Dimensions and functions as values

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  
1998

Abstract

We introduce dimension and function identifiers 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.

1 Introduction

In Indexical Lucid, dimensions can be passed as parameters to functions, but they are not real values: for example, one cannot create a stream of dimensions. However, this possibility is necessary whenever one wishes to build functions that work over a variable number of dimensions.

For example, consider the runningSum function defined below:

```
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:

```
runningSum.da(runningSum.db(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:
RunningSum.d(D,N) = M asa.d iseod(D)
where
  M = N fby.d runningSum.D(M);
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:

RunningSum.dA(da fby.dA db 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.

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

equal.d,D(E0,E1) = s asa.d iseod(D)
where
  s = E0 eq E1 fby.d collection.D s;
  collection.c E = (iseod(E) or E) asa.c
                 (iseod(E) or not E);
end;

The behavior of equal is illustrated by Figure 1.

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

applydim.d(D,fid,A) = result asa.d iseod(D)
where
  result = fid.D(A) fby.d fid.(next.d D)(result);
end;

Note that we are only manipulating function identifiers, not actual functions, so we only allow Pascal-like higher-order functions, with no partial application.
2 Syntax of Lucid

Now that dimension and function identifiers have become values, the abstract syntax for Lucid programs is greatly simplified. In the following syntax, identifiers $id$ can be constants (type $const$), data operations ($op$), variables ($var$), functions ($func$) or dimensions ($dim$). Lucid's operations are function application, intensional query and intensional navigation.

$$E ::= id$$
$$| E(E_1, \ldots, E_n)$$
$$| \text{if } E \text{ then } E' \text{ else } E''$$
$$| \#E$$
$$| E @ E' E''$$
$$| E \text{ where } Q$$

$$Q ::= \text{dimension } id$$
$$| id = E$$
$$| id(id_1, \ldots, id_n) = E$$
$$| Q Q$$

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 $v$. Identifiers of type $op$, $func$ and $dim$ evaluate to themselves. They can only be used in the appropriate situations.

$$\frac{D(id) = (const, c)}{D, P \vdash id : c}$$
$$\frac{D(id) = (op, f)}{D, P \vdash id : id}$$
$$\frac{D(id) = (dim)}{D, P \vdash id : id}$$
$$\frac{D(id) = (var, E)}{D, P \vdash id : v}$$

$$\frac{D, P \vdash E : id \quad D(id) = (op, f) \quad D, P \vdash E_i : v_i}{D, P \vdash E(E_1, \ldots, E_n) : f(v_1, \ldots, v_n)}$$
$$\frac{D, P \vdash E : id \quad D(id) = (func, id_i, E') \quad D, P \vdash E'[id_i \leftarrow E_i] : v}{D, P \vdash E(E_1, \ldots, E_n) : v}$$

$$\frac{D, P \vdash E : true \quad D, P \vdash E' : v'}{D, P \vdash \text{if } E \text{ then } E' \text{ else } E'' : v'}$$
$$\frac{D, P \vdash E : false \quad D, P \vdash E'' : v''}{D, P \vdash \text{if } E \text{ then } E' \text{ else } E'' : v''}$$

$$\frac{D, P \vdash E : id \quad D(id) = (dim)}{D, P \vdash \#E : P(id)}$$

$$\frac{D, P \vdash E' : id \quad D(id) = (dim) \quad D, P \vdash E'' : v'' \quad D, P \vdash [id \mapsto v''] \vdash E : v}{D, P \vdash E \otimes E' E'' : v}$$
\[
\begin{align*}
\mathcal{D}, \mathcal{P} \vdash Q & : \mathcal{D}', \mathcal{P}' \\
\mathcal{D}', \mathcal{P}' \vdash E & : v
\end{align*}
\]

\[
\mathcal{D}, \mathcal{P} \vdash E \text{ where } Q : v
\]

\[
\begin{align*}
\mathcal{D}, \mathcal{P} \vdash \text{dimension } id & : \mathcal{D}^{\dagger}[id \mapsto (\text{dim})], \mathcal{P}\text{ }^{\dagger}[id \mapsto 0] \\
\mathcal{D}, \mathcal{P} \vdash id = E & : \mathcal{D}^{\dagger}[id \mapsto (\text{var}, E)], \mathcal{P} \\
\mathcal{D}, \mathcal{P} \vdash id(id_1, \ldots, id_n) = E & : \mathcal{D}^{\dagger}[id \mapsto (\text{func}, id_i, E)], \mathcal{P}
\end{align*}
\]

\[
\begin{align*}
\mathcal{D}, \mathcal{P} \vdash Q & : \mathcal{D}', \mathcal{P}' \\
\mathcal{D}', \mathcal{P}' \vdash Q' & : \mathcal{D}'', \mathcal{P}''
\end{align*}
\]

4 Conclusions

We have shown that dimension and function identifiers can be write easily treated as first-class values with little change to the semantics of the language. It is now time to have a working implementation of these and to use them in different applications.