Higher-order Multidimensional Programming (Revised)

John Plaice
plaice@cse.unsw.edu.au

Jarryd P. Beck
jarrydb@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201332
November 2013

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

We present a higher-order functional language, called TransLucid, in which expressions and variables denote intensions, which are arbitrary-dimensional arrays in which any atomic value may be used as a dimension, and a multidimensional runtime context is used to index the intensions. In addition to atomic objects, the first-class objects of TransLucid are contexts, intensions and functions.

We give an intuitive presentation of the core principles of TransLucid, present its denotational semantics, and then develop a number of programming techniques, typically avoiding recursive function calls, taking advantage of the features of the language.

1 Introduction

We present TransLucid (TL), a language with higher-order functions in which expressions and variables denote *intensions*, arbitrary-dimensional arrays of unbounded rank (dimensionality) and extent. This presentation takes place in two parts: first, we present a core TransLucid, with a restricted set of primitives, and its semantics; second, we present a more evolved language, defined through syntactic extensions, with associated programming methodology, for programming with multidimensional data.

The objectives of the paper are twofold: first, to present the solution to several open semantic and implementation problems for languages descending from Lucid [1], some dating back to the 1970s; second, to present a new approach to higher-order functional programming, in which the focus is on "flat", multidimensional, infinite data structures, rather than on hierarchical, finite ones. In order to simplify the discussion, the TL language presented here is untyped; all results should generalize to the typed situation.

Behind the design of the TL language is the manipulation of multidimensional data, which appear in many areas of computer science. Two obvious examples are in business, where online analytical processing (OLAP) [9] is widely used, and in scientific computing, where multidimensional computer simulations are the norm. Nevertheless, the goal of the programming methodology section (§5) of this paper is more modest: we wish to show that even simple problems, such as those that are studied when one first learns to program, are inherently multidimensional.

The key intuition (§2) behind TransLucid is that if in a program, an array is being manipulated, and the rank of that array—the set of dimensions in which it varies—is not known at the time of writing of the program, then the programmer is forced to think in an *intensional* manner.

The word *intension* comes from logic: the *intension* of an utterance is a function from the set of *possible worlds*, in which the utterance may take place, to its meaning in each world, while the *extension* of an utterance is its meaning in a specific world. (See [6, 15] for writings on Montague's intensional logic.)

The TL language is explicitly designed around this intuition. For example, the phrase "*Five degrees less than yesterday's temperature*" could be written in TL as

$$(temperature - 5)\ @\ [date \leftarrow \#.date - 1],$$

where variable *temperature* is the temperature, and dimension *date* keeps track of the current date. Note that this expression does not make explicit where *temperature* is to be evaluated, which might vary with respect to many other dimensions, such as *latitude*, *longitude*, *altitude*, *timeOfDay*, and so forth. This idea was first proposed in the paper "Intensional Programming" [7], and again in the collective work *Multidimensional Programming* [2] (p.26).

In TL, a variable A is understood to vary in a fixed set of dimensions (its rank), but conceptually, it can be considered to vary in *all* possible dimensions; this means, say, that if A is defined using two dimensions d_1 and d_2, and B is defined using two dimensions d_2 and d_3, then both can be considered to vary in all three dimensions: the "variance" of A in d_3 is constant, as is the "variance" of B in d_1. As for their sum, $A + B$, it varies in all three dimensions. This approach is consistent with the use of dimensions in differential equations, in which one only writes down the dimensions of relevance, and with the use of attributes in the universal relation model, which is used to define the semantics of relational databases [10].

The origins of the work presented in this paper go back to the mid-1970s, to the Lucid programming language [1]. (See [11] for a detailed history of Lucid and its successors.) In Lucid, a variable is typically defined to be an infinite stream, typically by giving its first element and then the rules for creating subsequent elements from previous elements. For example,

$$A = 0\ fby\ (A + 1)$$

defines the sequence $\langle A_0, A_1, A_2, \ldots \rangle$ given by

$$\begin{aligned} A_0 &= 0 \\ A_{i+1} &= A_i + 1 \end{aligned}$$

i.e., $A = \langle 0, 1, 2, \ldots \rangle$.

