

## HIGHER ORDER LUCID

*Bill Wadge*

Computer Science Department  
University of Victoria  
Victoria, British Columbia  
Canada

### ABSTRACT

In this paper we remedy one of the most annoying weaknesses of Lucid and indexical languages in general: the lack of higher order functions. We describe a simple generalization of the "place code" approach (as discovered by Ostrum and formalized by Yaghi) which handles programs with functions of arbitrarily high orders; in other words, typed Iswim.

A source program  $P$  with function definitions (including recursive ones) is translated into a 0-order program  $P'$  (one with no user-defined functions) with extra indexical operators for manipulating place codes.

The only difference between the first order and the general case is that in the general case we need more than one dimension of place codes. In fact, the number of dimensions needed is exactly the order of the program.

The translated program  $P'$  can be implemented by education - tagged, demand-driven dataflow. The use of place codes causes no real complications: they are treated as just another field in the tag, along with the time and space tags. Naturally, the technique works even in the simple case that the original program has no indexical operators - is pure Iswim. In this case the technique gives us a promising new dataflow approach to implementing the typed lambda calculus.

### The Need for Higher-Order Dataflow

Indexical programmers have long felt that the lack of higher order functions was one of Lucid's greatest weaknesses - a weakness shared by all the indexical languages and, more generally, by most dataflow languages.

This weakness was discussed in the Lucid Book where, for example, it was pointed out that we have to write

```
comp(P) = optimize(code(parse(lex(P))))
```

We cannot simply write

```
lex | parse | code | optimize
```

even though the latter reveals much better the dataflow structure (a simple pipeline) of the expression. The problem is that reverse function composition (the operation  $|$ ) is second order - its arguments are (first order) functions.

A more serious instance of the problem arises in numerical analysis, one of the preferred application areas for dataflow. Many numerical algorithms (such as integration) naturally have a parameter which is a function (such as the function to be integrated). We cannot write a single Lucid function which performs integrations; instead we need a set of cloned function definitions, one for every particular function which needs (say) integrating

The problem is not really semantics: since the collection of Lucid streams forms a domain, we can in turn form domains of arbitrarily high order functions over streams. There are some subtle issues concerning the distinction between (say) a stream of second order functions and a second order filter. The question, however, is more one of elegance than practicality, and a promising approach was outlined in the Lucid book.

The real problem is an engineering one: how do we *implement* a higher order dataflow language? Until recently, the only way known is to use some hybrid system in which (for example) function closures are passed around a network. These hybrid approaches greatly complicate the implementation because they require nondataflow techniques such as pointers, heaps, recursive traversals and mark-and-sweep garbage collection.

We need an implementation which uses only dataflow techniques, such as tags, token matching, and a variable-value store.

### Translating Recursion to Iteration

It is possible to understand our approach to implementing functions as a generalization of the well known method for translating tail recursion into iteration.

Consider, for example, the following recursive "schema":

```
F(I)
where
  F(X) = if p(X) then q(X) else h(F(k(X))) fi;
end
```

This is equivalent to the following two loop imperative schema

```
input(I);
X := I
while not p(X) do X := k(X);
F := q(X);
X := I;
while not p(X) do begin X := k(X); F := h(F) end;
```

The iterative version is slightly more elegant in Lucid:



```

first F
where
  X = I fby k(X);
  F = if p(X) then q(X) else h(next F) fi;
end

```

The Lucid version is simpler because it uses "future recursion" to model more closely the recursive computation. Notice especially the role of  $x$  and  $F$ : up to the point that  $p(x)$  is first true,  $F$  is the result of applying the function  $F$  to the corresponding value of  $x$ . In the functional version,  $x$  was a formal parameter, but in the Lucid version it has become the stream of *actual* parameters which arise when the functional version is executed in the usual sense. In the functional version,  $F$  is a function variable; but in the Lucid version it is the stream of function results which arise during execution of the functional version.

It may not be genuine iteration, but it can be implemented directly using education. The demand for the first value of  $F$  generates demands for the second, third etc values of  $F$ . At each step the corresponding value of  $x$  is tested for the property  $p$ . Eventually (at time index  $i$ ) the computation advances to a point where  $p(x)$  is finally true. At that point the corresponding value of  $F$  is  $q(x)$ , and the demands for earlier values of  $F$  can start to unwind. Eventually the demand for  $F$  at time 0 produces the result of applying  $h$   $i$  times to the value of  $q(x)$  at time  $i$  - and this is the result of applying  $k$   $i$  times to the original value of  $x$ .

Of course the translation to Lucid fails if the original source already had temporal operators. (Many interesting effects can be obtained by mixing recursion and iteration). The translation introduces new occurrences of the temporal operators which interfere with those already in the source. The solution, however, is obvious: add a new, independent time dimension used only by the recursion elimination translator. We need not necessarily even think of the new dimension as being "time", although sometimes that might be helpful.

