TransLucid: Higher-order Indexical ISWIM

John Plaice*

Version 1, 21 June 2020
Version 2, 9 July 2020

Abstract

We present a higher-order functional language in which variables define arbitrary-dimensional entities, and a multidimensional runtime context is used to index the variables. We give an intuitive presentation of the language, present the denotational semantics, and demonstrate how function applications over these potentially infinite data structures can be transformed into manipulations of the runtime context.

1 Introduction

We present a functional language called TransLucid defining arbitrary-dimensional variables, which are evaluated in a lazy manner. The indexing of these variables is done through a runtime multidimensional context, an unordered set of (*dimension*, *ordinate*) pairs. Structuring of expressions is done with `where` clauses, in which one may introduce both dimension identifiers and variable identifiers. To simplify the presentation, we have chosen to make TransLucid untyped; all results presented here generalize naturally to the typed situation.

The paper will consist of two main parts: (1) an informal introduction to the language, through the presentation of more and more complex examples; and (2) a formal presentation of the language, with both syntax and denotational semantics.

Variables and expressions are indexed by the runtime context, which can be perturbed by changing some of the (*dimension*, *ordinate*) pairs defining it. Conceptually, all variables and expressions var in *all* possible dimensions. In practice, TransLucid is designed so that only a finite set of dimensions may affect any specific computation.

We introduce TransLucid with the problem of multiplying two matrices, $A_{row:m \times col:p}$ and $B_{row:p \times col:n}$, varying in dimensions *col* and *row*. The expression below defines their multiplication (*rotate* and *sum* will be defined in §2):

*UNSW Sydney, Australia; GrammaTech, USA; johnplaice@gmail.com

```
multiply.row.col p A B
where
  fun multiply.d_r.d_c k X Y = W
  where
    dim d ← 0
    var X' = rotate.d_c.d X
    var Y' = rotate.d_r.d Y
    var Z = X' × Y'
    var W = sum.d k Z
  end
end
```

where

- function *multiply* has *five* arguments;
- the two formal paramers d_r and d_c are *dimensional parameters*, and are introduced with a dot; these two parameters define *dimensions of variance* for X and Y;
- the three formal parameters p, A and B are *intensional parameters*, and are introduced with a space;
- actual parameters *row*, *col*, p, A and B are passed in to *multiply*, respectively, as formal parameters d_r, d_c, k, X and Y;
- d is an additional, temporary dimension identifier;
- X' corresponds to changing the d_r and d_c variance in X to d_r and d variance;
- Y' corresponds to changing the d_r and d_c variance in Y to d and d_c variance;
- Z is a 3-dimensional data structure corresponding to the pointwise multiplication of X' and Y';
- W corresponds to the collapsing through summation of the first k entries in the d direction of Z.

In the semantics for the language, in the environment providing meaning to all identifiers, each dimension identifier is mapped to a dimension, which is a natural number. When a `where` with a local dimension identifier is entered, a new dimension must be allocated. This is done by choosing the smallest dimension not currently being used.

The trickiest part of the semantics is for functions. In particular, the body of a function is evaluated with respect to a context that combines elements of the context in which the abstraction was created and the context in which the application takes place. The semantics ensures that there is never any confusion between the two.

2 Motivating examples

Like in all languages derived from ISWIM, a program in TransLucid is an expression. All expressions in TransLucid are manipulated in an arbitrary-dimensional *context*, which corresponds to an index in the Cartesian coördinate system. As an expression is evaluated, the context may be *queried*, dimension by dimension, in order to produce an answer. In so doing, other expressions may need to be evaluated in other contexts.

2.1 Constants

The simplest expression in TransLucid is the *constant*. If we consider the value 42, its value is 42, whatever the context. Below, we show the variance of 42 if we allow dimension δ_1 to vary:

	δ_1 $\rightarrow$						
42	0	1	2	3	4	5	$\cdots$
	42	42	42	42	42	42	$\cdots$

What this table means is that in context $\{\delta_1 \mapsto 0\}$, i.e., where the ordinate of dimension δ_1 is the value 0, the value of expression '42' is 42. The same holds true whatever the value of $j \in \mathbb{N}$, for all contexts $\{\delta_1 \mapsto j\}$.

The next example uses two dimensions. If, for example, the context is $\{\delta_0 \mapsto 0, \delta_1 \mapsto 0\}$, i.e., where both dimensions δ_0 and δ_1 take on the value 0, the value of expression '42' is still 42. The same holds true for all contexts $\{\delta_0 \mapsto i, \delta_1 \mapsto j\}$, where $i, j \in \mathbb{N}$.

	δ_1 $\rightarrow$						
42	0	1	2	3	4	5	$\cdots$
$\delta_0 \downarrow 0$	42	42	42	42	42	42	$\cdots$
1	42	42	42	42	42	42	$\cdots$
2	42	42	42	42	42	42	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The 'variance' of '42' in further dimensions would still yield, of course, the same result.

2.2 Dimension queries

For it to be possible for the context to affect the result of the evaluation of an expression, the context must be *queried*, using the `#` operator. For expression '`#`d_1', assuming that dimension identifier d_1 is mapped to dimension δ_1, then the expression varies as δ_1 varies:

	δ_1 $\rightarrow$						
`#`d_1	0	1	2	3	4	5	$\cdots$
	0	1	2	3	4	5	$\cdots$

In, for example, context $\{\delta_1 \mapsto 4\}$, the value of expression '`#`d_1' is 4. In fact, for every context $\{\delta_1 \mapsto j\}$, where $j \in \mathbb{N}$, the value of `#`d_1 is j.