A Lucid stream is not a physical data structure, but a conceptual one, so one can truly talk about an infinite stream, as it will never be built in a computer. As for elements of a Lucid stream, these can be accessed randomly. For example, if element 53 of a stream is needed without needing the computation of elements 0 through 52, then only element 53 need be computed.

In Lucid, it is also possible to define streams whose different slots are not "filled-in" in order, or some of which may never get filled in. This is because the semantics of Lucid is based on the Scott order for streams, in which a stream A is less defined than a stream B if A has fewer slots defined than B does, but where they both have slots filled in, the values are the same [18]. This is very different from languages such as LUSTRE [4], Lucide Synchrone [5], or the higher-order dataflow of [16, 17]; for all these formalisms, the semantics uses the prefix order on streams.

Attempts to generalize Lucid streams, which vary in one dimension, to multiple dimensions, led to Indexical Lucid, created by Faustini and Jagannathan [2]. This language introduced the dimensionally-abstract `where` clause and the dimensionally-abstract function, using syntax similar to, but not identical to, that of the examples given in §4. However, dimensions were not first-class values, functions could only be first-order, and there were no partially applied functions.

The first version of TransLucid, with first-class dimensions, was presented in [11], and memoization techniques for its implementation were presented in [12]. However, up to now, TransLucid has had no user-defined functions.

The TL language—TransLucid with functions—presented here solves all of these problems. Its syntax and denotational semantics (§3) have been adapted so that all atomic values can be used as dimensions, and functions can be higher-order and curried with partial application. In addition to a basic algebra of operations provided by a host language, the only domains of TL are for functions, intensions and *contexts*, which correspond to the possible worlds of logic and are used to index intensions. The solution uses completely standard notions from programming-language theory.

The dimensionally-abstract functions of Indexical Lucid do not correspond to primitives in TransLucid. Rather, these functions are defined in TransLucid using syntactic sugar, with the introduction of derived abstractions (§4), for creating functions whose bodies can be evaluated partially with respect to the context in which they are created and partially with respect to the context in which they will be called. With these new syntactic constructs, we outline the principles of a new programming methodology (§5) for declarative, higher-order, multidimensional programming.

The concluding section (§6) outlines the existing implementation of TransLucid, makes explicit the open problems solved by the current paper, and discusses further outstanding issues which need to be resolved before TL can become a widely used language.

2 Intensions, Contexts and Functions

In TL, a variable, and in fact every expression, defines a *multidimensional intension*, which is an array that may be indexed by as many dimensions as one wishes. In the discussion below, we make use of two dimensions, x and y, the ordinates of which are considered to be natural numbers. The word *ordinate* comes from the word *co-ordinates*, literally meaning "ordinates which are together". Below, in addition to two-dimensional intensions, we have one-dimensional and zero-dimensional intensions. These are visualized as tables, with the x dimension being displayed to the right, and the y dimension being displayed down the page. A zero-dimensional intension is simply a single value, such as the intension defined by the expression '42'.

In TL, we can write a definition for variable A using a declaration like the following one:

$$\texttt{var}\ A = 42 + (2 * \#.x) + \#.y$$

The above expression defining A is built up from subexpressions '42', '2', '+', '$*$', '$\#.x$', and '$\#.y$'. If we consider the evaluation of these subexpressions in the aforementioned $\{x, y\}$ two-dimensional

space, then these subexpressions give:

‘42’	
	42

‘2’	
	2

‘#.x’	0	1	2	3	$\overset{\#.x}{\rightarrow}$
	0	1	2	3	$\cdots$

‘#.y’	
0	0
1	1
2	2
3	3
$\#.y\downarrow$	$\vdots$

‘ + ’	
	+

‘ $*$ ’	
	$\times$

Subexpressions ‘42’, ‘2’, ‘ + ’ and ‘ $*$ ’ all define zero-dimensional entities; we say that the *rank* of each is $\emptyset$ (the empty set). It is important to remember that each expression defines a whole array, all at once. So the expression ‘42’ defines an array whose only entry is the value 42. One should not think of this as a two-dimensional one-by-one array, or even a one-dimensional array with one entry, because that is not what is going on here. The array truly is zero-dimensional, and holds one value, it does not have a number of cells holding different values, or even a number of cells all holding 42. Hence, the only value that can be retrieved from the array is the one value that defines it. We cannot emphasize this point enough, because it is critical to understanding the remainder of the text. Without understanding that every expression defines an array, any further attempt at understanding will be fraught with difficulty.