### The Place Dimension as Branching Time.

The method just described clearly fails for more general recursive definitions in which the body of the definition has more than one call. For example, there is no way to translate the schema

```

F(A)
where
  F(X) = if p(X) then q(X) else h(F(k0(X)), F(k1(X))) fi;
end

```

into an iterative version, whether imperative or in Lucid. The problem is that the collection of function calls generated by  $F(A)$  has, in general, a tree structure and not a linear structure. For example, the evaluation of  $F(A)$  may lead to the evaluation of  $F(k0(A))$  and  $F(k1(A))$ , the latter may lead to  $F(k0(k1(A)))$  and

$F(k1(k1(A)))$ , and so on. There is no simple way in which the collection of actual parameters form a stream.

In the imperative paradigm, this is the end of the story. Indexical languages, however, are not so limited. An ordinary lucid stream is a set of values indexed by a linearly ordered collection of "timepoints" (the natural numbers). There is no reason not to consider other, richer structures that would yield a generalization of the notion of stream powerful enough to handle general recursion.

One idea which suggests itself is branching time: a notion of time in which each timepoint has a number of independent successors. In our branching time model there is still a unique first instant, and every instant has a whole sequence of successors, one for each natural number. The structure is that of a tree with infinitely many branches at each node. Each timepoint is uniquely determined by the sequence of choices of branches, starting from the initial timepoint. We can therefore think of a timepoint as a finite sequence of natural numbers, with the initial time being the empty sequence.

We have a single *first* operator, an operator *next<sub>i</sub>* for every natural number *i*, and an operator *fby* which can take an arbitrary number of right hand arguments. The semantic equations are

$$\begin{aligned} first(X)_t &= X_{\langle \rangle} \\ next_i(X)_t &= X_{it} \\ A fby (B_0 B_1 B_2 \dots)_{0t} &= A_t \\ A fby (B_0 B_1 B_2 \dots)_{(i+1)t} &= (B_i)_t \end{aligned}$$

The translation of the recursive definition given above into branching iteration is now straightforward:

```
first F
where
  F = if p(X) then q(X) else h(next0 F, next1 F) fi;
  X = A fby ( k0(X) , k1(X) )
end
```

Again, if the source already has temporal operators, we have to consider the branching time dimension used in the translation as a fresh new dimension distinct from any others already in use.

### The Ostrum-Yaghi translation

All the different educative implementations of Lucid and related languages handle user-defined functions using essentially the technique described above. The first of these implementations was written by Calvin B. Ostrum, but the only documentation of his implementation techniques was the source code itself. Ostrum's compiler generated an intermediate form of the program but it was not human readable and in particular could not be considered as a special 0-order



(function-free) form of Lucid.

The place code approach was formalized by A. Yaghi in his PhD dissertation. He introduced the notion of an "intensional algebra", the formal basis for intensional (or indexical) programming. Yaghi showed that Ostrum's preprocessing step could be understood as a translation into 0-order indexical Iswim.

Yaghi's intuitions were slightly different and as a result he used a slightly different set of operators with different names. Yaghi viewed the place codes as nodes in the tree of function calls or invocations generated during a conventional call-return execution. Instead of *nexti* he used *calli*, the idea being that the value of (say) *call13 E* is the value *E* has in the third child of the current invocation. Instead of the generalized *fbby*, Yaghi used an operation *actuals* exactly like *fbby* but without the left hand argument. The operator got its name because it is used to define the value of a formal parameter as that of one of several actual parameters.

Yaghi's semantic equations are

$$\begin{aligned} call_i(X)_p &= X_{ip} \\ actuals(A_0, A_1, A_2, \dots)_{ip} &= (A_i)_p \end{aligned}$$

Assume for the sake of simplicity that we are translating a program with only one user defined function *F*. Assume also that there are no nested *where* clauses and that all variables (including formal parameters) are distinct. Then the Yaghi translation can be summarized as follows.

- (1) Number the textual occurrences of calls to *F* starting at 0 (including calls in the body of the definition of *F*).
- (2) Replace call *i* of *F* with the expression *calli(F)*.
- (3) Strip the formal parameters from the definition of *F*, so that *F* is defined as an ordinary individual variable.
- (4) Introduce a definition for each formal parameter of *F*. The right hand side of the definition is the operator *actuals* applied to a list of the actual parameters corresponding to the formal parameter in question, listed in the order in which the calls are numbered.

Here, for example, is the Yaghi translation of the branching recursive schema

```
call0(F)
where
  F = if p(X) then q(X) else h(call0(F), call1(F)) fi;
  X = actuals(A, k0(X), k1(X));
end
```

The eduction of the "Yaghi code" version is straight forward. The only difficulty is that the place codes can grow arbitrarily long; but the implementation uses a hashing scheme to assign small unique natural numbers to each place code (this is also due to Ostrum).

## Translating Higher Order Programs

The techniques just described are not new; Ostrum's interpreter appeared nearly 15 years ago, and Yaghi's work is nearly 10 years old. Nevertheless for all that time no one ever succeeded in extending the idea to higher order programs. The author and various colleagues made several unsuccessful attempts, all involving a futile search for a slightly more complex domain of place codes. The real solution is simple enough but requires an appreciation for notions of dimensionality which are only just emerging (or maybe we're just plain thick).

