

"LewPlewce"

E A Ashcroft

Computer Science and Engineering Department
Arizona State University

Abstract

In this paper I propose that a couple of modifications be made to the underlying structure of Lucid. I am not proposing "How to do bleen in Lucid." I am proposing ways of making some things simpler in Lucid and of making the language more expandable. In addition, I am proposing a way of making Lucid correspond even more closely to general operator nets.

At ISLIP91, Tony Faustini proposed "Indexical Lucid." In this paper I will consider Indexical Lucid more fully, suggest a couple of changes to it, and also describe a major extension to it that Jaggan Jagannathan calls Indexical Abstraction. I will also describe a feature (multi-headed where clauses) that makes Lucid more modular and compositional. I propose that these two revisions be incorporated into Lucid. I think the revised language should have the name "**Lu+**" and, since it is being proposed in Quebec, it would be appropriate if the name were pronounced in the French way: "LewPlewce."

The important point about Lu+ is that, using it, most of the other extensions to Lucid that have been proposed could be incorporated into it at the same time.

Introduction

I will describe the features that are related to Indexical Lucid, and then talk about multi-headed where clauses. First, however, I will quickly mention a couple of the features of multi-headed where clauses that will be incorporated into the examples in the Indexical Lucid section. In the examples, where clauses are only used as the

right-hand-sides of definitions, not as general terms. (In particular, the subjects of where clauses will not be where clauses.) The second change is that programs will be sets of definitions and declarations (like the bodies of where clauses), rather than terms or where clauses.

Indexical Lucid

In the out-of-print book by Bill Wadge and myself, "Lucid, the Dataflow Programming Language" [4] the idea of having values varying in space as well as in time was introduced. (This was in the "Beyond Lucid" chapter.) To do this, an infinite number of such space dimensions was allowed for: for every natural number i , values could vary in dimension i . An experimental version, with just three space dimensions, was very quickly implemented by Tony Faustini. We found it to be very useful but it tends to be clumsy, or, rather, it tends to emphasize our own clumsiness. We are always using different dimensions for different things and have to be very careful to use the right natural number to identify the dimension. We have introduced operators to change the orientation of a value, so that instead of varying in dimension 2, say, it varies in dimension 1, but nevertheless using the language is still clumsy.

It finally was realised by Tony Faustini that the various dimensions don't have to be ordered like natural numbers. There is nothing about dimension 2 that makes it the "successor" of dimension 1. The 2 and the 1 are really just names. In fact, there is no reason why dimensions cannot be named instead of numbered. This realization is one of the contributions of Indexical Lucid [2].

In Lucid the dimensions are indexed by natural numbers and the elements in each dimension also are indexed by natural numbers. In Indexical Lucid the dimensions themselves are not indexed by natural numbers, but the elements in the dimensions *are*. In Lu+ neither the dimensions nor the elements in dimensions need be indexed by natural numbers.

There really is no reason why the index sets of the different dimensions could not be arbitrary, different, abstract data types. (Having different abstract data types for different dimensions is how we can combine together the different extensions to Lucid that have been proposed, such, for example, the real time extension of John Plaice [3].) In this paper, however, we will only consider the subset of Lu+ that corresponds to Indexical Lucid, i.e., the index sets of the different dimensions all are the natural numbers. I leave it to the reader to imagine how different index sets could be specified for different dimensions.

In Indexical Lucid, the names of dimensions are needed in order to identify which version of **first**, **next**, **fby**, **asa**, etc. we are using. For

example, to move forward in the **x** dimension we use **next.x** and to construct a value that varies in the **y** dimension we use **fb.y**. If we don't use a suffix, we are talking about Lucid's original "time" dimension. For example, if we just say **first** it is the same as saying **first.time**. (Notice that we only get Plane Lucid operations such as **sby** [1]; there is nothing corresponding to the **cby** or **initial** from the Lucid Book.)

The set of dimension names is disjoint from the set of variable names. We never use an expression to give us a dimension. However, in this Indexical Lucid subset of Lu+ we *are* allowed to use a dimension name when an expression is needed - it gives us the current position in that dimension. Essentially, dimension names can act as constants. In fact, **time** can be used as **index** was in Original Lucid.

