Lucid is Second-Order:
Explaining Lucid and Intensionality to Functional Programmers

Ed Ashcroft
Arizona State University

Abstract

The notion of describing Lucid programs in terms of functional programs is considered. As we follow the development of Lucid, and Lucid gets richer, describing it functionally gets harder and harder, until surprisingly, with the advent of the notion of intensionality, it becomes very easy yet still elusive.

Throughout, the functional programs we need are second-order.

Introduction: the Early Years

Here are two very simple Lucid definitions:

\[ \text{m} = \text{N} \ \text{fby} \ \text{m} - \text{one}; \]
\[ \text{one} = 1; \]

To a Lucid programmer, these indicate that \text{m} starts off as the first value that \text{N} has (\text{N} could be varying), and then at successive times has a value one less than the value at the previous time. To a functional programmer, it probably indicates that \text{m} is an infinite list, starting with the head of \text{N} and repeatedly getting one less. (Infinite lists will be denoted \([a \ b \ f \ .. \ *]\), with "*" indicating the unspecified, and nonexistent, end of the list.) This way of describing Lucid, using large data structures, we will call the Monolithic approach. Using it here, \text{m} should be

\[ \[(\text{car N}) (\text{car N - 1}) (\text{car N - 1 - 1}) (\text{car N - 1 - 1 - 1}) .. *] \]

The best way to express this is by using \text{map}. (Throughout this paper we will use Haskell[1], but practically any functional language would do.) Here is the way to do it:

\[ \text{m} = \text{car N} : (\text{map } (\text{a} \rightarrow (\text{a} - \text{one})) \text{m}) \]
\[ \text{one} = 1 \]

This isn't much more complicated than the original Lucid program. \text{m} is defined re-
cursively in the same way as in Lucid, with `(cons)` acting for `fby`. However, in the Lucid program `one` could have been defined to be varying, so that, in that case, in the Haskell program `one` would be a list. That would mean that the definition of `m` would be incorrect (you can’t subtract lists). In Lucid, *everything* is potentially varying, so the definition of `m` would be the same whether `one` were really varying or not, that is, whether the value of `one` depended on the context or not. To have this true in the Haskell program, the program must be different, using a version of `map`, `map2`, that works pointwise for binary functions or operations:

\[ m = \text{car} \cdot N : (\text{map2} \ (-) \ m \ \text{one}) \]

For this to work, we need a definition of `map2` and a definition of `one`. `one` now has to be an infinite list:

\[ \text{map2} \ f \ c \ d = (f \ (\text{car} \ c)(\text{car} \ d)) : (\text{map2} \ f \ (\text{cdr} \ c)(\text{cdr} \ d)) \]
\[ \text{one} = 1 : \text{one} \]

It is easy to see how we got the Haskell definition of `m` from the Lucid definition, but it is not so obvious what the algorithm was that got us the definition of `one`. Really, what I should have done is say that constants are nullary operations, and introduce a version of `map`, `map0`, that works for nullary functions and operations:

\[ \text{map0} \ f = f : (\text{map0} \ f) \]
\[ \text{one} = \text{map0} \ (1) \]

The definition of `map0` really does follow the pattern set for `map` and `map2`, and the above definition really does define `one` in the right way.

So, from the Lucid definitions

\[ m = N \ fby \ m - \text{one}; \]
\[ \text{one} = 1; \]

we get

\[ m = \text{car} \cdot N : (\text{map2} \ (-) \ m \ \text{one}) \]
\[ \text{one} = \text{map0} \ (1) \]
(with appropriate definitions of `map0` and `map2`).

It isn’t radically different, but is a little clumsier, especially when it comes to reasoning about programs. One of the nice properties of Lucid is that it is easy to state and prove properties of programs. In this (Lucid) example, it is true, and easily proved, that

\[ m = \text{first} \cdot N - \text{now} \]

where `now` is a built-in constant of Lucid, corresponding to the definition

\[ \text{now} = 0 \ fby \ \text{now} + 1; \]

In fact, we could have used the above property of `m` as the definition of `m`.

In the Haskell example, if we have the corresponding definition

\[ \text{now} = \text{car}(\text{map0} \ 0) : (\text{map2} \ (+) \ \text{now} \ \text{one}) \]
it is true that

\[ m = \text{map2} (-) (\text{map0} (\text{car} \ N)) \text{ now} \]

but, I suspect, it is not as easy to prove. (It could, however, in the same way, be used as the definition of m.)

