Visual presentation of multidimensional dataflow programs

Joey Paquet    John Plaice
School of Computer Science and Engineering
The University of New South Wales
Sydney 2052 Australia
{jpaquet,plaice}@cse.unsw.edu.au
May 31, 1998

Abstract

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

This paper presents 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.

In fact, the rest of the paper will be an introduction to Lucid, using these formalisms as support.

Lucid has two primitive operators, called intensional operators. First, for intensional navigation (@) through a multidimensional context space; second for querying the current context of evaluation of the program (#).
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.

2 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 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 1:

\[
N @.d 3
\]
where
\[
\begin{align*}
\text{dimension } d; \\
N &= 42 \text{ fby.d N+1;}
\end{align*}
\]
end;

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

Figure 1: 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 2:

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

Figure 2: Totally exploded dataflow network for the natural numbers problem.

Figure 3 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 \( d : 1 \).
Figure 3: Evaluation tree of the natural numbers program
3 The Hamming Problem

The Hamming problem is a simple problem that consists in generating the stream of all numbers of the form $2^i3^j5^k$ in increasing order and without repetition. The statement of the problem is simple indeed, but its programmation in an imperative language is far from being so simple. However, this problem finds a trivial solution when thought of as a dataflow problem. The graph corresponding to the dataflow solution is given in Figure 4.

![Graph](image)

Figure 4: Dataflow network of the Hamming program.

```lucid
H
where
  H = 1 fby merge(merge(2*H,3*H),5*H);
  merge(x,y) = if (xx<=yy) then xx else yy
  where
    xx = x upon (xx<=yy);
    yy = y upon (yy<=xx);
  end;
end
```

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.
4 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.

\[
\text{transpose.d0,d1(M)} = \text{realign.tmp,d0(realign.d0,d1(realign.d1,tmp(A))))}
\]

where

dimension tmp;
realign.a,b(X) = X @.a #.b;
end;

Figure 5: Dataflow network of the merge function.

Figure 6: Dataflow network for the transpose program.
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.

5 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:

![Diagram for mm function](image)

Figure 8: Dataflow network for the mm function.
\[ mm.x, y(M1, M2, n) = \text{sum}(z) \times \text{product}(x, y, z(M1, M2, n)) \]

where
- \text{dimension} \; z;
- \text{product}.d1, d2, d3(M, N) = \text{realign}.d2, d3(M) \times \text{realign}.d1, d3(N);
- \text{realign}.a, b(X) = X \oplus a \# b;
- \text{sum}.d(X, n) = Y \oplus t \log n

where
- \text{dimension} \; t;
- Y = X \bowtie t (\text{firstOfPair}.d(Y) + \text{secondOfPair}.d(Y));
- \text{firstOfPair}.a(Z) = Z \oplus a \# (a+2);
- \text{secondOfPair}.a(Z) = Z \oplus a \# (a+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,

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

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

The 3-dimensional product stream is collapsed 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.

![Dataflow network for the product function.](image)

Figure 9: Dataflow network for the product function.
Figure 10: Dataflow networks for firstOfPair.d(Y) and secondOfPair.d(Y)

Figure 11: Dataflow network for the sum function.
Figure 12: Evaluation tree of the Matrix Multiplication program

6 Conclusion

We feel that the visual formalisms used in this presentation allow a novice user to readily understand what is happening in a Lucid program, hence help make intensional programming more accessible.