Below, we see the variance of '$\#d_1$' in two dimensions, where dimension identifier d_0 is mapped to dimension δ_0. There we can see that no matter what the ordinate for dimension δ_0, only the ordinate for dimension δ_1 in the current context is of relevance for evaluating expression '$\#d_1$'. For all contexts $\{\delta_0 \mapsto i, \delta_1 \mapsto j\}$, where $i, j \in \mathbb{N}$, the value of $\#d_1$ is j.

	δ_1 $\rightarrow$						
$\#d_1$	0	1	2	3	4	5	$\cdots$
$\delta_0 \downarrow 0$	0	1	2	3	4	5	$\cdots$
1	0	1	2	3	4	5	$\cdots$
2	0	1	2	3	4	5	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

2.3 Pointwise operators

As in all languages, TransLucid expressions allow the use of operators such as + for addition, × for multiplication, and so on. In TransLucid, these operators are applied *pointwise* to their arguments. In other words, in a given context κ, the expression $E_1 + E_2$ yields the sum of E_1 in κ and of E_2 in κ. Below is the variance of $\#d_0 + \#d_1$ in two dimensions:

	δ_1 $\rightarrow$						
$\#d_0 + \#d_1$	0	1	2	3	4	5	$\cdots$
$\delta_0 \downarrow 0$	0	1	2	3	4	5	$\cdots$
1	1	2	3	4	5	6	$\cdots$
2	2	3	4	5	6	7	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

In context $\{\delta_0 \mapsto i, \delta_1 \mapsto j\}$, for all $i, j \in \mathbb{N}$, $\#d_0 + \#d_1$ has the value $i + j$.

2.4 Context changes

If the context can be queried in TransLucid, then it also needs to be changeable. This is done by placing a tuple in square brackets, usint it to change the current context before continuing on with the evaluation of the expression. Below, the expression $\#d_0 + \#d_1$ is evaluated in a new context, which is created by incrementing dimension-δ_1's ordinate by 1.

	δ_1 $\rightarrow$						
$(\#d_0 + \#d_1)\ [d_1 \leftarrow \#d_1 + 1]$	0	1	2	3	4	5	$\cdots$
$d_0 \downarrow 0$	1	2	3	4	5	6	$\cdots$
1	2	3	4	5	6	7	$\cdots$
2	3	4	5	6	7	8	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

So, for all $i, j \in \mathbb{N}$, expression $(\#d_0 + \#d_1)\ [d_1 \leftarrow \#d_1 + 1]$ therefore evaluates to $i + j + 1$ in context $\{\delta_0 \mapsto i, \delta_1 \mapsto j\}$.

2.5 Factorial: version one

TransLucid, of course, needs variables, defined through equations. We introduce these with the factorial function, presented as a variable varying in dimension δ_1, whose first few entries can be found below:

	δ_1 $\rightarrow$								
fact	0	1	2	3	4	5	6	7	$\cdots$
	1	1	2	6	24	120	720	5040	$\cdots$

The definition in TransLucid is recursive, with a base case and an inductive case.

$$\begin{aligned}\texttt{var } \mathit{fact} = \texttt{if \#}d_1 \equiv 0 \texttt{ then } 1 \\ \texttt{else \#}d_1 \times \big(\mathit{fact}\ [d_1 \leftarrow \texttt{\#}d_1 - 1]\big)\end{aligned}$$

2.6 Ackermann: version one

The Ackermann function is one of the first recursive functions discovered that is not primitive recursive. It grows so fast that in general it cannot be computed once its first argument is greater than 3. Here it is presented as a variable varying in dimensions δ_0 and δ_1.

	δ_1 $\rightarrow$						
ack	0	1	2	3	4	5	$\cdots$
$\delta_0 \downarrow 0$	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	$\cdots$					
$\vdots$	$\ddots$						

Here is the definition for Ackermann in TransLucid.

$$\begin{aligned}&\texttt{var } \mathit{ack} = \\ &\quad \texttt{if \#}d_0 \equiv 0 \texttt{ then \#}d_1 + 1 \\ &\quad \texttt{else if \#}d_1 \equiv 0 \texttt{ then } \mathit{ack}\ [d_0 \leftarrow \texttt{\#}d_0 - 1, d_1 \leftarrow 1] \\ &\quad \texttt{else } \mathit{ack}\ \big[d_0 \leftarrow \texttt{\#}d_0 - 1, d_1 \leftarrow \mathit{ack}\ [d_1 \leftarrow \texttt{\#}d_1 - 1]\big]\end{aligned}$$

In the general (`else`) case, note that the nested context change is only changing the value for dimension δ_1, since the value for dimension δ_0 need not be changed. This is similar to what happens with differential equations, in which only the dimensions of relevance are written down.

2.7 Standard functions

A function can carry both *value parameters* and *named parameters*. Below are some standard TransLucid functions.

$$
\begin{aligned}
&\texttt{fun } \mathit{index}.d = \#d + 1\\
&\texttt{fun } \mathit{first}.d\ X = X\ [d \leftarrow 0]\\
&\texttt{fun } \mathit{next}.d\ X = X\ [d \leftarrow \#d + 1]\\
&\texttt{fun } \mathit{prev}.d\ X = X\ [d \leftarrow \#d - 1]\\
&\texttt{fun } \mathit{fby}.d\ X\ Y = \texttt{if } \#d < 1 \texttt{ then } X \texttt{ else } \mathit{prev}.d\ Y
\end{aligned}
$$

Each of these declared functions has a single dimensional parameter d. Function *index* takes no intensional parameters, while *first*, *prev*, and *next* take one intensional parameter each, and *fby* takes two intensional parameters.

