The semantics of dimensions as values*

Joey Paquet       John Plaice

May 19, 1999

Abstract

We introduce dimensions as first-class values in the Lucid language. We illustrate
the need for such values through examples, and show that their semantics simplifies
the presentation of Lucid. In order to make multidimensional dataflows easier to un-
derstand, two visual formalisms are introduced, through a number of examples. The
multidimensional dataflow diagram considers dimensions to be values, which can be
passed along flow lines. In addition, flows known to vary in a number of dimensions
are appropriately labeled. The multidimensional evaluation tree presents the demand-
driven process of eduction in a multiple-dimensional space.

1 Introduction

The idea of dimensions as values is not new. The idea was mentioned by the authors in
the Intensional Programming I book [3]. Subsequently, Paquet gave their semantics in
his Ph.D. thesis [2]. We present here a generalization, where a dimension can be created
from any constant value. After presenting the formal syntax and semantics of dimensions
as values, we illustrate their use in Lucid programming and in IHTML, along with their
visualization.

2 Syntax of Dimensional Lucid

Dimensional Lucid is the typed λ-calculus with operators and syntactic sugar (where
classes), along with the operators to manipulate dimensions. Operator dim creates a di-


*Presented to the International Symposium on Languages for Intensional Programming, Demokri-
tos Institute, Athens, Greece, 28–30 June 1999. Current addresses: Joey Paquet, Department of
Computer Science, Concordia University, Montréal (Québec) Canada H3G 1M8. John Plaice, School
of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia.
Email: paquet@cs.concordia.ca, plaice@cse.unsw.edu.au.
\[ E ::= \begin{align*}
& id \\
| & \text{const } c \\
| & \text{op } f \\
| & \text{fn } id_1, \ldots , id_n \Rightarrow E \\
| & E(E_1, \ldots , E_n) \\
| & \text{if } E \text{ then } E' \text{ else } E'' \\
| & \text{dim } E \\
| & \text{undim } E \\
| & E \oplus E' \oplus E'' \\
| & E \text{ where } Q \\
\end{align*} \]

\[ Q ::= id = E \quad \text{or} \quad Q Q \]

3 Semantics of Lucid

Semantic expressions for Lucid expressions are of the form \( D, P \vdash E : v \). In definition environment \( D \) and in context \( P \), expression \( E \) evaluates to value \( v \). The \texttt{const} tag designates that its argument is a constant; \texttt{op} is used for data operations. The definition environment is of type

\[ D : \text{Id} \rightarrow \text{Expr} \]

while the context is of type

\[ P : \text{Val} \rightarrow \text{Val} \]
Figure 1: Semantic rules for newest Lucid
4 Lucid examples

We begin with the runningSum function defined below:

\[
\text{runningSum}(N) = M \\
\text{where} \\
M = N \text{ fby } M + \text{ next } N; \\
\end{end}
\]

Suppose that the running sum of an integer stream \( A \) is supposed to take place in two dimensions, \( \text{da} \) and \( \text{db} \), where \( A \) is (dimension \( \text{da} \) goes horizontally rightwards and dimension \( \text{db} \) goes vertically downwards):

\[
\begin{array}{ccccccc}
1 & 3 & 4 & 2 & 7 & \ldots \\
3 & 1 & 6 & 7 & 9 & \ldots \\
1 & 9 & 2 & 1 & 4 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{array}
\]

Then \( \text{runningSum.d}a(\text{runningSum.db}(A)) \) gives:

\[
\begin{array}{cccccccc}
1 & 4 & 8 & 10 & 17 & \ldots \\
4 & 8 & 18 & 27 & 43 & \ldots \\
5 & 18 & 30 & 40 & 60 & \ldots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{array}
\]

Suppose that integer stream \( B \) varied in three dimensions, \( \text{da}, \text{db} \) and \( \text{dc} \); to compute the running sum in three dimensions, one would have to write:

\[
\text{runnigSum.d}a(\text{runningSum.db}(\text{runningSum.dc}(B)))
\]

But without dimensions as values, it is impossible to compute the general running sum for an \( n \)-dimensional stream, where \( n \) is arbitrary.

A dimension stream is simply a stream whose values are dimensions. We define the general running sum program:

\[
\text{RunningSum.d}(D, N) = M \text{ asa.d iseod}(D) \\
\text{where} \\
M = N \text{ fby.d runningSum.d}(D, M); \\
\end{end}
\]

where it is supposed that \( D \) is a stream of dimensions varying in dimension \( d \). RunningSum calls runningSum, in turn, for each dimension in which \( N \) varies.

For the example with stream \( A \), the expression becomes:

\[
\text{RunningSum.d}a(\text{da fby.d}A \text{ db fby.d}A \text{ eod, A})
\]

For stream \( B \), the expression becomes:
RunningSum.dA(da fby.dA db fby.dA dc fby.dA eod, B)

In each case, dimension dA is a local dimension.