The starting point is the observation that the Yaghi translation need not be applied to *all* parameters of *all* functions. It can be applied selectively. Consider, for example, the Lucid program

```
diff(sq(sq(index)),3)
where
  diff(X,N) = if N<1 then X else diff(d1(X),N-1) fi;
  d1(Y) = next Y - Y;
  sq(Z) = Z*Z;
end
```

Here is a partial translation which eliminates formal parameters *x* and *y*:

```
call0 diff(3)
where
  diff(N) = if N<1 then X else call1 diff(N-1) fi;
  d1 = next Y - Y;
  X = actuals(sq(sq(index)),call0 d1);
  Y = actuals(X);
  sq(Z) = Z*Z;
end
```

(Notice that we have eliminated all the *nonsynchronic* parameters.)

Now consider the following simple second-order program:

```
ffac(sq,3) + ffac(cb,4)
where
  ffac(H,N) = if N<1 then 1 else H(N) * ffac(H,N-1) fi;
  sq(P) = P*P;
  cb(Q) = Q*Q*Q;
end
```

If we apply the partial translation method above we can eliminate the formal parameter *H*:



```

call10 ffac(3) + call11 ffac(4)
where
  ffac(N) = if N<1 then 1 else H(N) * call2 ffac(N-1) fi;
  H = actuals(sq,cb,H);
  sq(P) = P*P;
  cb(Q) = Q*Q*Q;
end

```

The function `ffac` now has only 0-order arguments; it is first order. The whole program is first-order except for the definition of `H`, which is an equation between function expressions. But we can introduce a formal parameter `z` for `H` and rewrite the whole thing as a first-order indexical program

```

call10 ffac(3) + call11 ffac(4)
where
  ffac(N) = if N<1 then 1 else H(N) * call2 ffac(N-1) fi;
  H(Z) = actuals(sq(Z),cb(Z),H(Z));
  sq(P) = P*P;
  cb(Q) = Q*Q*Q;
end

```

Now we can apply the Yaghi translation again, but with a *new* place dimension isomorphic to the first, with operators `actuals'`, `call10'`, `call11'`, ...:

```

call10 call10' ffac + call11 call11' ffac
where
  ffac = if N<1 then 1 else call10' H * call2 call2' ffac fi;
  N = actuals'(3,4,N-1);
  H = actuals(call10' sq, call10' cb, call11' H);
  Z = actuals'(N,Z);
  sq = P*P;
  P = actuals'(Z);
  cb = Q*Q*Q;
  Q = actuals'(Z);
end

```

The result is a purely indexical, function-free form of the original second order program. It can be evaluated by the standard eduction technique, the only difference being that the tags have two place fields instead of one.

### Generalizing the Approach

It should be clear that (say) a fifth order program can be translated in five stages, yielding a 0-order indexical program with operators for five distinct but identical place dimensions. Notice that not all the variables will be five dimensional. In general, the dimensionality of a variable (the number of place codes needed to determine a value) corresponds to its order in the original source program.

This translation gives us a new technique for implementing the typed lambda calculus. The educative implementation uses the same run-time system already found in existing implementations. There is no need to manipulate closures or any other hierarchical structures; no pointers, no heaps, and no recursive mark-and-sweep garbage collection. We could argue that it is the first true dataflow implementation of higher order lambda calculus.

The technique breaks down when we attempt to translate an untypable term, such as the definition of the Y combinator in terms of self application. The stage-by-stage translation never ends, because each stage is equally high order. There may perhaps be a way of "summarizing" this process in a finite program over an infinite dimensional index space, but this remains to be investigated.

The approach has not yet been implemented but that is not a difficult task. The preprocessing is only slightly more complicated than that for the first order case, mainly because we have to infer function types for the program variables. Educating the translated programs should be simple - there are already implementations for multi-dimensional Lucid extensions, and place dimensions are not especially complex.

Now that the technique has been extended to higher order programs, it is more important than ever to prove it correct. This problem was also open for many years, but now I believe I have an approach that works. The basic idea is to consider a program P and the program P' which results when one applies one beta conversion (involving a user defined function). One simply shows that the translated programs Y and Y' are semantically equivalent. This would be easy if Y' could be obtained from Y by a corresponding beta reduction, but things are not that simple; functions in P are just variables in Y. However, it is possible to show that Y can be transformed into something clearly equivalent to Y', using rules such as

$$\begin{aligned}\text{call13}(P+Q) &= \text{call13}(P) + \text{call13}(Q) \\ \text{call13}(\text{actuals}(A,B,C,D,E)) &= E\end{aligned}$$

### Acknowledgements

The stages approach was suggested by George Nelan's "firstification" work, and by work with Tony Faustini on lazy tags.

Originally I intended to collaborate with Yaghi himself on this project, but he is at the University of Kuwait and will be incommunicado until the end of the Gulf war. Perhaps we will have heard from him by the time this is published. Until then, we can only hope that he and his family come to no harm.