Subexpressions ‘#.x’ and ‘#.y’ are 1-dimensional arrays: ‘#.x’ has rank $\{x\}$, which means that it is an array that has entries in the x direction. In fact, it is an array whose entries are simply the index of the entry, in the x direction. Again, this point is key to understanding TL: when specifying a cell in an intension, one must give, for each dimension in the rank of the intension, both the relevant dimension (the direction) and its ordinate. Similarly, ‘#.y’ has rank $\{y\}$ and is an array whose entries are the index of the entry in the y direction.

For subexpression ‘$2 * \#.x$’, since subexpressions ‘2’ and ‘ $*$ ’ are of rank $\emptyset$, they are naturally extended to rank $\{x\}$, and the resulting array is the multiplication of each pair of corresponding entries from the arrays ‘2’ and ‘#.x’.

‘2’	0	1	2	$\overset{\#.x}{\rightarrow}$
	2	2	2	$\cdots$

‘ $*$ ’	0	1	2	$\overset{\#.x}{\rightarrow}$
	$\times$	$\times$	$\times$	$\cdots$

‘$2 * \#.x$’	0	1	2	$\overset{\#.x}{\rightarrow}$
	0	2	4	$\cdots$

For expression ‘$42 + (2 * \#.x) + \#.y$’, the subexpressions ‘42’ and ‘ + ’ (both rank $\emptyset$), ‘$2 * \#.x$’ (rank $\{x\}$), and ‘#.y’ (rank $\{y\}$) are all extended to rank $\{x, y\}$, and so the value of ‘A’ is:

‘A’	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	42	44	46	48	$\cdots$
1	43	45	47	49	$\cdots$
2	44	46	48	50	$\cdots$
3	45	47	49	51	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Looking at example ‘A’, one could easily get the impression that ordinates must always be natural numbers. This is not the case. Here we show an intension ‘L’, without showing how it might be defined, giving the textual representation of the integers in several languages, varying in dimensions x and `lang`:

‘L’	$\overset{\#.x}{\leftarrow}$	-2	-1	0	1	2	$\overset{\#.x}{\rightarrow}$
`EN`	$\cdots$	minus two	minus two	zero	one	two	$\cdots$
`ES`	$\cdots$	menos dos	menos uno	cero	uno	dos	$\cdots$
`FR`	$\cdots$	moins deux	moins un	zéro	un	deux	$\cdots$
$\#.\texttt{lang}\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Viewing an intension as a multidimensional table does provide us with an intuition of what an intension is. Nevertheless, in general, this table is infinite, and cannot be constructed explicitly in a computer. Furthermore, programmers do not normally see their variables as infinite tables: [7] state with respect to the temperature example of the Introduction, "No one in their right mind would think of `temperature` as denoting some vast infinite table; nor would they consider statements about the temperature to be assertions about infinite tables."

In fact, the infinite table is the extensional view of an intension. The intensional view is from the perspective of a particular point, called a *context*, within the table. For example, we might want to query for the value of A at context $\{x \mapsto 3, y \mapsto 2\}$, which is 50.

In this intensional manner, an expression can be thought of as a mapping from the set of possible worlds (all the contexts for which the intension is defined) to its meaning in each specific world (each particular context). Then, when a specific entry is required, one need only compute the entries necessary.

This is, in fact, how the semantics in §3 is defined. To compute the value of an expression in an intensional manner, one must first define at which array entry, or at which context, one would like to reach into the intension. That context is called the "current" context, which corresponds to the # symbol appearing in our examples. Therefore, in evaluating A in the example above in the context $\{x \mapsto 3, y \mapsto 2\}$, the expressions $\#.x$ and $\#.y$ have the values 3 and 2 respectively.

