for each successive point in the $v$ dimension. The letter "v" is a good choice of name for this dimension because the binary tree computation can be viewed as a "V" shaped computation or an upside down binary tree (or a tournament). To achieve this we need operators that can select successive pair of elements in a dimension, namely :-:

```lucid
rightChild.h <expression> . h
leftChild.h <expression> . h
```

The following are two user defined functions that define these two operators for the time dimension:-

```lucid
rightChild(x) = x @ (2*time+1);
leftChild(x) = x @ (2*time);
```

A program to compute the binary sum of the even numbers using a tournament computation would then be defined a follows:-

```lucid
binarySum
where
    index h;
    data = 2*h;  /* the even sequence
```
on along the h dimension */
  binarySum = data fby leftChild.h binarySum +
              rightChild.h binarySum;
end;

The above binary sum program actually computes the running sum of all even binary
numbers beginning with the first pair. That is, at time 0 (and h=0) the value of
binarySum is 0, at time 1 (and h=0) it is 2, at time 2 (and h=0) it is 12 and so on. In
order to get a single answer, say the sum of the first 8 even numbers we need to
extract the appropriate value by using an operator like @. The expression binarySum
@ 3 is constant with respect to time and is the binary sum at depth 3 in the binary
tree, this is the value we want to select. The following is the complete program:

binarySum @ 3
  where
    index h;
    data = 2*h; /* the even sequence
    on along the h dimension */
      binarySum = data fby leftChild.h binarySum +
                  rightChild.h binarySum;
    end;

An Example Program

The following is an example of an indexical program for merge sorting a
sequence of numbers5. This program mergesorts the first 8 entires of data (this is
because we used the expression mergeSort @ 3). The program used the v
dimension to build up the sorted data. This data is then brought into the time
dimension through the operator "v to time".

v to time (mergeSort @ 3)
  where
    index v,h;
    data = (45,33,2,91,790,-3,73,......).h;
    merge(x,y) = if xx <= yy then xx else yy fi;
    where
      xx = x upon.v xx <= yy;
      yy = y upon.v xx > yy;
    end;
    mData = data fby
      merge(leftChild.h mData,rightChild.h mData);
  end;

5 Note the following syntactic shorthand will be used in the example program.
(<expr>,<expr>,......).h is equivalent to
<expr> fby.h <expr> fby.h ...<expr> fby.h ....
We are assuming that the reader is familiar with Lucid operations such as upon and @ [FMY84].

Subcomputation Indexically

In Original Lucid there were two types of where-clauses, one containing "is current" declarations used to define subcomputations [FMY84] and the other was one related to Landin's whererec-clause. In Indexical Lucid we no longer need "is currented" where clauses because "indexed" where clauses can be used to specify subcomputations. The following Lucid program using an example of a computation needing an "is current" subcomputation in Lucid. The following program computes the running root mean square of its input. Note the use of the is current in the definition of sqrtroot. Where clauses with is current declarations implicitly introduces a new dimension into a Lucid program. That is, for each point in time outside an "is currented" where, there corresponds a possibly different set of time sequences for each of the variables in the clause. The indexical version of the same program makes this implicit dimension visible (index 1). In a similar manner "is currented" where clauses nested within other "is currented" clauses introduce a new implicit dimension for each level of nesting. This can again be implemented using indexical where clauses nested in the same manner but explicitly introducing a new dimension of each level of nesting of the "is currented" where clauses. Here are the example programs referred to earlier:-

An program containing an "is currented" where clause

sqroot(avg(square(a)))
where

  square(x) = x*x;
  avg(y) = mean
  where
      n = 2 fby n+1;
      mean = y fby mean + d;
      d = (next y - mean) / n;
  end;
  sqrtroot(z) = approx asa err < 0.0001
  where
      Z is current z;
      approx = Z/2 fby (approx + Z/approx)/2;
      err = abs(square(approx)-Z);
  end