2.8 Factorial: version two

If we wish to write factorial as a function, we write it as taking a value parameter.

$$
\begin{aligned}
&\texttt{fun } \mathit{fact}\ n = f\\
&\texttt{where}\\
&\quad \texttt{dim } d \leftarrow n\\
&\quad \texttt{var } f = 1\ \mathit{fby}.d\ (\mathit{index}.d \times f)\\
&\texttt{end}
\end{aligned}
$$

It uses a local dimension identifier d, whose corresponding dimension δ's ordinate is initially set to n. The stream f varies in dimension d. Note that $\mathit{index}.d$ increments the δ-ordinate, while the second argument of $\mathit{fby}.d$ decrements the δ-ordinate, so the two cancel each other out, yielding $\#d$.

To simplify the discussion below, for a dimension identifier d, which has been mapped to dimension δ, we will interchangeably use the terms d-ordinate and δ-ordinate.

2.9 Ackermann: version two

Ackermann takes two value parameters, and is defined using two local dimensions.

$$
\begin{aligned}
&\texttt{fun } \mathit{ack}\ m\ n = f\\
&\texttt{where}\\
&\quad \texttt{dim } d_m \leftarrow m\\
&\quad \texttt{dim } d_n \leftarrow n\\
&\quad \texttt{var } f = \ \mathit{next}.d_n(\mathit{index}.d_n)\ \mathit{fby}.d_m\\
&\qquad\qquad \big((\mathit{next}.d_n\ f)\ \mathit{fby}.d_n\ (f\ [d_n \leftarrow \mathit{next}.d_m\ f])\big)\\
&\texttt{end}
\end{aligned}
$$

Note the elimination of explicit manipulation of dimensions.

2.10 Sieve of Eratosthenes

The sieve of Erastosthenes generates a stream in dimension d of the prime numbers. It is built using a local dimension d'. The first line is the naturals ≥ 2, and subsequent lines are the previous line without the multiples of the first element of the previous line.

```
fun sieve.d = f
where
  dim d' ← 0
  var f = (#d' + 2) fby.d
          (f wvr.d' (f mod (first.d' f) ≠ 0))
end
```

The function $wvr.d$ is a *filter* in the d dimension. It returns a stream in the d dimension that retains elements of the X input when the corresponding Y element is true.

```
fun wvr.d X Y =
  if first.d Y
  then X fby.d ((next.d X) wvr.d (next.d Y))
  else (next.d X) wvr.d (next.d Y)
```

Each row corresponds to the removal of the multiples of the first element in the previous row. The sequence of primes is formed by the first column.

		$d' \rightarrow$								
	f	0	1	2	3	4	5	6	7	$\cdots$
$d\downarrow$	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	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

2.11 Matrix multiplication

Here is the matrix multiplication example from the introduction:

```
fun multiply.dr.dc k X Y = W
where
  dim d ← 0
  var X' = rotate.dc.d X
  var Y' = rotate.dr.d Y
  var Z = X' × Y'
  var W = sum.d k Z
end
```

Function $rotate.d_1.d_2$ changes variance in dimension d_1 to variance in dimension d_2, while function $sum.d_x.n\ X$ adds up the first n elements in direction d_x of stream X.

$$
\begin{array}{l}
\texttt{fun}\ rotate.d_1.d_2\ X = X\ [d_1 \leftarrow \#d_2] \\
\texttt{fun}\ sum.d_x\ n\ X = Y\ [d_x \leftarrow n] \\
\texttt{where} \\
\quad \texttt{var}\ Y = 0\ fby.d_x\ (X + Y) \\
\texttt{end}
\end{array}
$$

2.12 Tournament computation

The factorial function was first presented in an iterative manner. It can also be computed in a *tournament computation* style, in which computation takes place in a series of stages, each stage applying the same operation upon pairs of adjacent elements.

For example, for the factorial function, if we let $n = 6$, computation proceeds as in the following table. The d_2 dimension corresponds to the successive stages. Initially, the sequence from 1 to 6 is embedded in a sea of 1s. Each element of the next stage is the result of the multiplication of two elements from the preceding stage. The result is, of course, 720.

	$d_1 \rightarrow$									
	0	1	2	3	4	5	6	7	8	$\cdots$
$d_2 \downarrow 0$	1	1	2	3	4	5	6	1	1	$\cdots$
1	1	6	20	6	1	1	$\cdots$			
2	6	120	1	1	$\cdots$					
3	720	1	1	$\cdots$						

To illustrate how this will take place in TransLucid, we use a general higher-order function called *tournament*:

$$
\begin{array}{l}
\texttt{fun}\ fact\ n = tournament\ n\ 1\ index\ mul \\
\texttt{where} \\
\quad \texttt{fun}\ tournament\ n\ val\ generate\ f = F \\
\quad \texttt{where} \\
\quad\quad \texttt{dim}\ d_1 \leftarrow 0 \\
\quad\quad \texttt{dim}\ d_2 \leftarrow \text{ilog}\ n \\
\quad\quad \texttt{var}\ F = (default.d_1\ n\ val\ generate)\ fby.d_2\ \big(f\ (lChild.d_1\ F)\ (rChild.d_1\ F)\big) \\
\quad \texttt{end} \\
\quad \texttt{fun}\ mul\ X\ Y = X \times Y \\
\quad \texttt{fun}\ lChild.d\ X = X\ [d \leftarrow \#d \times 2] \\
\quad \texttt{fun}\ rChild.d\ X = X\ [d \leftarrow \#d \times 2 + 1] \\
\quad \texttt{fun}\ default.d\ n\ val\ generate = \texttt{if}\ \#d = 0 \vee \#d > n\ \texttt{then}\ val\ \texttt{else}\ generate.d \\
\texttt{end}
\end{array}
$$

