The semantics of dimensions as values

Joey Paquet
Department of Computer Science, Concordia University,
Montréal (Québec) Canada H3G 1M8
paquet@cs.concordia.ca

John Plaice
School of Computer Science and Engineering,
The University of New South Wales, Sydney 2052 Australia
plaice@cse.unsw.edu.au

2000

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 pre-
sentation of Lucid. In order to make multidimensional dataflows easier to understand, 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 successive stages in the Lucid language have come about through the increased use of di-
dimensions. The earliest Lucid [2, 3] just had a single dimension, corresponding to “time”. The
version in the first Lucid book [12] had an extra construct, iscurrent, that allowed for the implicit
creation of a new dimension for the purposes of local computations.

Indexical Lucid,[1] used in the GLU system,[6] introduced the notion of dimensionally abstract
functions. Lucid “streams” were no longer restricted to a fixed set of dimensions but, rather, could
introduce dimensions at will and have them passed as arguments to functions.

It was only a matter of time before dimensions as full values be a subject of discussion. The
idea was mentioned by the authors in the first Intensional Programming book.[9] Subsequently,
Paquet gave their semantics in his Ph.D. thesis.[8] However, the dimensions in that work suppose
that all dimensions are identifiers appearing in the text of a program.

We present here the natural generalization of these Lucid versions, in which any constant value
can be used as a dimension. 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 and Semantics

As is standard, we consider Lucid to be a variant of iswim.[7] In addition to the typed λ-calculus
with operators and syntactic sugar (where clauses), there are two operators to manipulate dimen-
sions: intensional query (@) and intensional navigation (#).

The syntax is given in Figure 1. It supposes that there is a basic algebra of constants (const c)
and data operators (op f).
\[
E ::= \text{id} \\
| \text{const} c \\
| \text{op} f \\
| \text{fn} \ id_1, \ldots, \ id_n \Rightarrow E \\
| E(E_1, \ldots, E_n) \\
| \text{if} E \text{ then } E' \text{ else } E'' \\
| \# E \\
| E @ E' E'' \\
| E \text{ where } Q
\]

\[
Q ::= \text{id} = E \\
| Q Q
\]

Figure 1: Syntax for Lucid

The semantic rules for Lucid are given in Figure 2. In it, the judgments are of the form \( D, P \vdash E : v \), which means that in definition environment \( D \) and in context \( P \), expression \( E \) evaluates to value \( v \). The definition environment is of type

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

while the context is of type

\[
P : \text{Val} \rightarrow \text{Val}
\]

When a value \( v \) is of the form \( c \), it means that it is known to be a constant value.

## 3 Lucid examples

We begin with the `runningSum` function defined below:

```plaintext
runningSum(N) = M
where
  M = N fby M + next N;
end;
```

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

```
1 3 4 2 7 ... 
3 1 6 7 9 ... 
1 9 2 1 4 ... 
: : : : : 
```

Then `runningSum.da(runningSum.db(A))` gives:

```
1 4 8 10 17 ... 
4 8 18 27 43 ... 
5 18 30 40 60 ... 
: : : : : 
```
Suppose that integer stream $B$ varied in three dimensions, $da$, $db$ and $dc$; to compute the running sum in three dimensions, one would have to write:

$$\text{runningSum}.da(\text{runningSum}.db(\text{runningSum}.dc(B)))$$

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:

$$\text{RunningSum}.d(D,N) = \text{next}.d M \text{ asa}.d \text{ iseod} (D)$$

where

$$M = N \text{ fby}.d \text{ runningSum}.d(D,M);$$

end;

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

For the example with stream $A$, the expression becomes:

$$\text{RunningSum}.dA(da \text{ fby}.dA db \text{ fby}.dA eod, A)$$

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 given in Figure 1 does not include the dimension construct. However, we can suppose that appropriate syntactic sugar has been used. In this particular situation we can suppose that the three dimensions are "da", "db", and "dc".