An equivalent program written using an indexical where clause:-

sqroot(avg(square(a)))
where
\[
square(x) = x^2;
\]
\[
\text{avg}(y) = \text{mean}
\]
\[
\text{where}
\]
\[
n = 2 \ fby \ n + 1;
\]
\[
\text{mean} = y \ fby \ \text{mean} + d;
\]
\[
d = (\text{next} \ y - \text{mean}) / n;
\]
end;
\[
\sqrt{\text{root}}(z) = \text{approx asa} \ i \ err < 0.0001
\]
\[
\text{where}
\]
\[
\text{index} \ i;
\]
\[
\text{approx} = z / 2 \ fby \ i \ (\text{approx} + z / \text{approx}) / 2;
\]
\[
\text{err} = \text{abs}(\text{square}(\text{approx}) - z);
\]
end;

Collapsing Dimensions

The introduction of extra dimensions whether implicit, as in Lucid, or explicit, as in Indexical Lucid, means that the language can directly define objects in terms of objects of higher dimensionality. In other words the subject part of an "is current" or indexical where clause is defined in terms of functions and variables inside the body of the where clause and these functions and variables will probably be defined in terms of the extra dimensions introduced by the index declaration (otherwise there would be no point in having the subcomputation). The question then arises as to which of the values associated with the extra dimensionality is used to give meaning to the outside objects?

In Lucid a diagonalization method is used. That is, for each point in time (say i) in the outer world, we associate the ith point in time of the inner world. That is, if a variable v is defined in terms of an "is current" where clause. The value the variable will take on for each point in time, say i, in the outer environment is the value of the subject part of the "is current" where clause at inner time i. This works well in Lucid because inner subcomputations are always one dimension higher than their immediately enclosing lexical environment.

In Indexical Lucid diagonalization is not appropriate. The reason for this is that if a variable v is defined directly in terms of an indexical where clause, v itself might already be in an environment in which it varies with respect to many indices (not just time as in Lucid). Which of these indices will be used to diagonalize the inner subcomputation? Fortunately, we can use a solution that is far simpler than diagonalization. If a variable v (which may vary in many dimensions) is defined directly in terms of an indexical where clause (which can vary in even more dimensions) we can make the inner dimensionality compatible with the outer dimensionality by setting all the new dimensions created by the index declaration of the where clause to 0. This means that whatever result is being computed by the indexical where clause is carried to the outside by being placed at the origin of the indexical where clause. We call this method of collapsing the origin method. Note in practice most Lucid programs use the origin method since the subject part of an "is current" where clause is usually defined in terms of an asa operator. In indexical
Lucid values can be pinned to the origin by the use of \texttt{as a} or \texttt{@} operators. See the example programs.

**Future Work**

The following are two ideas are worthy of further study.

The first is related to the material in the last section and that is why bother collapsing dimensions at all. Is their a way in which hierarchical objects can be defined and manipulated in the language?

The second idea is very important. In Indexical Lucid function definitions enable the programmer to define their own user defined recursive functions. The introduction of user defined indices brings with it the need for indexical abstraction (as opposed to value abstraction). In other words we need to be able to define functions with indexical parameters as well as value parameters. The observant reader will have noticed that earlier in the paper when addition indexical operators where being introduced we carefully avoided defining a generic \texttt{leftChild} or \texttt{rightChild} function. Instead we gave a definition for the time dimension only. We could have written instead the following generic definition for \texttt{leftChild} and \texttt{rightChild}:

\begin{verbatim}
rightChild.h(x) = x @.h (2*h+1);
leftChild.h(x) = x @.h (2*h);
\end{verbatim}

The reason that we did not do this is that we have no semantics for indexically parametrized functions. We hope to solve this problem in the near future.

**Conclusion**

Indexical Lucid liberates the Lucid programmer from the stream oriented mode of computation. It enables a programs to be written without the introduction of unnecessary monolithic data structures or recursive function definitions. We believe that this will lead to improved performance in any parallel implementation. This work has opened an important an new topic for research namely the problem of indexically parameterised user defined functions.

