Higher-order Multidimensional Programming

John Plaice

Senior Scientist, GrammaTech, Ithaca

Adjunct, UNSW Australia, Sydney

7 March 2018

Joint work with William W. Wadge
Previously: Jarryd Beck, Gabriel Ditu, Blanca Mancilla

Once upon a time ···

In 1975, William W. Wadge realized that a variable in a `while` program could be understood as a stream:

```
I := 0;
...
while ...
  I := I+1;
  ...
```

The stream $\langle 0, 1, 2, \ldots \rangle$ became, in the language Lucid:

$$\mathit{first}\ I = 0;$$
$$\mathit{next}\ I = I + 1;$$

· · · there was a beautiful idea

One of my favorite old languages is one called Lucid by [Bill Wadge and] Ed Ashcroft. It was a beautiful idea. He said, "Hey, look, we can regard a variable as a stream, as some sort of ordered thing of its values and time, and use Christopher Strachey's idea that everything is wonderful about tail recursion and Lisp, except what it looks like."

Strachey said, "I can write that down like a sequential program, as a bunch of simultaneous assignment statements, and a loop that makes it easier to think of." That's basically what Lucid did—there is no reason that you have to think recursively for things that are basically iteration, and you can make these iterations as clean as a functional language if you have a better theory about what values are.

Alan C. KAY, 2004.

Semantics? What semantics?

Wadge's colleague, E. A. Ashcroft, asked what the semantics was.

It turns out that there were two possible choices.

- Prefix order:

$$S_1 \sqsubseteq S_2 \quad \text{iff} \quad \text{len}(S_1) \leq \text{len}(S_2) \ \wedge \ \forall i < \text{len}(S_1),\ S_1(i) \equiv S_2(i)$$

- Scott order:

$$S_1 \sqsubseteq S_2 \quad \text{iff} \quad \forall i \in \mathbb{N},\ S_1(i) \equiv \bot \vee S_1(i) \equiv S_2(i)$$

This is the story of those two choices.

LUSTRE = LUcid Synchrone Temps-RÉel

- Early 1980s: Paul Caspi and Nicolas Halbwachs were working on the semantics of timed systems, collaborating with SNCF (Société Nationale des Chemins de fer Français).
- Paul Caspi discovered Lucid and saw its relevance.
- They added a timed, synchronous interpretation to Lucid, hence "Synchronous Real-time Lucid".
- The French ≪lustre≫ corresponds to the English "æon".
- *Sémantique et Compilation de LUSTRE, un langage à flux de données temps-réel*
  John Plaice, PhD thesis, INP Grenoble, 1988.

Lustre: Clocked Streams

- All streams sharing a clock are updated once per instant.

$$X = 0 \texttt{ -> } 1 + \texttt{pre } X$$
$$Y = X \texttt{ -> } X + \texttt{pre } Y$$

- Here, 0, 1, X and Y all share the base clock.

instant	0	1	2	3	4	⋯
0	0	0	0	0	0	⋯
1	1	1	1	1	1	⋯
pre X	nil	0	1	2	3	⋯
X	0	1	2	3	4	⋯
pre Y	nil	0	1	3	6	⋯
Y	0	1	3	6	10	⋯

Lustre Becomes Real

- Denotational semantics
  - A clocked stream is a pair $(clock, stream)$.
  - A clock is the base clock (a Boolean stream) OR a Boolean clocked stream.
  - Least-fixed-point semantics for systems of equations.
- Static semantics
  - Allocates a clock to each variable.
  - Similar to type inference in functional languages.
- Code generation
  - single-loop OR
  - finite state automaton.

Lustre Flies!

- Verilog develops a toolkit around Lustre.
- Initial work with Merlin-Gérin (nuclear reactor control) AND Aérospatiale (Airbus avionics).
- Successive takeovers, ultimately to Esterel Technologies.
- Scade Toolkit: visual programming with Lustre semantics:
  - Airbus A380, A340-500, A340-600 (EU).
  - Sukhoi SuperJet 100 (Russia).
  - Eurocopter Écureuil/Astar AS 350 B3 (EU).
  - Embraer Phenom 100 (Brazil), with Pratt-Whitney PW617F.
  - Cessna Citation Encore+ (USA), with Pratt-Whitney PW535B.
  - Dassault Aviation Falcon 7X (France).
- Airbus A380: 750,000 lines of automatically generated C code, formally verified using abstract interpretation (Patrick Cousot, ENS–Paris; Reinhard Wilhelm, Saarbrücken).