Because the rank of A is $\{x, y\}$, A is defined whenever the current context defines *at least* x and y. So, for example, if the current context is $\{x \mapsto 2, y \mapsto 1, z \mapsto 12, w \mapsto 10\}$, then the value of A is the same as if the current context were $\{x \mapsto 2, y \mapsto 1\}$, i.e., the value is 47.

If we view an intension from the extensional point of view, i.e., as a giant, infinite, multidimensional table, then the context is the set of Cartesian coordinates that allows us to pick out a specific value in the table.

However, if the intensional point of view is taken, the current context can be considered to be an implicit parameter of an expression that can be manipulated explicitly as needed, using the context constructor $[\cdots]$ and the context change operator ' `@` '.

The context is changed by specifying the relevant dimensions in a context constructor, and the new ordinates for each of these. For each dimension, the change can be either relative to the current context, or an absolute change, as seen in the following examples:

$$\texttt{var}\ B = A\ \texttt{@}\ [x \leftarrow \#.x + 1, y \leftarrow \#.y + 2] \qquad \texttt{var}\ B' = A\ \texttt{@}\ [x \leftarrow \#.x + 1, y \leftarrow 3]$$

The variable B defines an intension that has the same values as A, but shifted one 'to the left' and two 'up'; the context change is relative for both x and y. As for B', one absolute row in the y direction is chosen, making B' a one-dimensional intension.

'B'	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	46	48	50	52	$\cdots$
1	47	49	51	53	$\cdots$
2	48	50	52	54	$\cdots$
3	49	51	53	55	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

'B''	0	1	2	3	$\overset{\#.x}{\rightarrow}$
	47	49	51	53	$\cdots$

In both cases, the context constructor to the right of the ' `@` ' produces a new context in each context:

'$[x \leftarrow \#.x + 1, y \leftarrow \#.y + 2]$'

	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	$\{x \mapsto 1, y \mapsto 2\}$	$\{x \mapsto 2, y \mapsto 2\}$	$\{x \mapsto 3, y \mapsto 2\}$	$\{x \mapsto 4, y \mapsto 2\}$	$\cdots$
1	$\{x \mapsto 1, y \mapsto 3\}$	$\{x \mapsto 2, y \mapsto 3\}$	$\{x \mapsto 3, y \mapsto 3\}$	$\{x \mapsto 4, y \mapsto 3\}$	$\cdots$
2	$\{x \mapsto 1, y \mapsto 4\}$	$\{x \mapsto 2, y \mapsto 4\}$	$\{x \mapsto 3, y \mapsto 4\}$	$\{x \mapsto 4, y \mapsto 4\}$	$\cdots$
3	$\{x \mapsto 1, y \mapsto 5\}$	$\{x \mapsto 2, y \mapsto 5\}$	$\{x \mapsto 3, y \mapsto 5\}$	$\{x \mapsto 4, y \mapsto 5\}$	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

‘$[x \leftarrow \#.x + 1, y \leftarrow 3]$’

0	1	2	3	$\overset{\#.x}{\rightarrow}$
$\{x \mapsto 1, y \mapsto 3\}$	$\{x \mapsto 2, y \mapsto 3\}$	$\{x \mapsto 3, y \mapsto 3\}$	$\{x \mapsto 4, y \mapsto 3\}$	$\cdots$

For example, suppose the current context were $\{x \mapsto 2, y \mapsto 1, z \mapsto 12, w \mapsto 10\}$. Then the expression ‘$[x \leftarrow \#.x + 1, y \leftarrow \#.y + 2]$’ would evaluate to the context $\{x \mapsto 3, y \mapsto 3\}$, and so the new context resulting from the application of the ‘ `@` ’ would be $\{x \mapsto 3, y \mapsto 3, z \mapsto 12, w \mapsto 10\}$, i.e., the ordinates of z and w would not be affected, and so the result would be 51. As for the expression ‘$[x \leftarrow \#.x + 1, y \leftarrow 3]$’, it would also evaluate to the context $\{x \mapsto 3, y \mapsto 3\}$, so the result would also be 51.

So, putting the two perspectives together, programming in TL can be called *Cartesian intensional programming*.