**References**


[Ash85] Ashcroft, E.A. *Ferds - Massive Parallelism in Lucid*  

[Ash84] Ashcroft, E.A.
The Syntax and Semantics of Lucid

Lucid, a Nonprocedural Language with iteration.

Lucid - A Formal System for Writing and Proving Programs.

Intensional Programming,
In, The Role of Languages in Problem Solving 2, J.C. Boudreaux,
B.W. Hamill, and R. Jernigan (Editors).
Elsevier Science Publishers B.V. (North Holland),
pp. 119-132. 1987,

The pLucid Programmer's Manual
Technical Report TR84-004, Computer Science Department,
Arizona State University, September, 1984.
(This also appears as an appendix to [WaAs85])

[Lan66] P.J. Landin
The next 700 Programming Languages,

[Tho74] R. Thomason, editor
Formal Philosophy, Selected Papers of R. Montague, Yale
University Press, New Haven, Conn., 1974

[WaAs85] W.W. Wadge and E.A. Ashcroft
Lucid, the Dataflow Programming Language,

[WaDu90] W.W. Wadge and Weichang Du
The Eductive Implementation of a Three-dimensional
Spreadsheet,
Software -- Practice & Experience, Nov. 1990.

[WaDu90] W.W. Wadge and Weichang Du
A 3D Spreadsheet Based On Intensional Logic,

[WaDu88] W.W. Wadge and Weichang Du
Appendix

The Syntax and Semantics of Indexical Lucid

This appendix attempts to reproduce for indexical Lucid a concise syntax and semantics. It is based on a scheme used by E.A. Ashcroft [Ash84] to give a syntax and semantics to original Lucid.

Syntax

Informal Introduction

Indexical Lucid is an expression language: Indexical Lucid programs are simply expressions (terms); there are no statements in the language. The language is declarative, and, apart from terms, the other main syntactic category is definitions.

Terms are built up using constants, variables, operation symbols and user-defined function symbols. Compound terms, called where clauses 4, are terms with subsidiary definitions. These clauses may contain indexical declarations. An indexical declaration introduces one or more index names.

Function symbols are defined in the same way as variables, as being the values of terms, but they also have formal parameters.

Variables and function symbols are indexical objects. As will be seen, this enables the language to specify multi-dimensional computations including subcomputations.

The variables and function symbols defined in a clause and the index names declared in the clause are called the locals of the clause. The other variables, index names and function symbols occurring in the clause are called the globals of the clause.

A program is simply a term in which every function symbol that is used is defined in some clause and every index name (except time) is declared in some clause. The variables in the program that are not local to some clause are called the input variables of the program. (The value of the term is called the output of the program.)

---

4 After P.J. Landin [Lan66].
These informal notions are made precise by means of an abstract syntax.

Abstract Syntax

- A program is a term;

- A term is either
  - a constant, or
  - a variable, or
  - an operation symbol together with one or more operands, which are terms, or
  - a function call, i.e., a function symbol together with one or more actual parameters, which are terms, or
  - a clause;

- A clause is a subject, which is a term, together with a body;

- A body is a set of indexical declarations of distinct index names (possibly empty), and a set of definitions of distinct variables and function symbols;

- An indexical declaration is a distinguished set of distinct index names,

- A definition is a right-hand-side, which is a term and a left-hand-side, which is either
  - a variable (for a variable definition), or a function symbol together with formal parameters, which are distinct variables (for a function symbol definition).

The term that is a program must be such that every function symbol appearing in it is defined in some clause.

Implicit in this syntax is the fact that the operands, actual parameters, and formal parameters are ordered. Also, function calls always have the same number of actual parameters as there are formal parameters in the definition of the function symbol in question, and each operation symbol is always used with a particular number of operands, namely the "arity" of the operation.

Concrete Syntax

The following decisions concerning concrete syntax were made for the current Indexical Lucid implementation.