Once we have dimension names, it is natural to talk of some names being global and some being local. Using local names we get local dimensions, dimensions that are useful right now but which we will not need later and, in fact, we will not be able to refer to later. We get *temporary* dimensions. Using such temporary dimensions, we can define subcomputations without using the Original Lucid **is current** feature. This is the other contribution of Indexical Lucid. (Differing from Indexical Lucid, we will not require every dimension that is used to be declared as a temporary dimension in some enclosing where clause. Some dimensions can be global.)

For example, if **x** varies in various dimensions (which particular dimensions doesn't matter here), in any context we can find the square root of **x** (provided it is 1 or more) by using a temporary dimension, say **temp**, in which to compute closer and closer approximations to the square root using Newton's Method:

Program 1

```
temporary temp;
approx = x/2 fby.temp (approx + x/approx)/2;
err = abs(square(approx) - x);
result = approx asa.temp err < 0.0001;
```

In Original Lucid we had to use **is current**:

Program 2

```
result = approx asa err < 0.0001
where
  X is current x;
  approx = X/2 fby.temp (approx + X/approx)/2;
  err = abs(square(approx) - X);
end
```

Using temporary dimensions in this way, we can eliminate most uses of **is current**. Complications arise when the original Lucid program uses *mixed* clauses, where some of the global variables of the clause are frozen and some are not, as in

Program 3

```

    avg(b) = s/next index
              where
                s = b fby s + next b;
              end;
    end;
    result = root(avg(divergence*divergence))
              where
                A is current avg(X);
                divergence = X - A;
              end;

```

which calculates the running root mean square of **X**. Here, **A** is frozen (the frozen value of **avg(X)**) but **X** is not. Although cases like this are more complicated, we can easily do without **is current**, as we will see.

Note that in the **is current** approach, we say explicitly what is frozen, and everything else is not frozen. In the temporary dimensions approach, on the other hand, we have to say explicitly what is *not* frozen, and everything else *is*. This contrast between the two approaches is best seen if we first get the **is currented** program into a form where only individual variables are frozen, rather than expressions. (In Program 3, the expression **avg(X)** is frozen.) Program 3 should therefore be put into the form

Program 4

```

    A = avg(X);
    avg(b) = s/next index
              where
                s = b fby s + next b;
              end;
    result = root(avg(divergence*divergence))
              where
                A is current A;
                divergence = X - A;
              end;

```

The same program, but using a temporary dimension (say **t**), is

Program 5

```
A = avg(X);
avg(b) = s/next time
      where
          s = b fby s + next b;
      end;
result = root(tavg(divergence*divergence)) @.t time
      where
          temporary t;
          divergence = XX - A;
          XX = X @ t;
          tavg(b) = s/next.t t
              where
                  s = b fby.t s + next.t b;
              end;
      end;
```

Here, we have had to include two versions of "average" - one, **avg**, for **time** and one, **tavg**, for the new temporary dimension, **t**, because **avg** is used in two places in Program 4, one for the outer time and one for the inner time. As predicted, the values of **X**, which were implicitly not frozen in Program 4, have to be *explicitly* unfrozen (as **XX**) in Program 5. (In the definition of **XX**, the operator **@** with no dimension specified really means **@.time**, of course.) The explicit freezing of **A** in Program 4 (by using **is current**) is unnecessary in Program 5.

This program clearly is more complicated than the original Original Lucid program, Program 4, but at least it doesn't use the dreaded **is current**. Also, part of the complication will be removed when we use indexical abstraction, in the next section.

Indexical Abstraction

This last example should motivate indexical abstraction. Tony Faustini's idea is to use a dummy dimension name in the definition of a function and later call the function with a specific dimension name. In the previous example, we can define the "average" function once, but with a dummy dimension, say **d**:

```
avg.d(a) = s/next.d d
      where
          s = a fby.d s + next.d s;
      end;
```

Using this definition (and assuming it occurs globally somewhere, along with **root**), Program 6 becomes

Program 7

```
result = root(avg.t(divergence*divergence)) @.t time
      where
        temporary t;
        divergence = XX - A;
        XX = X @ t;
      end;
A = avg(X);
```

(Using **avg** without a dimension is equivalent to using **avg.time**, of course, just as **@** by itself means **@.time**.)