When a function appears in TransLucid, it can be considered to be an *encapsulated* intension with arguments. For example, here we define a function with two arguments:

‘ `var` $C = \lambda a \to \lambda b \to a + b + 2$’

‘C’

a, b

	0	1	2	3	$\overset{a}{\rightarrow}$
0	2	3	4	5	$\cdots$
1	3	4	5	6	$\cdots$
2	4	5	6	7	$\cdots$
3	5	6	7	8	$\cdots$
$b\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

‘$C.(\#.x).(\#.y)$’

	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	2	3	4	5	$\cdots$
1	3	4	5	6	$\cdots$
2	4	5	6	7	$\cdots$
3	5	6	7	8	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

In the table to the left, the box is an atomic value, meaning that the value of ‘C’ is an intension of rank $\emptyset$. Note that the box has a little box attached, holding the arguments a and b. This is how a function with two arguments is visualized. In the table to the right, ‘$C.(\#.x).(\#.y)$’ is the application of ‘C’ to arguments ‘$\#.x$’ and ‘$\#.y$’.

Local identifiers can be introduced both for variables, with primitive `wherevar`, and dimensions, with primitive `wheredim`. Here is the definition for iterative factorial, using a local dimension identifier d. When this expression is evaluated, a new dimension is allocated for d, and then used.

```
var fact = λn → F
wherevar
  F = if #.d ≡ 0 then 1 else #.d × (F @ [d ← #.d − 1]) fi
      wheredim
        d ← n
      end
end
```

It is, in general, possible for there to be several active instantiations of the same `wheredim` clause. Therefore, the dimension allocation must ensure that a different dimension be allocated for each of these instantiations. This is done by using a series of χ^i_ν dimensions, which are indexed by the current path—encoded as a list ν—in the currently-being-evaluated expression tree (held in the current context by dimension ρ), and by the position i in the `wheredim` clause (for the current example, $i = 1$).

‘F’	0	1	2	3	4	$\overset{\#.\chi^1_{\#.\rho}}{\rightarrow}$
	1	1	2	6	24	$\cdots$

The current context is initialized with $\#.\chi^1_{\#.\rho}$ set to n.

The above definition could be rewritten, using syntactic sugar, as:

```
fun fact.n = F
where
  dim d ← n
  var F = if #.d ≡ 0 then 1 else #.d × (F @ [d ← #.d − 1]) fi
end
```

Note the introduction of the `fun` *fact.n* $= \cdots$ notation, as well as that of the `where` clause, combining the `wherevar` and `wheredim` clauses into one.

All of the primitives of TL have now been presented informally; their formalization follows in the next section. In §4, we will present the rest of the language, using new syntactic constructs for functions whose bodies are context-sensitive.

3 Semantics

The denotational semantics computes least fixed points of systems of equations in a semantic domain where variables denote *intensions*. The semantic rules are of the form

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

where E is an expression, ι is an interpretation of the constant symbols, ζ is an environment mapping variables to intensions, and κ is the current context. The section will define basic notation, the domains and the formal syntax, then give the rules.

3.1 Notation for function manipulation

- Let A and B be two sets. A *partial function* f from A to B is written $f : A \rightarrowtail B$.
- Let f be a function with finite domain $\{v_1, \ldots, v_m\}$. Then f can be given as its *graph* $\{v_1 \mapsto f(v_1), \ldots, v_m \mapsto f(v_m)\}$. When the graph is empty, it is written $\emptyset$.
- Let $f, g : A \rightarrowtail B$. The *perturbation of f by g* is defined by:

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

- Let $f : A \to B$. The *substitution of f's value for v by v'* is defined by:

$$f[v/v'] \quad = \quad f \dagger \{v \mapsto v'\}.$$

- Let $f : A \rightarrowtail B$, and let $S \subseteq A$. The *domain restriction of f to S* is defined by

$$(f \lhd S)(v) \quad = \quad \{ \ f(v), \quad v \in S.$$

3.2 Domains

Definition 1 *Let D be an enumerable set of values. The* semantic domain $\mathbf{D}$ derived from D *is the least solution to the equations*