Dyadic operation symbols are written infix. Monadic operation symbols are written prefix (no parentheses necessary). All indexical operation symbols are implicitly defined to operate on the time index. Any indexical operation symbol may be followed immediately by a period ("." and a single valid index name. An index name is valid if it is declared within an enclosing indexical where clause.

Function symbols are always written prefix, with parentheses around the actual parameter list, with the actual parameters separated by commas. The same convention is used in the left-hand-sides of definitions of function symbols, i.e., the
function symbol precedes the list of formal parameters, and the formal parameters are enclosed in parentheses and are separated by commas.

An indexical declaration is a sequence of distinct names separated by commas and appearing after the keyword index. The index name time is reserved for the implicit time context. Definitions are written as equations. Indexical declarations and definitions are always terminated with semicolons.

A clause is written with the subject first and then the body. The body is enclosed by the key words where and end, and within the body the indexical declarations are written before the definitions.

Example

The following program (which computes the running root mean square of its input a) illustrates most of the features of the syntax of Indexical Lucid.

```lucid
sqroot(avg(square(a)))
  where
    square(x) = x*x;
    avg(y) = mean
      where
        n = 2 fby n+1;
        mean = y fby mean + d;
        d = (next y - mean) / n;
      end;
    sqroot(z) = approx asa.i err < 0.0001
      where
        index i;
        approx = z/2 fby.i (approx + z/approx)/2;
        err = abs(square(approx)-z);
      end;
  end
```

Semantics

Introduction

The primary semantics is denotational. Any implementation that produces outputs that are those specified by the denotational semantics is a correct implementation. The denotational semantics does not specify operational details, and there are several different ways of thinking operationally about Indexical Lucid programs, none of which might correspond to an actual implementation, but all of which might be helpful in understanding programs or designing programs.

Denotational Semantics
Indexical Lucid is actually a family of languages, not one particular language. Any one of these languages is differentiated from the others in the family by its choice of the data objects that programs can use, and the simple operations on these objects, and the constants, that are available as primitives. In other words, the language is characterized by an algebra. If the algebra is $A$, we will denote the corresponding version of Indexical Lucid by $\text{IndexicalLucid}(A)$.

We will consider an algebra to be a function from the symbols in its signature to their meanings, elements of the universe (carrier) of the algebra for constants, and operations over the universe of the algebra for operation symbols.

Because the denotational semantics of Indexical Lucid is given using fixpoint theory, the algebra on which an Indexical Lucid language is based has to be a continuous algebra. That is, the universe of the algebra must be a complete partial order (cpo), and all the operations in the algebra must be continuous (using the ordering of the cpo). The least element $\bot$ in the cpo will be called bottom.

$I\text{Lu}(A)$, The Algebra of Indexed Sets from the Universe of $A$

Given a continuous algebra $A$, we can define a continuous algebra $I\text{Lu}(A)$ as follows. The signature of $I\text{Lu}(A)$ is the set of symbols in the signature of $A$, together with various "Indexical Lucid operation symbols" including the dyadic operation symbol $@$ (which is read "at"). (All the different Indexical Lucid operators are definable in terms of $@$.) The universe of $I\text{Lu}(A)$ is the set of all indexed sets of elements from the universe of $A$. (We can consider an indexed set of elements of a set $S$ to be a function from a finite cross product of natural numbers into $S$.) The operation symbols, other than the Indexical Lucid operation symbols, are assigned, as meanings, operations on the universe of $I\text{Lu}(A)$ that are the indexwise extensions of the meanings given to the symbols by the algebra $A$. For example, in $I\text{Lu}(A)$ the addition of two indexed sets of elements from $A$ means simply the indexed set of results produced by adding (according to $A$) corresponding indexical elements of the two arguments.