Thirty Years of Lucid Development

- *Eduction* coined simultaneously in 1978 for
  - Demand-driven, out-of-order, evaluation of Lucid AND
  - *Naming and Synchronization in a Decentralized Computer System* (David P. Reed, PhD thesis, MIT)
    - software transactional memory for distributed systems;
    - Reed was the / in TCP/IP (Kay); invented UDP.
  - E-duce. *tr.v.* To draw or bring out.
  - From Latin *educo, is, ere, duxi, ductum*: root for both "educe" and "educate".
- Ferd Lucid (1985): Experiments in multidimensionality.
- Field Lucid (1986): Lucid variables are *intensions*, i.e., mappings from multidimensional contexts to values.
- Indexical Lucid (1993): Dimensionally abstract `where` clauses (Tony Faustini and Jagannathan.)

TransLucid

- TransLucid project began with visit to UNSW Australia by William W. Wadge in late 2005.
- *The TransLucid Programming Language* (Gabriel Cristian Ditu, PhD thesis, UNSW Australia, 2007).
- *Cartesian Programming* (John Plaice, Habilitation thesis, U.Grenoble, 2010).
- *TransLucid: From Theory to Implementation* (Jarryd Phillip Beck, PhD thesis, UNSW Australia, 2015).
- Jarryd P. Beck, John Plaice, and William W. Wadge. Multidimensional infinite data in the language Lucid. *Mathematical Structures in Computer Science*, 25(7):1546–1568, 2015.

TransLucid

- **TransLucid** is a language in which
  variables and expressions define intensions.
- An **intension** is a function mapping contexts to values.
- A **context** is a set of (**dimension**, **ordinate**) pairs.
- During evaluation of an expression,
  there is always a current runtime context.

Suppose we wish to compute the factorial of n (here $n = 6$), using an approach called *tournament computation*. We will use two *dimensions*, here d_1 (horizontal) and d_2 (vertical):

F	d_1	$\rightarrow$								
d_2	1	1	2	3	4	5	6	1	1	$\cdots$
$\downarrow$	1	6	20	6	1	1	$\cdots$			
	6	120	1	1	$\cdots$					
	720	1	1	$\cdots$						

The first row is simply the enumeration from 1 to n, with an additional 1 to the left and a sea of 1s to the right. In every other row, the i-th element is the product of the $2i$-th and $(2i + 1)$-st elements from the previous row.

F is the entire, infinite table.

Here is the 0th-order TransLucid definition of F:

$$
\begin{array}{l}
F \\
\texttt{where} \\
\quad \texttt{dim}\ d_1 \leftarrow 0 \\
\quad \texttt{dim}\ d_2 \leftarrow \texttt{if}\ n \equiv 0\ \texttt{then}\ 1\ \texttt{else}\ \left\lfloor log_2(n) \right\rfloor + 1 \\
\quad \texttt{var}\ F = \texttt{if}\ \#d_2 \equiv 0 \\
\qquad\qquad \texttt{then if}\ \#d_1 \equiv 0\ ||\ \#d_1 > n\ \texttt{then}\ 1\ \texttt{else}\ \#d_1 \\
\qquad\qquad \texttt{else}\ F\ \texttt{@}\ [d_2 \leftarrow \#d_2 - 1, d_1 \leftarrow \#d_1 \times 2] \times \\
\qquad\qquad\qquad F\ \texttt{@}\ [d_2 \leftarrow \#d_2 - 1, d_1 \leftarrow \#d_1 \times 2 + 1] \\
\texttt{end}
\end{array}
$$

For order-1, we begin by defining some standard functions.

$$\texttt{fun } \mathit{lPair}.d(X) = X \texttt{ @ } [d \leftarrow \#d \times 2]$$
$$\texttt{fun } \mathit{rPair}.d(X) = X \texttt{ @ } [d \leftarrow \#d \times 2 + 1]$$
$$\texttt{fun } \mathit{fby}.d(X, Y) = \texttt{if } \#d \equiv 0 \texttt{ then } X \texttt{ else } Y \texttt{ @ } [d \leftarrow \#d - 1]$$
$$\texttt{fun } \mathit{default}.d(\mathit{val}, n, X) = \texttt{if } \#d \equiv 0 \texttt{ || } \#d > n \texttt{ then } \mathit{val} \texttt{ else } X$$