$$\begin{aligned} \mathbf{D} &= D \cup \mathbf{D}_{\texttt{atomic},m} \cup \mathbf{D}_{\texttt{ctxt}} \cup \mathbf{D}_{\texttt{intens}} \cup \mathbf{D}_{\texttt{func}} \\ \mathbf{D}_{\texttt{atomic},m} &= D^m \rightarrowtail D, \quad m > 0 \\ \mathbf{D}_{\texttt{ctxt}} &= D \rightarrowtail \mathbf{D} \\ \mathbf{D}_{\texttt{intens}} &= \mathbf{D}_{\texttt{ctxt}} \rightarrowtail \mathbf{D} \\ \mathbf{D}_{\texttt{func}} &= \mathbf{D} \rightarrowtail \mathbf{D} \end{aligned}$$

where for all $\eta \in \mathbf{D}_{\texttt{intens}}$*, if* $\kappa \in \text{dom } \eta$*, then for all* κ' *such that* $\kappa = \kappa' \lhd (\text{dom } \kappa)$*, we have* $\eta(\kappa) = \eta(\kappa')$*. We call this the* intension restriction.

We call

- D *the set of* atomic values*; an atomic value is written* d*; a subset of* D *is written* Δ*;*
- $\mathbf{D}_{\texttt{atomic},m}$ *the set of* atomic functions *of arity* m*, such as arithmetic and Boolean operators, which are provided by a host environment; an atomic function is written* op*;*
- $\mathbf{D}_{\texttt{ctxt}}$ *the set of* contexts*; a context is written* κ*; elements of the domain of a context are called* dimensions*; elements of the codomain of a context are called* ordinates*;*
- $\mathbf{D}_{\texttt{intens}}$ *the set of* intensions*, mapping contexts to values; an intension is written* η*;*
- $\mathbf{D}_{\texttt{func}}$ *the set of* functions*; a function is written* f*.*

Note that $\mathbf{D}_{\texttt{atomic},1}$, $\mathbf{D}_{\texttt{ctxt}}$ and $\mathbf{D}_{\texttt{intens}}$ are all subsets of $\mathbf{D}_{\texttt{func}}$. Because of this situation, we will only need to define one kind of application in the abstract syntax.

The intension restriction, that for all κ' such that $\kappa = \kappa' \lhd (\text{dom } \kappa)$, we have $\eta(\kappa) = \eta(\kappa')$, ensures that a TL expression gives a certain result in context κ, that adding to the context will not change the value of the expression. This is a finitary requirement, essential given that we are working with infinite data structures. This precludes any sort of belief revision or non-monotonic reasoning.

Definition 2 *Let* D *be an enumerable set of values,* $\mathbf{D}$ *be the semantic domain derived from* D*, and* $\bot \notin \mathbf{D}$*. Then we define the order* $\sqsubseteq$ *over* $\mathbf{D}_\bot = \mathbf{D} \cup \{\bot\}$ *by:*

- *For all* $d \in \mathbf{D}_\bot$*,* $\bot \sqsubseteq d$*.*
- *For all* $d \in D$*,* $d \sqsubseteq d$*.*
- *For all* $op, op' \in \mathbf{D}_{\texttt{atomic},m}$*,* $op \sqsubseteq op'$ *iff* $op = op' \lhd (\text{dom } op)$*.*
- *For all* $\kappa, \kappa' \in \mathbf{D}_{\texttt{ctxt}}$*,* $\kappa \sqsubseteq \kappa'$ *iff* $\kappa = \kappa' \lhd (\text{dom } \kappa)$*.*
- *For all* $\eta, \eta' \in \mathbf{D}_{\texttt{intens}}$*,* $\eta \sqsubseteq \eta'$ *iff* $\eta = \eta' \lhd (\text{dom } \eta)$*.*
- *For all* $f, f' \in \mathbf{D}_{\texttt{func}}$*,* $f \sqsubseteq f'$ *iff* $f = f' \lhd (\text{dom } f)$*.*

Proposition 1 *The pair* $(\mathbf{D}_\bot, \sqsubseteq)$ *is a complete partial order, such that the following are also cpos:*

1. $(D_\bot, \sqsubseteq)$*, where* $D_\bot = D \cup \{\bot\}$*;*
2. $(\mathbf{D}_{\texttt{atomic},m}, \sqsubseteq)$*;*
3. $(\mathbf{D}_{\texttt{ctxt}}, \sqsubseteq)$*;*
4. $(\mathbf{D}_{\texttt{intens}}, \sqsubseteq)$*;*
5. $(\mathbf{D}_{\texttt{func}}, \sqsubseteq)$*.*

Proof. Mostly standard. See Appendix A.