The Indexical operation symbol $@$ is assigned the operation $@$ which, given two indexed sets, produces the indexed set consisting of a rearrangement of the first indexed set determined by the values of the second indexed set. All the other Indexical Lucid operation symbols, which are definable in terms of $@$ and if-then-else-fi, will be assigned the meanings that would be assigned to functions with these definitions. For example:

\[
\begin{align*}
\text{first}(x) &= x @ 0; \\
\text{next}(x) &= x @ (\text{time}+1); \\
\text{fby}(x,y) &= \text{if } (\text{time}==0) \text{ then } x \text{ else } y @ (\text{time}-1) \text{ fi};
\end{align*}
\]

(It is easy to verify that $I\text{Lu}(A)$ is a continuous algebra if we take as the ordering on the universe of $I\text{Lu}(A)$ the indexwise extension of the ordering on the universe of $A$. That is, index set $X$ is less than or equal to index set $Y$ (by $I\text{Lu}(A)$'s ordering) if and only if the $v$th element of $X$ is less than or equal to the $v$th element of $Y$ (by $A$'s ordering), for all indexical context $v$.)

32
The algebras $ILu(A)$ will give meaning to the constants and operation symbols in
IndexedLucid($A$) programs, but, in addition to constants and operation symbols,
Indexed Lucid programs contain variables and function symbols. Therefore, to give
meanings to programs in a denotational manner, we must give meaning to the
variables and function symbols. This is done by environments.

**Environments**

An *environment* is a function that maps variables into elements of $ILu(A)$ and function
symbols into operations on the universe of $ILu(A)$, of the appropriate arity. (For
theoretical simplicity, the domain of all environments is the set of all variables and
function symbols, but, in practice, only the variables and function symbols occurring in
the program in question will be relevant.)

We now have all the mathematical apparatus necessary for the specification of the
semantics of Indexed Lucid programs. To specify the meaning of an
IndexedLucid($A$) program $P$, it is necessary to be given an environment that gives
meanings to all of the input variables of $P$ (what it says about the other variables and
the function symbols is irrelevant). Since a program is simply a term, to give meaning
to programs it is sufficient to give a general definition of the meaning of terms, given
an environment.

**The Meaning of Terms**

Assuming a continuous algebra $A$, the meaning $M(t,E)$ of a term $t$ in an environment
$E$ is defined as follows.

1. If $t$ is a constant $k$, $M(t,E) = A(k)$.

2. If $t$ is a variable $x$, $M(t,E) = E(x)$.

3. If $t$ is an operation symbol $r$ together with operands $e_1$, $e_2$, ..., $e_n$,
   $M(t,E)$ is $A(r)$ applied to $M(e_1,E)$, $M(e_2,E)$, ..., $M(e_n,E)$.

4. If $t$ is a function symbol $f$, together with actual parameters $e_1$, $e_2$, ..., $e_n$,
   $M(t,E)$ is $E(f)$ applied to $M(e_1,E)$, $M(e_2,E)$, ..., $M(e_n,E)$.

5. If $t$ is a clause with subject $S$ and indexical declarations $I$ and definitions $D$,
   $M(t,E)$ is $M(S,G\#I)$,
   where $G$ is the least environment that agrees with environment $E$,
   except possibly for the locals defined in $D$, and satisfies all the definitions in $D$.
   $G\#I$ (read $G$ "retracted" by $I$) is an environment in which variables
depending on the index names defined in $I$ and declared in $D$ are mapped into
an environment made compatible with $E$ by assigning 0 to the index names in $I$. 
This completes the definition of $M(t,E)$, apart from saying what it means for an environment to satisfy a definition. This is done in terms of $M$, so these two things are really defined simultaneously, mutually recursively.

An environment $E$ satisfies a definition of a variable $x$ with right-hand-side $e$ if the value of $x$ is $M(e,E)$. It satisfies a definition of a function symbol $f$ with formal parameters $g$ and right-hand-side $e$ if, for all environments $F$ that differ from $E$ only in the values given to the variables $g$, the value of the function symbol $f$, applied to the values given to $g$ by $F$, is $M(e,F)$.

The existence of least environments, which is needed for the cases when the term being evaluated is a clause, is guaranteed by an argument from fixpoint theory that depends crucially on the fact that the algebra $A$ in question is a continuous algebra.