Here is the 1st-order TransLucid definition of F:

$$
\begin{array}{l}
F \\
\texttt{where} \\
\quad \texttt{dim}\ d_1 \leftarrow 0 \\
\quad \texttt{dim}\ d_2 \leftarrow \texttt{if}\ n \equiv 0\ \texttt{then}\ 1\ \texttt{else}\ \left\lfloor log_2(n) \right\rfloor + 1 \\
\quad \texttt{var}\ F = default.d(1, n, \#d_1)\ fby.d_2 \\
\qquad\qquad\qquad lPair.d_1(F) \times rPair.d_1(F) \\
\texttt{end}
\end{array}
$$

Of course, we want a factorial *function*:

$$
\begin{array}{l}
\texttt{fun } \mathit{fact}(n) = F \\
\texttt{where} \\
\quad \texttt{dim } d_1 \leftarrow 0 \\
\quad \texttt{dim } d_2 \leftarrow \texttt{if } n \equiv 0 \texttt{ then } 1 \texttt{ else } \left\lfloor \mathit{log}_2(n) \right\rfloor + 1 \\
\quad \texttt{var } F = \mathit{default}.d(1, n, \#d_1)\ \mathit{fby}.d_2 \\
\qquad\qquad\qquad \mathit{lPair}.d_1(F) \times \mathit{rPair}.d_1(F) \\
\texttt{end}
\end{array}
$$

Note that *fact* is not a recursively defined function, rather it just picks out the appropriate element from an infinite table, the latter defined recursively.

For higher-order, we can encode the tournament computation:

$$
\begin{array}{l}
\texttt{fun } \mathit{tournament}.d(\mathit{pos}, \mathit{op}, X) = F \\
\texttt{where} \\
\quad \texttt{dim } d_2 \leftarrow \texttt{if } \mathit{pos} \equiv 0 \texttt{ then } 1 \texttt{ else } \left\lfloor \mathit{log}_2(\mathit{pos}) \right\rfloor + 1 \\
\quad \texttt{var } F_2 = X \ \mathit{fby}.d_2 \ \mathit{op}(\mathit{lPair}.d(F_2), \mathit{rPair}.d(F_2)) \\
\texttt{end}
\end{array}
$$

Note that *tournament* has two different kinds of arguments: d is a dimensional argument, while *pos*, *op* and X are normal arguments.

Here is our first higher-order factorial *function*:

$$
\begin{array}{l}
\texttt{fun } \mathit{fact}(n) = F \\
\texttt{where} \\
\quad \texttt{dim } d_1 \leftarrow 0 \\
\quad \texttt{fun } \mathit{mul}(x, y) = x \times y \\
\quad \texttt{var } F = \mathit{tournament}.d_1\left(n, \mathit{mul}, \mathit{default}.d_1(1, n, \#d_1)\right) \\
\texttt{end}
\end{array}
$$

Of course, this is the λ Meetup!

$$\begin{aligned}
&\texttt{fun } \mathit{fact}(n) = F \\
&\texttt{where} \\
&\quad \texttt{dim } d_1 \leftarrow 0 \\
&\quad \texttt{var } F = \mathit{tournament}.d_1\big(n, \lambda(x,y) \to x \times y, \\
&\qquad\qquad\qquad\qquad \mathit{default}.d_1(1, n, \#d_1)\big) \\
&\texttt{end}
\end{aligned}$$

But there's a catch!
mul and the λ function are not equivalent.