We explain below:

- Functions *lChild* and *rChild* reach the appropriate element from the previous stage.
- Function *default* produces a sequence in the d direction. Elements from $\{d \mapsto 1\}$ to $\{d \mapsto n\}$ are created by the *generate* function passed in as argument. Elements $\{d \mapsto 0\}$ and $\{d \mapsto m\}$, $m > n$, are all set to the default value *val*.
- Function *tournament* uses two local dimensions, d_1 and d_2, to set up a tournament.

2.13 Families of functions

In this section, we present two approaches to computing the power function. In the first, *power* takes two arguments, n and x, and looks up the n-th element in the sequence P.

$$
\begin{array}{l}
power\ n\ x = P \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ P = 1\ cby.d\ \big((lParent.d\ P \times lParent.d\ P)\ alt.d\ (x \times prev.d\ P)\big) \\
\texttt{end}
\end{array}
$$

The *power* function uses the following three auxiliary functions:

$$
\begin{array}{l}
\texttt{fun}\ lParent.d\ X = X\ [d \leftarrow \#d/2] \\
\texttt{fun}\ cby.d\ X\ Y = \texttt{if}\ \#d < 1\ \texttt{then}\ X\ \texttt{else}\ Y \\
\texttt{fun}\ alt.d\ X\ Y = \texttt{if}\ \#d\ mod\ 2 = 0\ \texttt{then}\ X\ \texttt{else}\ Y
\end{array}
$$

Another approach is to create a sequence of functions, i.e., a *family* of functions.

$$
\begin{array}{l}
power'\ n = P \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ P' = (\texttt{fn}\ x \rightarrow 1)\ cby.d\ \big((\texttt{fn}\ x \rightarrow lParent.d\ P'\ x \times lParent.d\ P'\ x)\ alt.d \\
\qquad\qquad\qquad\qquad (\texttt{fn}\ x \rightarrow x \times prev.d\ P'\ x)\big) \\
\texttt{end}
\end{array}
$$

Variable P' is a sequence of functions, varying in the d dimension.

2.14 Families of filters

The next example is a family of filters. The function *nnext* takes the intensional parameter n as argument.

$$
\begin{array}{l}
nnext\ n\ = N \\
\texttt{where} \\
\quad \texttt{dim}\ d_n \leftarrow n \\
\quad \texttt{var}\ N = (\texttt{fn}.d\ X \rightarrow X)\ fby.d_n\ \big(\texttt{fn}.d\ X \rightarrow next.d\ (N.d\ X)\big) \\
\texttt{end}
\end{array}
$$

It returns the n-th element in the d-sequence

$$N = \langle id, next.d, next.d^2, \ldots, next.d^n, \ldots \rangle.$$

2.15 Taylor series expansion

Our last example is to compute the Taylor series expansion for a function f around point a for point x.

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^{\mathrm{n}}$$

Function *taylor* takes as input a stream *derivs* of derivatives of f at point a in direction d.

```
fun taylor.d a x derivs = F
where
  var F = sum.d (index.d) G
  var G = derivs / (fact (#d)) × (pow (#d) (x − a))
end
```

The Taylor series expansion for the sine function around integral multiples of 2π yields:

```
taylor.d a x sinderivs
where
  var sinderivs =
        0 fby.d (1 fby.d (0 fby.d (−1 fby.d sinderivs)))
end
```

2.16 On intensionality

The fact that a variable, conceptually varies in all dimensions forces the programmer to think in a *intensional* manner, as it is in general not possible, nor desirable, to enumerate all of these dimensions. For example, the phrase "*Five degrees less than yesterday's temperature*" could be written as the expression

$$(t-5)\ [date \leftarrow \#date - 1]$$

where t is the temperature and dimension *date* keeps track of the current date. Note that this expression does not make explicit where t is to be evaluated, which might vary according to other dimensions, such as *latitude*, *longitude*, *altitude*, *timeOfDay*, and so forth. This aproach is consistent with the use of dimensions in differential dimensions, where only relevant dimensions are written down.

3 Semantics

3.1 Notation for function manipulation

Definition 1. *Let S be a set and $\bot \notin S$. We define $S_\bot = S \cup \{\bot\}$. The trivial order $\sqsubseteq$ over $S_\bot$ is given by*

$$s \sqsubseteq s' \quad \textit{iff} \quad s = \bot \vee s = s'.$$

Definition 2. *Let A and B be two sets. A* total function *f from A to B is written $f : A \to B$. If $\sqsubseteq$ is defined over B, then the order $\sqsubseteq$ over $A \to B$ is given by*

$$f \sqsubseteq f' \quad \textit{iff} \quad \forall a \in A,\ f(a) \sqsubseteq f'(a).$$

Definition 3. *Let A and B be two sets. A* partial function *f from A to B is written $f : A \rightarrowtail B$.*

When f is a function with finite domain $\{v_i\}_{i \in m}$, we will write it as its graph

$$\{v_i \mapsto f(v_i)\}_{i \in m}.$$

When the graph is empty, we will write it as $\emptyset$.