Let us look at a slightly less simple Lucid definition:

\[
\text{fact} = \text{if } m \leq 1 \text{ then } 1 \text{ else } m^* (\text{next fact}) \text{ fi};
\]

where m is as before. The Lucid operation next gives the value in the next time context. For a corresponding Haskell program next can be thought of as cdr. The straightforward translation would give

\[
\text{fact} = \text{map3} (\text{if}) (\text{map2} (\leq) \text{ one}) \text{ one} (\text{map2} (*) \text{ m} \text{ (cdr fact)})
\]

\[
\text{if } T \ t \ e = t \\
\text{if } F \ t \ e = e
\]

but does this work? I suspect that the answer is "yes", provided \text{cons} ($) is nonstrict in \textit{both} its arguments. Where is \text{cons}? Inside the various \text{map} functions. The list fact is constructed backwards. If \text{car} \ N is 6, say, the list fact will be [720 120 24 6 2 1 .. *], with 720 being the last element that is added. Just one list is created, but at first its initial elements are suspended computations (□)

\[
[\square \square \square \square \square .. *]
\]

while the list m decreases

\[
[6 5 4 3 .. *]
\]

Eventually, m gets down to 1, and an element of fact, the furthest element needed, gets a value:

\[
[\square \square \square \square \square \square 1 .. *]
\]

The earlier elements get filled in, backwards, by multiplication with m, yielding [720 120 24 6 2 1 .. *]. Of course, the Lucid program behaves similarly: initially the values of fact for particular time contexts are defined in terms of the values of fact for the next time contexts. Clearly, this "time" is not real time.

Of course, it would make sense to say

\[
\text{factorial} = \text{first fact};
\]

in Lucid, and

\[
\text{factorial} = \text{car fact}
\]

in Haskell, but notice that in both languages we just get the factorial of the first value of N. If N is varying, say 6, 3, 5, etc. at times 0, 1, 2, etc., \text{factorial} is always 720. If we wanted to get 720 at time 0, 2 at time 0, 6 at time 1, 120 at time 2, etc., we would need the Lucid definition

\[
\text{factorial} = \text{first fact}
\]

where

\[
\begin{align*}
n &\text{ is current } N; \\
m &\text{ = n fby m - 1;}
\end{align*}
\]

\[
\text{fact} = \text{if } m \leq 1 \text{ then } 1 \text{ else } m^* (\text{next fact}) \text{ fi;}
\]

\[
\text{end;}
\]
In Haskell, we can get the same effect for this example using `map`:

```haskell
factorial = map f n
f n = car fact
where
  fact = map3 (if) (map2 (<=) m one) one (map2 (*) m (cdr fact))
  m = n : (map2 (-) m one)
```

Summarizing, simple Lucid programs, i.e., pre-1985 - those only involving "time" - are equivalent to rather more complicated functional programs that make heavy use of `map`.

1984: Newspeak and Doublethink?

Starting in the late 1970's, Bill Wadge and I tried to add arrays to Lucid. We came up with various array algebras, but never found one that really satisfied us. In 1984 we gave up the search for an array algebra, and decided to get space sequences as elements of time sequences, without any of the complications of designing an array algebra, by simply saying that variables can vary in space as well as in time. The change to the interpreter would be minimal, just requiring tags to be more complicated (space points as well as time points), and requiring no big change to the syntax and semantics of Lucid. It was the easy way out of our dilemma, but we didn't know if it would give us the functionality we wanted, and it probably would be messy and weird. It was only later that we realised that, far from opening Pandora's Box or a can of worms, we had in fact let the genie out of the bottle. The genie is called Intensionality. It is very powerful but is quite tame.

We will consider 'Plane Lucid'. In Plane Lucid, the space dimensions and the time dimension are all on an equal footing. We might talk about a variable being a time sequence of vectors of vectors of integers, say, but really all we can really say is that the variable is integer valued when given a context of a time and two space positions, one for each of the two space dimensions considered. We could equally well talk about the variable being a vector of vectors of time sequences, or a vector of time sequences of vectors. This is difficult to express as a list, monolithically. Perhaps harder to fit into the list model is the fact that a dimension 0 variable is different from a dimension 1 variable, and these two 1-dimensional objects, when added together, for example, give a 2-dimensional result.