Here is an example of a stream of functions:

$$
\begin{array}{l}
pow(3)(4) \\
\texttt{where} \\
\quad \texttt{fun}\ pow(n) = P \\
\quad \texttt{where} \\
\quad\quad \texttt{dim}\ d \leftarrow n \\
\quad\quad \texttt{var}\ P = \big(\lambda(m) \rightarrow 1\big)\ fby.d \\
\quad\quad\quad\quad\quad\quad \big(\lambda(m) \rightarrow m \times P(m)\big) \\
\quad \texttt{end} \\
\texttt{end}
\end{array}
$$

to produce $4^3 = 64$. P is the table

P	d	$\rightarrow$			
	$\lambda m.m^0$	$\lambda m.m^1$	$\lambda m.m^2$	$\lambda m.m^3$	$\cdots$

Standard Functions (1)

$$\texttt{fun } \mathit{index}.d = \#.d + 1$$
$$\texttt{fun } \mathit{first}.d\ (X) = X\ \texttt{@}\ [d \leftarrow 0]$$
$$\texttt{fun } \mathit{next}.d\ (X) = X\ \texttt{@}\ [d \leftarrow \#.d + 1]$$
$$\texttt{fun } \mathit{fby}.d\ (X, Y) = \texttt{if } \#.d \equiv 0 \texttt{ then } X \texttt{ else } Y\ \texttt{@}\ [d \leftarrow \#.d - 1]$$
$$\begin{aligned}\texttt{fun } \mathit{wvr}.d\ (X, Y) = \texttt{if } & \mathit{first}.d\ Y \\ & \texttt{then } X\ \mathit{fby}.d\ \big((\mathit{next}.d\ X)\ \mathit{wvr}.d\ (\mathit{next}.d\ Y)\big) \\ & \texttt{else } (\mathit{next}.d\ X)\ \mathit{wvr}.d\ (\mathit{next}.d\ Y)\end{aligned}$$

Standard Functions (2)

- $A = \langle a_0, a_1, \ldots \rangle$ and $B = \langle b_0, b_1, \ldots \rangle$ are d-streams.

	dim d $\rightarrow$						
	0	1	2	3	4	5	$\cdots$
index.d	1	2	3	4	5	6	$\cdots$
first.d A	a_0	a_0	a_0	a_0	a_0	a_0	$\cdots$
next.d A	a_1	a_2	a_3	a_4	a_5	a_5	$\cdots$
A fby.d B	a_0	b_0	b_1	b_2	b_3	b_4	$\cdots$

Standard Functions (3)

- $B = \langle T, F, T, T, F, T, T, F, T, \ldots \rangle$ is a Boolean d-stream.

	dim d $\rightarrow$						
	0	1	2	3	4	5	$\cdots$
A *wvr.d* B	a_0	a_2	a_3	a_5	a_6	a_8	$\cdots$

Ackermann

	dim d_n $\rightarrow$						
ack	0	1	2	3	4	5	$\cdots$
dim $d_m \downarrow$ 0	1	2	3	4	5	6	$\cdots$
1	2	3	4	5	6	7	$\cdots$
2	3	5	7	9	11	13	$\cdots$
3	5	13	29	61	125	253	$\cdots$
4	13	65533	$\cdots$				
5	65533	$\cdots$					
$\vdots$	$\ddots$						

Ackermann

$$
\begin{array}{l}
\texttt{fun}\ ack(m,n) = A \\
\texttt{where} \\
\quad \texttt{dim}\ d_m \leftarrow m \\
\quad \texttt{dim}\ d_n \leftarrow n \\
\quad \texttt{var}\ A = (index.d_n)\ fby.d_m \\
\qquad\qquad \big((next.d_n\ A)\ fby.d_n\ (A\ @\ [d_n \leftarrow next.d_m\ A])\big) \\
\texttt{end}
\end{array}
$$

$$
\begin{aligned}
ack(0,n) &= n+1 \\
ack(m,0) &= ack(m-1,1) \\
ack(m,n) &= ack\big(m-1, ack(m,n-1)\big)
\end{aligned}
$$

Sieve of Eratosthenes

$$
\begin{array}{l}
\texttt{fun } \mathit{sieve}.d = S \\
\texttt{where} \\
\quad \texttt{dim } d' \leftarrow 0 \\
\quad \texttt{var } S = (\#.d' + 2)\ \mathit{fby}.d \\
\qquad\qquad \big(S\ \mathit{wvr}.d'\ \big(S \bmod (\mathit{first}.d'\ S) \not\equiv 0\big)\big) \\
\texttt{end}
\end{array}
$$

		dim d' $\rightarrow$								
	S	0	1	2	3	4	5	6	7	$\cdots$
dim $d\downarrow$	0	2	3	4	5	6	7	8	9	$\cdots$
	1	3	5	7	9	11	13	15	17	$\cdots$
	2	5	7	11	13	17	19	23	25	$\cdots$
	3	7	11	13	17	19	23	29	31	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Matrix multiplication