Original Lucid to Lu+

I will show how to translate all Original Lucid programs to Lu+ programs. Really, the only problem is getting rid of **is current**, and, in particular, handling mixed clauses. I have shown how to do it for Program 4, but the general algorithm has not been described. By continuing to study this particular example, an algorithm will be found.

Program 7 is very similar to Program 4, but what we really need is a program that uses the pieces of Program 4 *unchanged*, so that we can see a simple, general technique that can be applied repeatedly to remove all the **is currents** from an Original Lucid program, no matter how complicated that program is.

In fact, we can make Program 7 almost identical to Program 4.. What we have to do is to have the temporary dimension be called **time**, rather than **t**! Then **avg.t(...)** becomes just **avg(...)** (**time** is implicit). Can we do this? Yes! There are no restrictions on what can be used as the names of dimensions. The only problem in program 7 is that there are two places where expressions use both **time** and **t**. If **t** is changed to **time** the expressions will change their meaning because the distinction is then lost. What we have to do is introduce a dummy dimension (say **cache**) in which to copy **X** (and call it **Y**) and which we can synchronize with the outer **time** dimension, so that then the dimension name, **cache**, will have the same value as **time** when used as an expression:

Program 8

```
A = avg(X);
result = Z @.cache time
      where
        temporary cache;
        Y = X @ cache;
```



```

        Z = root(avg.t(divergence*divergence))
            @.t cache
        where
            temporary t;
            XX = Y@.cache t;
            divergence = XX - A;
        end;
    end;

```

To see that this works, and that **Y** is the same as **X**, but in dimension **cache** rather than dimension **time**, ask yourself what **Y** is in a context where the position in dimension **cache** is *i*. Clearly, it is the value of **X** in the *i*-th position in the **time** dimension. By similar reasoning, the value of **XX** at the *j*-th position in dimension **t** is the value of **Y** at the *j*-th position in dimension **cache**, which, in turn, is the value of **X** at the *j*-th position in dimension **time**. And this is all done without referring to **time** within the innermost **where** clause, where **XX** is defined!

We are now able to change the name of the temporary dimension from **t** to **time**, and at the same time rename **XX** to be **X**. This is done, not to make the program more understandable, but just to get it into right form for the transformation.

Program 9

```

A = avg(X);
result = Z @.cache time
  where
    temporary cache;
    Y= X @ cache;
    Z = root(avg(divergence*divergence))@ cache
      where
        temporary time;
        X = Y@.cache time;
        divergence = X - A;
      end;
  end;
end;

```

We should compare this with the original program using **is current**, i.e., Program 4:

```

A = avg(X);
result = root(avg(divergence*divergence))
  where
    A is current A;
    divergence = X - A;
  end;

```

The crucial point is that everything except the **is current** declaration is used without change. This means that we can easily translate Original Lucid programs to Lu+ programs as follows.

Given a definition using a **where clause**

```

lhs = S where
                                x1 is current E1;
                                x2 is current E2;
                                .
                                D1;
                                D2;
                                .
                                end

```

we get the definitions

```

x1 = E1;
x2 = E2;
.
lhs = z @.cache time
  where
    temporary cache;
    yy1 = y1 @ cache;
    yy2 = y2 @ cache;
    .
    z = S @ cache
      where
        temporary time;
        y1 = yy1 @.cache time;
        y2 = yy2 @.cache time;
        .
        D1;
        D2;
        .
      end;
    end;

```

where **y1**, **y2**, etc. are the variables, used in the definitions **D1**, **D2**, etc., that are global to the original where clause. The crucial point is that we don't need to look into the expressions **S** and **E1**, **E2**, etc. or the definitions **D1**, **D2**, etc. Very simple text substitution operations will produce the required text.

Multidimensionality

It wasn't mentioned earlier, but more than one temporary dimension can be introduced in a where clause. Similarly, there can be more

than one dummy dimension-parameter when we are defining an indexically abstract function.

To have more than one local dimension in a where clause, we just have a list of dimension names in the **temporary** statement. Similarly, to have more than one dimension-parameter in a function definition or in its use, we again just separate them by commas.