Definition 4. *Let $f : A \to B$ and $g : A \rightarrowtail B$. The* perturbation of f by g, *written $f \dagger g$, is defined by:*

$$(f \dagger g)(v) \quad = \quad \begin{cases} g(v), & v \in \operatorname{dom} g \\ f(v), & \textit{otherwise.} \end{cases}$$

Definition 5. *Let $f : A \to B$, and let $S \subseteq A$. The* domain restriction of f to S, *written $f \mid S$, is defined by*

$$(f \mid S)(v) \quad = \quad \begin{cases} f(v), & v \in S. \end{cases}$$

3.2 Syntax

Definition 6. *The* identifiers *($s \in S$) that can appear in expressions are:*

- *constant symbols ($c \in C$),*
- *dimension identifiers ($d \in D$),*
- *variable (and function) identifiers ($x \in X$),*

where C, D and X are mutually disjoint and $S = C \cup D \cup X$.

Definition 7. *The* abstract syntax *for an expression* $E \in \mathbf{Expr}$ *is as follows:*

$$
\begin{array}{rll}
E ::= & x & \text{variable } (x \in X) \\
| & c(E, \ldots) & \text{constant } (c \in C) \\
| & \texttt{\#}d & \text{context lookup } (d \in D) \\
| & E\ [d \leftarrow E, \ldots] & \text{context perturbation} \\
| & E.(d, \ldots)(E, \ldots) & \text{function application} \\
| & \texttt{fn}.(d, \ldots)(x, \ldots) \rightarrow E & \text{function abstraction} \\
| & E\ \texttt{where} & \text{local declarations} \\
& \quad \texttt{dim}\ d \leftarrow E\ , \ldots & \text{dimensions} \\
& \quad \texttt{var}\ x = E\ , \ldots & \text{variables} \\
& \quad \texttt{fun}\ x.(d, \ldots)(x, \ldots) = E\ , \ldots & \text{functions} \\
& \texttt{end} &
\end{array}
$$

3.3 Semantics

Definition 8. *Let* $V \supseteq \{true, false\}$ *be a set of atomic values, where* $\perp \notin V$*. The* domains defined over V *are:*

$$
\begin{aligned}
\mathbf{V} &= V_\perp && (1)\\
\mathbf{A} &= \textstyle\bigcup_{m \in \omega} (\mathbf{V}^m \rightarrow \mathbf{V}) && (2)\\
\mathbf{K} &= \mathbb{N} \rightarrow \mathbf{V} && (3)\\
\mathbf{H} &= \mathbf{K} \rightarrow (\mathbf{V} \cup \mathbf{F}) && (4)\\
\mathbf{F} &= \textstyle\bigcup_{m,n \in \omega} \mathbb{N}^m \times \mathbf{H}^n \rightarrow \mathbf{H} && (5)\\
\mathbf{Env} &= (C \rightarrow \mathbf{A}) \cup (D \rightarrow \mathbb{N}_\perp) \cup (X \rightarrow \mathbf{H}) && (6)
\end{aligned}
$$

- $\mathbf{V}$ $(\ni v)$ *is the set of* basic values.
- $\mathbf{A}$ $(\ni op)$ *is the set of* atomic operators of arity $m \in \omega$*. This set will typically include the standard arithmetic and comparison operators, as well as the* `if`–`then`–`else` *operator.*
- $\mathbf{K}$ $(\ni \kappa)$ *is the set of* contexts*. Elements of the domain of a context are called* dimensions*. Elements of the range of a context are called* ordinates*. Thus a context consists of a set of* $(dimension, ordinate)$ *pairs.*
- $\mathbf{H}$ $(\ni \eta)$ *is the set of* intensions*, mapping contexts to values.*
- $\mathbf{F}$ $(\ni f)$ *is the set of* indexed functions of dimension arity m and intension arity n*.*
- $\mathbf{V} \cup \mathbf{F}$ *is the set of* first-class values*, which are manipulated by expressions.*
- $\mathbf{Env}$ *is the set of* environments*.*
- $\perp_{\mathbf{H}} = \lambda\kappa.\perp$ *is the completely undefined intension.*
- *The order* $\sqsubseteq$ *over these domains is given using Definitions 1–2.*

Definition 9. *For* $\kappa \in \mathbf{K}$, $\mathrm{dims}(\kappa)$ *is the set of* active dimensions *of* κ, *i.e.,*

$$\mathrm{dims}(\kappa) = \big\{\delta \in \mathbb{N} \mid \kappa(\delta) \neq \bot\big\}.$$

The set $\mathbf{K}^f$ *consists of all those* $\kappa \in \mathbf{K}$ *that are* finite-dimensional, *i.e.,*

$$\mathrm{card}\big(\mathrm{dims}(\kappa)\big) < \infty.$$

For $\kappa \in \mathbf{K}^f$, $\mathrm{rk}(\kappa)$ *is the least dimension greater than all of the active dimensions of* κ, *i.e.,*

$$\forall \delta \in \mathrm{dims}(\kappa), \delta < \mathrm{rk}(\kappa).$$

Definition 10. *The set* $\mathbf{H}^m$ *consists of all those* $\eta \in \mathbf{H}$ *that are* monotonic, *i.e.,*

$$\forall \kappa, \kappa',\ \kappa \sqsubseteq \kappa' \implies \eta(\kappa) \sqsubseteq \eta(\kappa').$$