$$
\begin{array}{l}
\texttt{fun}\ multiply.d_r.d_c(k, X, Y) = W \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow 0 \\
\quad \texttt{var}\ X' = rotate.d_c.d(X) \\
\quad \texttt{var}\ Y' = rotate.d_r.d(Y) \\
\quad \texttt{var}\ Z = X' \times Y' \\
\quad \texttt{var}\ W = sum.d(k, Z) \\
\quad \texttt{fun}\ rotate.d_1.d_2(X) = X\ \texttt{@}\ [d_1 \leftarrow \#.d_2] \\
\quad \texttt{fun}\ sum.d_x(n, X) = Y\ \texttt{@}\ [d_x \leftarrow n] \\
\quad \texttt{where} \\
\quad\quad \texttt{var}\ Y = 0\ fby.d_x\ (X + Y) \\
\quad \texttt{end} \\
\texttt{end}
\end{array}
$$

Implementation of Order-0 TransLucid

- Any order-0 TransLucid program can be rewritten into an equivalent program with a single `where` clause.
- For an order-0 program with a single `where` clause, dimensions can be statically attributed to dimension identifiers.
- The results for a (*variable*, *context*) pair can be cached. This is equivalent to memoization of previously computed results in ordinary functional programs.
- The difficulty with caching order-0 TransLucid programs is that the context can be of unbounded size, even though most of the context is irrelevant to the computation at hand.
- The cache is therefore a decision tree minimizing context information. In iteration i, the cache is asked for the value of (x, κ_i), where $\kappa_0 = \emptyset$ and $\kappa_i \subseteq \kappa$. Should a query for further dimensions D_i be returned, then the cache is asked for the value of $(x, \kappa_i \cup (\kappa|_{D_i}))$.

Implementation of Order-1 TransLucid

- All order-1 TransLucid programs can be rewritten into order-0 TransLucid programs.
- Function calls for non-recursive functions can be inlined.
- Function calls of recursively defined filters, such as *wvr*, can be replaced by demands made into recursively defined streams.
- Function calls to ordinary recursively defined functions can also be replaced by demands made into recursively defined streams, using *placeholder dimensions* whose ordinates are stacks of unsigned integers encoding the calling stack.
- Ordinary recursively defined functions are rarely needed in TransLucid. One exception is *quicksort*, as its branching is completely unpredictable, depending entirely on the data at hand and the choice of pivot.

Implementation of Higher-order TransLucid

- This is much harder, as static allocation of dimensions to dimension identifiers does not work.
- If the `lambda` construct is not used, then an extension of the placeholder dimension technique, used for implementing ordinary recursive functions in order-1 TransLucid, may well work. A new dimension would be needed for all of the functions of the same higher-order type.
- For `lambda` constructs, a completely different technique is needed. The interpreter developed by Jarryd Beck for his PhD thesis did this, but it was never proven correct.

The Two Paths Meet . . . Soon

- Ideas developed to a greater or lesser degree.
- A *TransLucid system* has the behavior of a Lustre program, i.e., is a synchronous reactive system.
- A *TransLucid functional system* maps arrays of *finite* dimensionality and extent to other such arrays; callable from host language such as `C++`, `go` or `rust`.
- A *clock dimension*, which may not be queried into the future, can drive a whole family of functional systems.
- An *object-oriented system* maps sequences of sets of declarations to other such sequences, generated as side-effects during the evaluation of the output. This is Alan Kay's vision.
- A *declaration multiplexer* passes output declarations from one system as input declarations to other systems.

Status

- Denotational semantics is finally correct.
  Properties to be proven written, proofs still not all written.
- Operational semantics for order-0 and order-1 written.
  Proofs of validity need to be written.
- New interpreter needs to be written.
- Examples need to be developed.
- Old Web site: `http://translucid.web.cse.unsw.edu.au`
- Dormant blog: `http://cartesianprogramming.com`
- Email: `johnplaice@gmail.com`

TransLucid

This insight, which expresses itself by what is called Imagination, is a very high sort of seeing, which does not come by study, but by the intellect being where and what it sees; by sharing the path or circuit of things through forms, and so making them translucid to others.

The Poet, Ralph Waldo EMERSON, 1844.