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

\[
\text{equal.d,D(E0,E1) = asa.d iseod(D)}
\]

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 3.

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:

\[
\text{applydim.d(D,fid,A) = next.d result asa.d iseod(D)}
\]

where

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

end;

4 Other examples

In the above examples, the dimensions "da", "db" and "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, from his work in implementing higher-order functions.[10, 11]

\[
f.d = \text{if (...) then (...) else (next.e f.e where e = d+1)};
\]

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.[4] In the reference manual, there is the following quote:
For example, language:english+cuisine:french describes a version with the value english in the language dimension, and french in the cuisine dimension. If the current version is alpha:bravo+charlie:delta, then the expression x$alpha:y$charlie evaluates to the version xbravo:xdelta.

The IHTML dimension xbravo corresponds to the Lucid expression

("x"·("alpha").)

while the IHTML value xdelta corresponds to the Lucid expression

("x"·("charlie").)

5 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.

5.1 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:

\(\langle 42, 43, 44, 45, \ldots \rangle\)

Let us arbitrarily pick the second value of the stream, starting from zero. 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 4:

```lucid
N 0.d 2
where
dimension d;
N = 42 fby.d N+1;
end;
```

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 5:
Figure 4: Dataflow graph for the natural numbers problem

\[
N @.d 2
\]
\[
\text{where}
\]
\[
\text{dimension } \text{d};
\]
\[
N = \text{if } \text{#.d} \leq 0
\]
\[
\text{then } 42
\]
\[
\text{else } (N+1) @.d (\text{#.d}-1)
\]
\[
f1;
\]
\[
\text{end};
\]

Figure 5: Totally exploded dataflow network for the natural numbers problem.

Figure 6 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 \( \text{d} : 1 \).

The previous examples defined and manipulated only one-dimensional intensions. The most interesting feature of Lucid is its ability to naturally define and manipulate multidimensional 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.

5.2 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.

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

where
5.3 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 given below, and the corresponding dataflow diagram is given in Figure 9.

```plaintext
mm.x,y(M1,M2,n) = sum.z(product.x,y,z(M1,M2),n)
where
dimension z;
product.d1,d2,d3(M,N) = realign.d2,d3(M)*realign.d1,d3(N);
realign.a,b(X) = X @.a #.b;
sum.d(X,n) = Y @.t log n
where
```

Figure 7: Dataflow network for the transpose program.

Figure 8: Dataflow network for the realign function.

The tmp dimension used here is local to the where clause and is used as a temporary dimension to transpose the matrix. Also, the realign function is here defined locally to the expression defining the transpose function. Here it would not be possible to use the realign function in other definitions in the outermost where clause.
dimension t;
Y = X fby.t (firstOfPair.d(Y) + secondOfPair.d(Y));
  firstOfPair.a(Z) = Z @.a (#.a*2);
  secondOfPair.a(Z) = Z @.a (#.a*2+1);
end;
end;

\[
\text{mm.x.y(M1,M2,n)}
\]

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,

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

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

The *product* program is given below, and its dataflow program is given in Figure 10.

\[
\text{product.d1,d2,d3(M,N)} = \text{realign.d2,d3(M)} \times \text{realign.d1,d3(N)};
\]

\[
\text{product.x.y.z(M,N)}
\]

The 3-dimensional *product* stream is collapsed into two dimensions by running a sum in the z dimension, exactly as in the problem statement. The result is shown in Figure 11.

The *sum* function uses two functions, *firstOfPair* and *secondOfPair*, illustrated in Figures 12 and 13.

The behavior of the entire system is illustrated by the evaluation tree appearing in Figure 14.
6 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,[5] should lead to more powerful visual programming tools, as well.

References

Figure 13: Dataflow network for $\text{secondOfPair.d}(Y)$.

Figure 14: Evaluation tree of the Matrix Multiplication program