For example, the function **move**, defined below, copies its argument from one dimension to another

```
move.a,b(Z) = Z @.a b;
```

and we can use it as follows

```
result = C asa.x C>B asa.y C>A  
where  
  temporary x,y;  
  C = move.time,x(A) + move.time,y(B);  
end
```

We can also use it in the program we have just been developing:

Program 11

```
A = avg(X);  
result = Z @.cache time  
where  
  temporary cache;  
  Y = move.time,cache(X);  
  Z = root(avg(divergence*divergence))@ cache  
    where  
      temporary time;  
      X = move.cache,time(Y);  
      divergence = X - A;  
    end;  
end;
```

Another possible use of indexical abstraction is in the tournament function (which has already been referred to many times when discussing GLU applications). Previously we always had to rewrite the tournament function to take into account the dimension that the data values are laid out in, and the dimension we want the results to appear in. Now, these two dimensions can just be dummy dimension arguments of the tournament function. Even better, the function could have a function argument, namely the operation we want to use in the tournament. The tournament takes place in some temporary dimension.

Multi-Headed Where Clauses

In the "Beyond Lucid" chapter of the Lucid Book, Bill and I considered adding "tuples" to Lucid, to enable where clauses to return more than one object. We found problems with adding tuples, mainly in the area of type checking. We didn't persist with the problem, I think, because pLucid has list structures and people could just return a list if they wanted to return more than one object. However, I feel that with the ability to have objects varying in space as well as in time, the use of lists in Lucid has waned. I feel that the need to be able to return more than one object from a where clause has to be readdressed. I also feel that we should avoid going the "tuples" route. We can solve the problem by tackling it head on, by simply allowing where clauses to have more than one subject expression.

Such multi-headed where clauses are *not* terms. You can't perform arithmetic on them and you can't use them as arguments of functions. In fact, you can only use them to define variables and functions, which is all people usually use where clauses for anyway, so I suggest we make this restriction apply to *all* where clauses, even singly-headed ones. Following this suggestion, all the programs given above use where clauses this way. I have also made the programs be sets of definitions, which is another syntactic change I recommend.

The syntax is easy to specify, but we first have to clarify the notion of *arity*. There are three classes of identifiers that are distinct: variables, constants, and dimensions. Each variable and constant has an arity, which is a pair of natural numbers indicating the number of dimension arguments and the number of other arguments that its value takes. For example, the value, in a particular context, of a variable or constant of arity (0, 0) is an individual object, such as the natural number 7. The value of a constant of arity (0, 1) is a monadic operation on individual objects, such as the squaring operation. The value of a constant of arity (1, 0) is an object whose value depends on the position in question in some dimension, such as the constant **index** which just gives the position in question in the time dimension. The value of a constant of arity (1, 1) is a monadic operation that depends on the position in question in some dimension, such as the operation **next.x** that gives the value of its argument at the position following the one in question, going in the direction of the **x** dimension.

Dimension identifiers do not have values; they are just used to parameterize variables or constants. They do not have arities.

Abstract Syntax

A Lu+ program is a sequence of temporary dimension names and a set of definitions of various sizes.

A *definition of size n* consists of a left-hand-side and a right-hand-side. The *right-hand-side* is simply a where clause of size n . The *left-hand-side* consists of a sequence of n distinct variables, all with the same arity (which we call the arity of the definition), and two sequences of identifiers: a sequence of distinct dimension names and a sequence of distinct variables. The lengths of these last two sequences must match the arity of the definition. Thus, if the arity of the definition is (i, j) , the left-hand-side consists of n variables of that arity, i dimension names, and j variables. The n variables will be called *siblings*.

A *where clause of size n* is a sequence of n terms, called the *subject* of the clause, a sequence of *temporary-dimension names*, which are dimension identifiers, and a set of definitions of various sizes. Since the sequence of temporary-dimension names and the set of definitions may be empty, a where clause might just be a sequence of terms or a single term. Together, the sequence of temporary-dimension names and the set of definitions form the *body* of the where clause.

A *term* is a variable or constant, together with arguments. If the variable or constant is of arity (i, j) , the arguments consist of a length i sequence of dimensions and a length j sequence of terms.

Concrete Syntax

As far as their non-dimension arguments are concerned, variables are written prefix (with parentheses) and constants are written infix. The sequence of dimension arguments follows the variable or constant, with a period between the two. (The period may be missing, of course, if there are no dimension arguments.)