But this is not quite right, since the syntax for dimensions is of the form dim E. In this particular situation we can define the three dimensions as dim "da", dim "db", and dim "dc".

A second example is for testing the equality of n-dimensional finite rectangular arrays:

\[
\text{equal.d,D(E0,E1) = s asa.d iseod(D)}
\]
where
\[
\begin{align*}
  s &= E0 \text{ eq E1 fby.d collection.D s;} \\
  \text{collection.c } E &= (\text{iseod(E) or } E) \text{ asa.c} \\
  &\quad (\text{iseod(E) or not } E);
\end{align*}
\]
end;

The behavior of equal is illustrated by Figure 2.

As we can see, defining a function that acts over an arbitrary stream of dimensions is quite useful. With functions as values, we can do more. Here we have a general-use function that applies a function iteratively to a stream of dimensions:

\[
\text{applydim.d(D,fid,A) = result asa.d iseod(D)}
\]
where
\[
\begin{align*}
  \text{result} &= \text{fid.D(A) fby.d fid.(next.d D)(result)}; \\
\end{align*}
\]
end;

5 Other examples

In the above examples, the dimensions — dim "da", dim "db", and dim "dc" — were all fixed. However, there are situations where one wishes to create dimensions from
existing ones. The following example comes from discussion with Panagiotis Rondogiannis:

\[
f \cdot d = \begin{cases} 
\text{if} \ (\ldots) \ & \text{then} \ (\ldots) \\
\text{else} \ (\text{next} \cdot e \ f \cdot e \ \text{where} \ e = \dim ((\text{undim} \ d) + 1))
\end{cases}
\]

In this situation, \( e \) is a dimension created by unpackaging the dimension in \( d \), adding one to the value, and repackaging as a dimension.

This general approach is also consistent with the current stage of Gord Brown's IHTML.

In the reference manual, there is the following quote:

For example, \texttt{language:english+cuisine:french} describes a version with the value \texttt{english} in the language dimension, and \texttt{french} in the cuisine dimension. If the current version is \texttt{alpha:bravo+charlie:delta}, then the expression \texttt{x$alpha:y$charlie} evaluates to the version \texttt{xbravo:xdelta}. 

\[ \ldots \]

Gord BROWN.

The IHTML dimension \texttt{xbravo} corresponds to the Lucid expression

\[
\dim("x" \cdot \#"alpha")
\]

The IHTML value \texttt{xdelta} corresponds to the Lucid expression

\[
("x" \cdot \#"charlie")
\]

6 Visualizing dimensions

We now present two new visual formalisms for presenting the multidimensional aspects of Lucid programs. The multidimensional dataflow diagrams or networks allow one to present the structure of a program. The multidimensional evaluation tree allows one to understand the demand-driven evaluation process.

These formalisms are quite intuitive, so they are essentially presented through a number of example Lucid programs, beginning from simple, one-dimensional examples, towards more interesting multi-dimensional ones. We begin by giving a description of the problem solved, then give the Lucid approach to the resolution of the problem, which is generally far from the conventional imperative approach.

These two operators can be used to create user-defined functions. The following examples use commonly used functions for the generation of streams of values. Lucid programs using these functions can be thought of as multidimensional dataflow networks, which are simply dataflow networks carrying multidimensional objects. The dimensionality of the objects carried is annotated in the graphs as tags on the edges. We give the dataflow network representation along with the Lucid program for each of the examples.
6.1 The natural numbers

This first example is really simple. However, it captures all the essential aspects of intensional programming. The problem is to extract a value from the stream representing the natural numbers, beginning from the ubiquitous number 42:

\[ (42, 43, 44, 45, \ldots) \]

Let us arbitrarily pick the fourth value of the stream, which is assigned tag number three. Let also the whole stream vary in the \( d \) dimension. The program doing this is simply the following, which is represented in a dataflow graph in Figure 3:

\[
N \oplus_d 2 \\
\text{where} \\
\quad \text{dimension } d; \\
\quad N = 42 \; \text{fby} \cdot d \; N+1; \\
\quad \ldots \\
\text{end};
\]

![Dataflow graph for the natural numbers problem](image)

Figure 3: Dataflow graph for the natural numbers problem

With not much intuition, one can readily expect the program to return the value 45. However, as computers do not rely on intuition for the evaluation of programs, we will give more details of the evaluation method.