Definition 3 *The* rank *of an intension* η*, written* $\text{rank}(\eta)$*, is the minimal set of dimensions needed to fully define* η*. It is given by:*

$$\text{rank}(\eta) = \bigcup \{\text{dom}(\kappa) \mid \kappa \in \text{dom}(\eta) \text{ and } \nexists \kappa' \sqsubset \kappa,\ \eta(\kappa') = \eta(\kappa)\}$$

$$
\begin{array}{rll}
E ::= & x & \textit{identifier} \\
| & {}^{m}c & \textit{constant symbol} \\
| & \# & \textit{current context} \\
| & [E \leftarrow E,\ \ldots] & \textit{context constructor} \\
| & \lambda x \to E & \textit{function abstraction} \\
| & E\ .\ (E, \ldots, E) & \textit{function application} \\
| & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E\ \texttt{fi} & \textit{conditional} \\
| & E\ \texttt{@}\ E & \textit{context perturbation} \\
| & E\ \texttt{wheredim}\ x \leftarrow E,\ \ldots\ \texttt{end} & \textit{local dimensions} \\
| & E\ \texttt{wherevar}\ x = E,\ \ldots\ \texttt{end} & \textit{local variables}
\end{array}
$$

Figure 1: Syntax of TL expressions

Definition 4 *Let D be an enumerable set of values and X be a set of variables. Then* $\mathbf{Env}(X, D)$ *is the set of* environments over X and D, *i.e., mappings* $\zeta : X \rightarrowtail \mathbf{D}_{\texttt{intens}}$. *We extend* $\sqsubseteq$ *to environments and define the extended rank (*erank*) and extended range (*eran*):*

$$
\begin{array}{rcl}
\zeta \sqsubseteq \zeta' & \textit{iff} & \mathrm{dom}(\zeta) \subseteq \mathrm{dom}(\zeta') \textit{ and } \forall x \in \mathrm{dom}\ \zeta,\ \zeta(x) \sqsubseteq \zeta'(x) \\
\mathrm{erank}(\zeta) & = & \bigcup \{\mathrm{rank}(\zeta(x)) \mid x \in \mathrm{dom}(\zeta)\} \\
\mathrm{eran}(\zeta) & = & \bigcup \{\mathrm{ran}(\zeta(x)) \mid x \in \mathrm{dom}(\zeta)\}.
\end{array}
$$

3.3 Syntax

Definition 5 *A* signature $\Sigma = (C, ar)$ *is a pair consisting of a set C of* constant symbols *and an* arity *function $ar : C \to \mathbb{N}$. We write ${}^{m}c$ for a constant symbol in C of arity m.*

Definition 6 *Let Σ be a signature and let D be a set of atomic values. An* interpretation *of Σ over D is a function $\iota : C \to D \cup \bigcup_{m>0}(D^m \rightarrowtail D)$ such that $\iota({}^{0}c) \in D$ and $\iota({}^{m}c) : D^m \rightarrowtail D$, $m > 0$. We write* $\mathbf{Interp}(\Sigma, D)$ *for the set of interpretations of Σ over D.*

The pair (Σ, ι) together form an algebra.

Definition 7 *Let Σ be a signature and X ($\ni x$) be a set of identifiers. Then* $\mathbf{Expr}(\Sigma, X)$ *($\ni E$) is the set of* TL expressions *over Σ and X. The* free variables *of a TL expression E are written $FV(E)$. The* abstract syntax *for TL expressions is given in Figure 1.*

The function application can have multiple arguments because the function may be a host function of arity $m > 1$. User-defined TL functions are all curried.