For $\eta \in \mathbf{H}^m$, $\mathrm{minctxts}(\eta)$ *is the set of* minimal contexts *for* η, *i.e.,*

$$\mathrm{minctxts}(\eta) = \big\{\kappa \in \mathbf{K} \mid \eta(\kappa) \neq \bot \wedge \forall \kappa' \sqsubset \kappa,\ \eta(\kappa') = \bot\big\}.$$

Definition 11. *For* $\eta \in \mathbf{H}^m$, $\mathrm{dims}(\eta)$ *is the set of* active dimensions *of* η, *i.e.,*

$$\mathrm{dims}(\eta) = \bigcup\big\{\mathrm{dims}(\kappa) \mid \kappa \in \mathrm{minctxts}(\eta)\big\}.$$

The set $\mathbf{H}^f$ *consists of all those* $\eta \in \mathbf{H}^m$ *that are* finite-dimensional, *i.e.,*

$$\mathrm{card}\big(\mathrm{dims}(\eta)\big) < \infty.$$

For $\eta \in \mathbf{H}^f$, $\mathrm{rk}(\eta)$ *is the least dimension greater than all of the active dimensions of* η, *i.e.,*

$$\forall \delta \in \mathrm{dims}(\eta), \delta < \mathrm{rk}(\eta).$$

Definition 12. *For* $\zeta \in \mathbf{Env}$, $\mathrm{dims}(\zeta)$ *is the set of* active dimensions *of* ζ, *i.e.,*

$$\mathrm{dims}(\zeta) = \big\{\zeta(d) \mid d \in D \wedge \zeta(d) \neq \bot\big\} \cup \textstyle\bigcup_{\eta \in \{\zeta(x) \mid x \in X\}} \mathrm{dims}(\eta).$$

The set $\mathbf{Env}^f$ *consists of all those* $\eta \in \mathbf{Env}$ *that are* finite-dimensional, *i.e.,*

$$\mathrm{card}\big(\mathrm{dims}(\zeta)\big) < \infty.$$

For $\zeta \in \mathbf{Env}^f$, $\mathrm{rk}(\zeta)$ *is the least dimension greater than all of the active dimensions of* ζ, *i.e.,*

$$\forall \delta \in \mathrm{dims}(\zeta), \delta < \mathrm{rk}(\zeta).$$

Definition 13. *Let* $E \in \mathbf{Expr}$, $\zeta \in \mathbf{Env}^f$, *and* $\kappa \in \mathbf{K}^f$. *Then the* semantics *for* E *with respect to* ζ *and* κ *is given by*

$$[\![E]\!]\zeta\kappa,$$

where the rules for

$$[\![\cdot]\!] : \mathbf{Expr} \to \mathbf{Env}^f \to \mathbf{K}^f \to \mathbf{V} \tag{7}$$

are given in Equations 8–14.

$$\begin{aligned}
[\![x]\!]\zeta\kappa &= \zeta(x)(\kappa) && (8)\\
[\![c(E_i)_{i\in m}]\!]\zeta\kappa &= \zeta(c)\big([\![E_i]\!]\zeta\kappa\big)_{i\in m} && (9)\\
[\![\#d]\!]\zeta\kappa &= \kappa\big(\zeta(d)\big) && (10)\\
[\![E \ \texttt{@} \ [d_i \leftarrow E_i]_{i\in m}]\!]\zeta\kappa &= [\![E]\!]\zeta\big(\kappa \dagger \big\{\zeta(d_i) \mapsto [\![E_i]\!]\zeta\kappa\big\}_{i\in m}\big) && (11)
\end{aligned}$$

$$\left[\!\!\left[\begin{array}{ll}
E \ \texttt{where} & \\
\quad \underset{i\in p}{\texttt{dim}} & d_i \leftarrow E_i \\
\quad \underset{j\in p+1..q}{\texttt{var}} & x_j = E_j \\
\quad \underset{k\in q+1..r}{\texttt{fun}} & x_k.(d_{i,k})_{i\in m_k}(x_{j,k})_{j\in n_k} = E_k \\
\texttt{end} &
\end{array}\right]\!\!\right] \zeta\kappa \tag{12}$$

$$\begin{aligned}
= \quad \text{let } \nu &= \max\big(\mathrm{rk}(\zeta), \mathrm{rk}(\kappa)\big)\\
\delta_i &= \nu + i, \quad i \in p\\
\kappa' &= \kappa \dagger \big\{\delta_i \mapsto [\![E_i]\!]\zeta\kappa\big\}_{i\in p}\\
\zeta_0 &= \zeta \dagger \big\{d_i \mapsto \delta_i\big\}_{i\in p}\\
&\qquad \dagger \big\{x_j \mapsto \bot_{\mathbf{H}}\big\}_{j\in p+1..q}\\
&\qquad \dagger \big\{x_k \mapsto \bot_{\mathbb{N}^{m_k}\times\mathbf{H}^{n_k}\to\mathbf{H}}\big\}_{k\in q+1..r}\\
\zeta_{\alpha+1} &= \zeta \dagger \big\{d_i \mapsto \delta_i\big\}_{i\in p}\\
&\qquad \dagger \big\{x_j \mapsto [\![E_j]\!]\zeta_\alpha)\big\}_{j\in p+1..q}\\
&\qquad \dagger \big\{x_k \mapsto [\![\texttt{fn}.(d_{i,k})_{i\in m_k}(x_{j,k})_{j\in n_k} \to E_k]\!]\zeta_\alpha\big\}_{k\in q+1..r}\\
\zeta' &= \lim_{\alpha\to\infty} \zeta_\alpha\\
\text{in } [\![E]\!]\zeta'\kappa'
\end{aligned}$$