Here are some constants and their arities:

+	(0, 2)	time	(1, 0)
if then else fi	(0, 3)	1	(0, 0)
first	(1, 1)	2	(0, 0)
next	(1, 1)		

Variables and constants of arity $(1, i)$ for some i can be used in terms or on the left-hand-sides of definitions *without* a dimension argument. In such cases, it is assumed that the missing dimension is **time**. The dimension **time** can be used both implicitly and explicitly in the same program. Such a program is equivalent to the one obtained by making **time** explicit, and this latter program is said to be *fully dimensioned*. Notice that **time** could be a temporary dimension, and that a variable or constant that is used with a missing dimension would then be referring to that temporary dimension. There is no renaming of temporary dimensions to avoid clashes with the implicit **time** dimension.

The length- n sequences of terms that are subjects of where clauses, and the sequences of n distinct variables in the left-hand-sides of definitions of size n , will be enclosed in square brackets (unless n is 1).

Temporary dimensions are introduced using the keyword **temporary**, of course, and the body of a where clause begins with the keyword **where** and ends with the keyword **end**.

Example:

It is often useful to be able, given integers m and n , to refer to the integer portion of m divided by n and also to the remainder. (In fact, we have operations in Lucid, **div** and **mod**, to do just that. We will assume in this example that they do not already exist, and we are defining them.) In Lu+ we do not have a tupling facility, so we do not define a function that returns a pair. Instead, we define two functions, siblings, that share formal parameters and share a body:

```
[ div, mod ] (m, n) =  
  [ stoppingPoint, whatsLeft @.d stoppingPoint]  
  where  
    temporary d;  
    whatsLeft = m fby.d whatsLeft - n;  
    stoppingPoint = d asa.d whatsLeft < n;  
  end;
```

This example shows a multi-headed where clause of size 2 and shows its use in a definition of size 2. The functions **div** and **mod** can be called, in terms, in the usual way.

The motivation for defining the two functions this way is the possibility of increased efficiency. If we call **div(327, 7)** we will generate over 40 elements of **whatsLeft** in order to get the answer. If we subsequently call **mod(327, 7)**, with a clever implementation we may be able to avoid generating any more elements. To do this, the implementation just has to be able to recognise at compile time that the arguments in the two cases are textually the same, and then give the two calls the same place tag. The second call then just picks up the value of **stoppingPoint** at that place tag (it has already been computed) and uses it as the **d**-context of the value of **whatsLeft** that it needs. (That value has already been computed, also.)

Without multi-headed where clauses, we would have to define the two functions separately, and the two calls would do the same amount of work.

Incidentally, this idea of giving different calls of a function with identical arguments the same place tag would also be useful in Original Lucid. We just never did it.

Another motivation for multi-headed where clauses is that it improves the correspondence between Lucid and operator nets. In Original Lucid, every where clause corresponded to an operator net with one exit, and every operator net with one exit corresponded to a where clause. With multi-headed where clause Lucid, every where clause of size n corresponds to a general operator net with n exits, and every operator net with n exits corresponds to a where clause of size n . Given an operator net, any subnet with m ingoing edges and n outgoing edges can be thought of (encapsulated, abstracted) as a node for n sibling functions, with the functions being defined with one where clause of size n .

Conclusion

This paper has attempted to describe a couple of ways of tidying up Lucid to make it more textual and also to make it more flexible and usable as far as space dimensions are concerned.

References

1. Ashcroft, E.A., and Faustini, A.A. "Adding Intensionality to Functional Programs," Proceedings of ISLIP89, the 2nd International Symposium on Lucid and Intensional Programming, Tempe, Arizona, May 1989.
2. Faustini, A.A. and Jagannathan, R., "Indexical Lucid," Proceedings of ISLIP91, the 4th International Symposium on Lucid and Intensional Programming, Menlo Park, California, April 1991, pp. 19-34.
3. Plaice, J., "ML-Lucid, An Intensional Functional Language," Proceedings of ISLIP92, the 5th International Symposium on Lucid and Intensional Programming, Oakland, California, April 1992, Workshop of the 1992 International Conference on Computer Languages, pp. 54-62.
4. Wadge, W.W. and Ashcroft, E.A., "Lucid, the Dataflow Programming Language," Academic Press, U.K., 1985.