Just as Lucid has the built-in time varying constant `now`, it has an arbitrary number of built-in space varying constants `here_0`, `here_1`, etc. These correspond to the definitions:

```haskell
here_0 = 0 sby 0 here_0 + 1;
here_1 = 0 sby 1 here_1 + 1;
here_2 = 0 sby 2 here_2 + 1;
```

etc.

So, if we have

```haskell
a = here_0;
```
\[ b = \text{here}_1; \]
\[ c = \text{here}_2; \]

what can we say? How can we describe it in a Haskell program? We can imagine three orthogonal vectors in space, but what is it in Haskell? Two approaches: (I) a vector in dimension \( i \) is a pair, consisting of an integer (the dimension, \( i \)) and a list of elements; (II) a vector in dimension 0 is a list, and a vector in dimension 1 is a plane that doesn’t vary in the 0-th dimension, so it is a constant list of lists, and so on. Hence, a vector in dimension \( i \) is an \( i-1 \) dimensional object, a constant list of constant list of constant lists of... of lists. Thus, under (I), \( a \) is

\[ 0 : (L : \text{nil}) \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

while \( b \) is

\[ 1 : (L : \text{nil}) \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

and \( c \) is

\[ 2 : (L : \text{nil}) \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

On the other hand, under (II), \( a \) is

\[ L \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

while \( b \) is

\[ \text{map0} L \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

and \( c \) is

\[ \text{map0(map0 L)} \]
\[ \text{where} \]
\[ L = 0 : (\text{map2} (+) L \text{ one}) \]

Which approach is better? It appears that approach (I) is better because it treats dimensions equally. However, we can get a better appraisal if we see how they they are capable of dealing with simple Lucid situations. Consider, for example, \( a + c \). In Lucid, this denotes an object that varies in two dimensions, dimension 0 and dimension 2. In any context, if the dimension 0 position is \( j \) and the dimension 2 position is \( k \), its value is \( j + k \).

Using approach (II), this would be denoted in Haskell by

\[ \text{map g a} \]
\[ g i = \text{map0(map2 (+) (map0 i) (car(car c))} \]

Using approach (I), on the other hand, one immediately runs into the problem of representing an object that varies in more than one dimension. And, since it is approach (I), we would want to do so while treating dimensions equally. It really does
seem to be beyond (I)'s capabilities.

Reluctantly, we will have to abandon approach (I), and, holding our nose, try to use approach (II).

We haven't considered space-varying things that vary also in time. Using approach (II), we have to add a further level, and consider the outermost level to be indicating the time-variation. Really, each of the expressions we had for a, b, and c should have had map0 applied to them, because they are constant in time.

Approach (II) is distasteful because there is an ordering of the dimensions that is built into the structure itself. Is that really so bad? To see that it is, consider the following simple Lucid definition, which is a large part of a Lucid program for solving Laplace's equation in two dimensions by relaxation, i.e., by, at each stage, replacing the field value at each point in the plane by the average of its four neighbors (north, south, east, and west):

\[
\text{field} = \text{ORIG fby}
\]
\[
(succ_0 \text{ field} + (pred_0 \text{ field}) + (succe_1 \text{ field}) + (pred_1 \text{ field}))/4;
\]

What are succ_0, pred_0, etc? succ, meaning "successor", is like next but applied to space. The natural number indicates the dimension in question. (next is like cdr, remember?) pred, meaning "predecessor", is like the opposite of succ - it moves earlier along the dimension. At this point the functional programmers are calling "Foul!" - there is no way to move backwards along a list. To have a level playing field, we will modify this program so that it only uses succ. The fragment becomes:

\[
\text{field} = \text{ORIG fby (succ_0 G + succe_1 G + succe_0 succe_1 G + succe_1 succe_0 G)/4;}
\]

\[
G = 0 \text{ sby} 0 (0 \text{ sby_1 field});
\]

Now the plane starts at coordinate (0, 0), and goes in the positive directions. (Before, it was infinite in the negative directions, too.)

Can we do this in Haskell? Originally I thought it would be nearly impossible, but it is not too bad. (Notice that field and G vary in time and in dimensions 0 and 1.) Here is a Haskell coding:

\[
\text{field} = \text{car ORIG : (map4 f1 A B C D) A = map (a->(cdr a)) G B = map (a->(map (b->(cdr b)) a)) G C = map (a->(map (b->(cdr (cdr b))) a)) a A D = map (a->(cdr (cdr a))) B f1 a b c d = map4 f2 a b c d f2 a b c d = map4 avg a b c d avg a b c d = (a + b + c + d)/4 G = map0 0 : map (\L->(0 : L)) field
\]

The lists A, B, C, and D correspond to the four things that are added in the Lucid
definition of field. Notice that they are very different in the Haskell program, but are similar in the Lucid program. This is due to the fact that the ordering of the dimensions affects the data structure used.