$$[\![E.(d'_i)_{i\in m}(E'_j)_{j\in n}]\!]\zeta\kappa = \big([\![E]\!]\zeta\kappa\big)\big(\zeta(d'_i)\big)_{i\in m}\big([\![E'_j]\!]\zeta\big)_{j\in n}\big(\kappa\big) \tag{13}$$

$$[\![\texttt{fn}.(d_i)_{i\in m}(x_j)_{j\in n}]\!]\zeta\kappa = \lambda(\delta_i)_{i\in m}.(\eta_j)_{j\in n}.\kappa'. \tag{14}$$

$$\begin{aligned}
&\text{let } \nu = \max\big(\mathrm{rk}(\zeta), \mathrm{rk}(\kappa), \mathrm{rk}(\kappa'), \delta_i, \mathrm{rk}(\eta_j)\big)_{i\in m, j\in n}\\
&\quad \delta'_i = \nu + i, \quad i \in m\\
&\quad \kappa'' = \kappa \dagger \big\{\delta'_i \mapsto \kappa'(\delta_i)\big\}_{i\in m}\\
&\quad \zeta' = \zeta \dagger \big\{d_i \mapsto \delta'_i\big\}_{i\in m}\\
&\qquad\qquad \dagger \big\{x_j \mapsto \lambda\kappa_j.\eta_j\big(\kappa' \dagger \big\{\delta_i \mapsto \kappa_j(\delta'_i)\big\}_{i\in m}\big)\big\}_{j\in n}\\
&\text{in } [\![E]\!]\zeta'\kappa''
\end{aligned}$$

We explain all of the different cases for Equations 8–14 below.

(8) *Variable*: identifier x is looked up in environment ζ to produce an intension η, which is applied to the context κ to produce a value v.

(9) *Constant*: each expression E_i is evaluated with respect to environment ζ and context κ to produce an atomic value v_i. Constant symbol c is looked up in environment ζ to produce an atomic function op, which is then applied to the v_i to produce an atomic value v. Should $m = 0$, the value is op itself.

(10) *Context lookup*: dimension identifier d is mapped to dimension $\delta = \zeta(d)$. The ordinate for δ is $\kappa(\delta)$.

(11) *Context perturbation*: each expression E_i is evaluated with respect to environment ζ and context κ to produce atomic value v_i. Context κ is then perturbed by changing, for each i, the ordinate for dimension $\delta_i = \zeta(d_i)$ to v_i. Expression E is then evaluated in this new context.

(12) *Local declarations*: we give the semantics for an expression of the form:

$$\begin{array}{ll}
E\ \texttt{where} & \\
\quad \underset{i\in p}{\texttt{dim}} & d_i \leftarrow E_i \\
\quad \underset{j\in p+1..q}{\texttt{var}} & x_j = E_j \\
\quad \underset{k\in q+1..r}{\texttt{fun}} & x_k.(d_{i,k})_{i\in m_k}(x_{j,k})_{j\in n_k} = E_k \\
\texttt{end} &
\end{array}$$

Given an environment ζ and a context κ, expression E must be evaluated with respect to a new environment ζ' and a new context κ'.

For each of the new dimension identifiers d_i, a new dimension δ_i needs to be allocated. The first freely available dimension ν is the maximum of the ranks of ζ and κ. For

dimension identifier d_i, the new dimension is $\delta_i = \nu{+}i$. The new context κ' perturbs κ by mapping δ_i to $[\![E_i]\!]\zeta\kappa$.

A sequence of environments ζ_α, $\alpha \in \mathbb{N}$, is initialized by creating environment

$$\begin{aligned}\zeta_0 = \zeta \dagger \left\{d_i \mapsto \delta_i\right\}_{i\in p} \\ \dagger \left\{x_j \mapsto \bot_{\mathbf{H}}\right\}_{j\in p+1..q} \\ \dagger \left\{x_k \mapsto \bot_{\mathbb{N}^{m_k}\times\mathbf{H}^{n_k}\to\mathbf{H}}\right\}_{k\in q+1..r.}\end{aligned}$$

Environment $\zeta_{\alpha+1}$ is created from the meanings of the variables x_j in environment ζ_α:

$$\begin{aligned}\zeta_{\alpha+1} = \zeta \dagger \left\{d_i \mapsto \delta_i\right\}_{i\in p} \\ \dagger \left\{x_j \mapsto [\![E_j]\!]\zeta_\alpha)\right\}_{j\in p+1..q} \\ \dagger \left\{x_k \mapsto [\![\texttt{fn}.(d_{i,k})_{i\in m_k}(x_{j,k})_{j\in n_k} \to E_k]\!]\zeta_\alpha\right\}_{k\in q+1..r.}\end{aligned}$$

Environment ζ' is the limit of the sequence of ζ_α:

$$\zeta' = \lim_{\alpha\to\infty} \zeta_\alpha.$$

Finally, expression E is evaluated with respect to environment ζ' and context κ' to produce a value v.

(13) *Function application*: a function f is produced by evaluating expression E with respect to ζ and κ. For each i, environment ζ is applied to actual dimensional parameter d_i to produce dimension δ_i. For each j, the meaning of actual expression parameter E_j is applied to ζ to produce intension η_j. Then, function f is applied to the dimensional actual parameters δ_i, the intensional actual parameters η_j, and the abstraction context κ.