To see how the program is evaluated, we rewrite it in terms of the basic \( @ \) and \( # \) operators, which is represented in Figure 4:

\[
N \oplus_d 3 \\
\text{where} \\
\quad \text{dimension } d; \\
\quad N = \text{if } #.d \leq 0 \\
\quad \quad \text{then } 42 \\
\quad \quad \text{else } (N+1) \oplus_d (#.d-1) \\
\quad \quad \text{fi}; \\
\quad \text{end};
\]

102
Figure 4: Totally exploded dataflow network for the natural numbers problem.

Figure 5 shows how evaluation takes place by generating successive demands for the appropriate values of \( N \), until the final computation can be effected. The tree notation should be easily understood. A demand is made at the beginning of one of the long vertical arrows, and its result is found at the head of the arrow. Changes of context are of the form \( d : 1 \).

The previous examples defined and manipulated only one-dimensional intensions. The most interesting feature of Lucid is its ability to naturally define and manipulate multi-dimensional intensions. The following examples use multiple dimensions. Dataflow networks are normally used for the expression of one-dimensional systems. We here generalize dataflow networks by permitting the edges to carry multidimensional tokens. The edges in the dataflow graphs are tagged with the dimension names of the tokens they carry.
Figure 5: Evaluation tree of the natural numbers program

6.2 Matrix Transposition

Scientific programming is mostly about the mathematical manipulation of matrices. A common operation on matrices is the transposition. An imperative program to do matrix transposition typically copies all the matrix elements through an iterative process. In Lucid the transpose is simply done by renaming the dimensions in which varies the matrix.
Figure 6: Dataflow network for the transpose program.

\[
\text{transpose.} \ d_0, \ d_1 (M) = \text{realign.} \ \text{tmp}, \ d_0 (\text{realign.} \ d_0, \ d_1 (\text{realign.} \ d_1, \ \text{tmp}(A)))
\]

where

- dimension \( \text{tmp} \);
- \( \text{realign.} \ a, \ b(X) = X @ a \# b \);
- end;

Figure 7: Dataflow network for the realign function.

The \( \text{tmp} \) dimension used here is local to the \text{where} clause and is used as a temporary dimension to transpose the matrix. Also, the \text{realign} function is here defined locally to the expression defining the \text{transpose} function. Here it would not be possible to use the \text{realign} function in other definitions in the outermost \text{where} clause.

6.3 Matrix Multiplication

Matrix multiplication is one of the most basic and common problems in scientific computation. To multiply two \( n \times n \) matrices \( A \) and \( B \) we have to multiply, pointwise, the rows of \( A \) and the columns of \( B \) and add together the values produced. More precisely, the \((i, j)\)-th element of the product is the sum of the \( n \) values

\[
A_{i, k} \times B_{k, j}, \quad k \in 1..n.
\]

The required program is:
\[
\text{mm} \cdot x, y(M_1, M_2, n) = \text{sum} \cdot z(\text{product} \cdot x, y, z(M_1, M_2), n)
\]

where

\[
\text{dimension } z;
\]
\[
\text{product} \cdot d_1, d_2, d_3(M, N) = \text{realign} \cdot d_2, d_3(M) \ast \text{realign} \cdot d_1, d_3(N)
\]
\[
\text{realign} \cdot a, b(X) = X \otimes .a \# .b;
\]
\[
\text{sum} \cdot d(X, n) = Y \otimes .t \log n
\]

where

\[
\text{dimension } t;
\]
\[
Y = X \text{ fby} .t (\text{firstOfPair} .d(Y) + \text{secondOfPair} .d(Y));
\]
\[
\text{firstOfPair} .a(Z) = Z \otimes .a (#.a*2);
\]
\[
\text{secondOfPair} .a(Z) = Z \otimes .a (#.a*2+1);
\]
end;
end;

In this program, matrix A is turned so that its variation in dimension y is instead changed into variation in dimension z. Similarly, B is turned so that its variation in dimension x is changed into variation in dimension z. As a result,

\[
\text{product} \cdot x, y, z(A, B)_{(x,y,z)} = A_{(x,y,z)} \times B_{(x,y,z)}
\]
\[
A_{(x,z)} \times B_{(x,y)}
\]

The last equality holds since both A and B are supposed to be constant in dimension k.

The 3-dimensional product stream is collapsed into two dimensions by running a sum in the \( z \) dimension, exactly as in the problem statement.

Figure 8: Dataflow network for the mm function.
Figure 9: Dataflow network for the `product` function.

Figure 10: Dataflow networks for `firstOfPair.d(Y)` and `secondOfPair.d(Y)`
7 Conclusion

Dimensions as values simplify the semantics of Lucid and implement a variety of features. Combined with multidimensional diagrams, Lucid’s expressivity increases enormously. However, we are not currently satisfied with the multidimensional diagrams, for they are not fully intensional, and they seem to be only usable for specifying static dimensions. Our intuition is that we need a family of diagrams, which present themselves as needed and explain the situation at any given moment. This intuition, similar to the work by Jagannathan presented in ISLIP95 [1], should lead to more powerful visual programming tools, as well.

References


Figure 12: Evaluation tree of the Matrix Multiplication program