Is it as understandable, or as easy to write in the first place, as the Lucid program?

No.

(Despite the heavy use of map, it is only a second-order program.)

The Explicit-Intension Method

I think it must be admitted that the Haskell programs given in this paper have all been longer and harder to follow than the Lucid programs they were designed to emulate. There are features of post-1984 Lucid that are even harder to emulate functionally Monolithically. We have seen the operators \texttt{pred 0}, \texttt{pred 1}, etc., but there are others. For example, there is a binary operator \texttt{atSpace i} that will give the value of its first argument in a context that differs from the current one in the \texttt{i}-th dimension in a way specified by its second argument. It sounds complicated, but it is very easy to do in Lucid, and is very useful. It is very difficult to do in Haskell, or, at least, very expensive. Provided, that is, we use the Monolithic method. Is there any other? In fact, yes. There is an approach that follows Lucid very closely, which we call the Explicit-Intension approach.

All the expressions in Lucid denote intensions, that is, functions from implicit contexts to basic data values. A context will say, for example, what is the current "time", and what are the positions in the various space dimensions. As indicated earlier, the expression \texttt{here 0 + here 2} denotes a function that, for any context where the position in dimension 0 is \texttt{i} and the position in dimension 2 is \texttt{j}, has the value \texttt{i + j}.

In the Explicit-Intension approach, a Lucid program is converted to a set of Haskell functions that are exactly the intensions denoted by the Lucid program. To specify these Haskell functions, their domain, the set of contexts, must be defined. Of course, in Lucid these contexts are implicit, and any attempt to actually say what they are exactly, to utter their names aloud, as it were, is to risk a bolt of lightening, or, at least, excommunication. But have no fear: the intrinsic mystery of Lucid is preserved. The Haskell intension "functions" are not really functions. They have to be written in a notation that leaves questions unanswered.

We introduce the notation \( C[\zeta] \), where \( \zeta \) specifies part of a context. \( C[\zeta] \) denotes any context that has that part. For example, \( C[s2: 4, t: 6] \) denotes a context for which the time is 6 and the dimension 2 position is 4. Such a context expression will be used (illegally) as part of the pattern in a function definition. For the Lucid expression \texttt{here 0 + here 2}, for example, we can give its Haskell meaning as

\[
\text{f where}
\]

\[
\text{f } C[s0: i, s2: j] = i + j
\]
This isn’t legal Haskell, but the meaning is clear.

I could now go through all the examples considered earlier and show what they would be as Explicit-Intensions. They would all be very simple. I will show only the most complicated one, for the original Laplace’s equation program (the one with \texttt{pred}). The Explicit-Intension program is

\[
\begin{align*}
\text{field } C[t: 0] &= \text{ORIG } C[t: 0] \\
\text{field } C[t: i + 1] &= (\text{succ}_0 \text{ field } C[t: i]) + \\
&\quad (\text{pred}_0 \text{ field } C[t: i]) + (\text{succ}_1 \text{ field } C[t: i]) + \\
&\quad (\text{pred}_1 \text{ field } C[t: i])/4
\end{align*}
\]

The intension functions associated with the operators \texttt{pred} \_i and \texttt{succ} \_i are part of the Explicit-Intension formulation of Lucid. They are

\[
\begin{align*}
\text{succ} \_i A C[si: a] &= A C[si: a + 1] \\
\text{pred} \_i A C[si: a] &= A C[si: a - 1]
\end{align*}
\]

The drastic simplification, and the direct correspondence to the Lucid fragment should be obvious.

The intensions for variables in Lucid all are functions from contexts to basic values. Functions in Lucid are all first-order (until Bill Wadge’s extension[2] is adopted), so they are functions from intensions (functions) to values - they are really second-order. Unsurprisingly, all the Haskell programs we have got from Lucid programs are second-order.

Conclusion

The Monolithic approach is very complicated and difficult and requires \texttt{cons} to be nonstrict in both its arguments. The Explicit-Intension approach is straightforward and simple, but can’t, strictly speaking, be completely captured in Haskell. In either case, second-order programs are produced. Lucid should really be thought of as a second-order language.

References