(14) *Function abstraction*: the evaluation of an abstraction expression takes place in an environment ζ and an *abstraction context* κ. The result is an abstraction that takes as input a set of dimensional actual parameters δ_i, a set of intensional actual parameters η_j, and an *application context* κ'. The technical difficulty to be resolved is the interaction between κ and κ' in the evaluation of the function body E.

For each formal dimensional parameter d_i, a new dimension δ'_i needs to be allocated. The first freely available dimension ν is the maximum of the ranks of ζ, κ and κ', the ranks of each of the η_j, and the δ_i values themselves. For each i, the new dimension is $\delta'_i = \nu + i$.

The context κ'' of evaluation of the body E is the abstraction context κ, perturbed by mapping, for each i, the new dimension δ'_i to the ordinate of the actual dimensional parameter δ_i in application context κ':

$$\kappa'' = \kappa \dagger \left\{\delta'_i \mapsto \kappa'(\delta_i)\right\}_{i\in m.}$$

As E is being evaluated, κ'' will possibly be modified, and it is important that when a variable x is encountered, that it be evaluated to the appropriate context. To make this possible, the environment ζ' for evaluating E is given below:

$$\begin{aligned}\zeta' = \zeta \dagger \left\{d_i \mapsto \delta'_i\right\}_{i\in m} \\ \dagger \left\{x_j \mapsto \lambda\kappa_j.\eta_j\big(\kappa' \dagger \left\{\delta_i \mapsto \kappa_j(\delta'_i)\right\}_{i\in m}\big)\right\}_{j\in n.}\end{aligned}$$

Each formal dimensional parameter d_i is mapped to the new dimension δ'_i. Each formal intensional parameter x_j is mapped to an intension, which works as follows: should x_j need to be evaluated, it will be evaluated with respect to the *application context* κ, taking into account changes that have been made to the ordinates of the new dimensions during the evaluation up to that point. If κ_j is the context encountered when x_j is to be evaluated, the value needed is precisely

$$\eta_j\big(\kappa' \dagger \left\{\delta_i \mapsto \kappa_j(\delta'_i)\right\}_{i\in m}\big)_{j\in n.}$$

In all other cases, a variable x needs to be evaluated with respect to the *abstraction context*, taking into account changes that have been made to the ordinates of that context during the evaluation up that point. The use of κ'' also works here, since the new dimensions are all greater than the dimensions needed to evaluate x.

If we let $m = 0$, then Equations 13–14 become:

$$\llbracket E(E'_j)_{j\in n}\rrbracket\zeta\kappa = \big(\llbracket E\rrbracket\zeta\kappa\big)\big(\llbracket E'_j\rrbracket\zeta\big)_{j\in n}(\kappa) \tag{15}$$

$$\begin{aligned}\llbracket \mathtt{fn}.(x_j)_{j\in n}\rrbracket\zeta\kappa &= \lambda(\eta_j)_{j\in n}.\kappa'. \\ &\quad \text{let } \zeta' = \zeta \dagger \left\{x_j \mapsto \lambda\kappa_j.\eta_j\kappa'\right\}_{j\in n} \\ &\quad \text{in } \llbracket E\rrbracket\zeta'\kappa\end{aligned} \tag{16}$$

Note that in Equation 16, intension η_j is always applied to κ', and since $\eta_j(\kappa)$ is constant, it can be computed before entering the function. Equations 15–16 can thus be rewritten as:

$$\llbracket E(E'_j)_{j\in n}\rrbracket\zeta\kappa = \big(\llbracket E\rrbracket\zeta\kappa\big)\big(\llbracket E'_j\rrbracket\zeta\kappa\big)_{j\in n} \tag{17}$$

$$\begin{aligned}\llbracket \mathtt{fn}.(x_j)_{j\in n}\rrbracket\zeta\kappa &= \lambda(v_j)_{j\in n}. \\ &\quad \text{let } \zeta' = \zeta \dagger \left\{x_j \mapsto v_j\right\}_{j\in n} \\ &\quad \text{in } \llbracket E\rrbracket\zeta'\kappa\end{aligned} \tag{18}$$

So in the special case where there are no dimensional parameters, a TransLucid function is simply a normal function in a standard functional language.

4 Future work

In lieu of a conclusion, we focus on some possible future work.

The TransLucid language presented in this paper is a functional language manipulating intensions, which map contexts to values. Since this is a functional language, these values can include functions.

However, most modern functional languages allow the use of partially applied functions, using currification. For the moment, TransLucid does not support currification, for the following reason. Consider a function of the form

$$\texttt{fun}\ f.d\ X = E$$

and the possibility of computing $g = f.d_1$ and then later applying g to an argument E_2. It could be possible that the way the calls have been set up, that dimension identifier d_1 would not be available in the environment at the application point $g\ E_2$. To do this would require some kind of static semantics, i.e., a type checker. We leave this as future work.

The denotational semantics of TransLucid leads naturally to an implementation, although not necessarily to an efficient one. For recursively defined structures, it is common to use memoization, i.e., caching, to ensure that previously computed values do not need to be recomputed.

Memoization for variables, indexed by the current context, is a promising possibility. The difficulty, like for the semantics, lies in the implementation of functions with memoization. When the body of a function would be evaluated, and an intensional parameter encountered, it is important to keep track of the different possible invocations of the function. This can be done by hashing the expressions that are passed in as arguments. Once again, we leave this as future work.