3.4 Semantic rules

Before giving the semantic rules, we need to define notation appearing therein:

- We write ν for a list of natural numbers, where $\nu \in \mathbb{N}^*$. The empty list is written ϵ, and the appending of element n to list ν is written $\nu : n$.
- Let $n \in \mathbb{N}$ and κ be a context. Then

$$
{}_{n}\kappa \quad = \quad \kappa\big[\rho/(\kappa(\rho) : n)\big]
$$

- Let d be a value. Then $\widehat{d}$ is a constant intension, defined by $\widehat{d} = \lambda\kappa \to d$.

$$
\begin{aligned}
[\![x]\!]\iota\zeta\kappa &= \zeta(x)(\kappa) && (1)\\
[\![{}^m c]\!]\iota\zeta\kappa &= \iota({}^m c) && (2)\\
[\![\#]\!]\iota\zeta\kappa &= \kappa && (3)\\
[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]\iota\zeta\kappa &= \big\{[\![E_{i0}]\!]\iota\zeta({}_i\kappa) \;\mapsto\; [\![E_{i1}]\!]\iota\zeta({}_{(i+m)}\kappa)\big\} && (4)\\
[\![\lambda x \to E_0]\!]\iota\zeta\kappa &= \lambda d_a.[\![E_0]\!]\iota\big(\zeta[x/\widehat{d_a}]\big)\{\rho \mapsto {}_0\kappa(\rho)\} && (5)\\
[\![E_0.(E_i)_{i=1..m}]\!]\iota\zeta\kappa &= \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_i]\!]\iota\zeta({}_i\kappa)\big) && (6)\\
[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\zeta\kappa &= \text{let } d_0 = [\![E_0]\!]\iota\zeta({}_0\kappa) && (7)\\
&\quad \text{in } \begin{cases} [\![E_1]\!]\iota\zeta({}_1\kappa), & d_0 \equiv \mathit{true}\\ [\![E_2]\!]\iota\zeta({}_2\kappa), & d_0 \equiv \mathit{false} \end{cases}\\
[\![E_0 \mathbin{@} E_1]\!]\iota\zeta\kappa &= [\![E_0]\!]\iota\zeta\big({}_0\kappa \dagger [\![E_1]\!]\zeta({}_1\kappa)\big) && (8)\\
[\![E \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa &= \text{let } d_i = \chi^i_{\kappa(\rho)} && (9)\\
&\quad\; d'_i = [\![E_i]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } [\![E_0]\!]\iota\big(\zeta[x_i/\widehat{d_i}]\big)\big({}_0\kappa[d_i/d'_i]\big)\\
[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa &= \text{let } \zeta_0 = \zeta[x_i/\emptyset] && (10)\\
&\quad\; \zeta_{\alpha+1} = \zeta_\alpha\big[x_i/[\![E_i]\!]\iota\zeta_\alpha\big]\\
&\quad\; \zeta_\sqcup = \text{lfp } \zeta_\alpha\\
&\quad \text{in } [\![E_0]\!]\iota\zeta_\sqcup\kappa
\end{aligned}
$$

Figure 2: Semantics of TL expressions

Definition 8 *Let X be a set of variables; $D = \Delta_S \cup \Delta_H$, where Δ_S is a set of atomic values, and Δ_H is a second set of atomic values, called* hidden dimensions*; Σ be a signature; $E \in \mathbf{Expr}(\Sigma, X)$; $\iota \in \mathbf{Interp}(\Sigma, \Delta_S)$; $\zeta \in \mathbf{Env}(X, D)$; and κ be a context such that*

$$
\begin{aligned}
\Delta_S \cap \Delta_H &= \emptyset\\
\{\mathit{true}, \mathit{false}\} &\subseteq \Delta_S\\
\Delta_H &= \{\chi^i_\nu \mid \nu \in \mathbb{N}^*, i \in \mathbb{N}\} \cup \{\rho\}\\
\text{erank}(\zeta) \cup \text{eran}(\zeta) &\subseteq \Delta_S\\
\text{dom}(\kappa) \cup \text{ran}(\kappa) &\subseteq \Delta_S
\end{aligned}
$$

Then the semantics for E with respect to ι, ζ and κ is given by

$$[\![E]\!]\iota\zeta\big(\kappa[\rho/\epsilon]\big),$$

where the rules for $[\![\cdot]\!]$ are given in Figure 2.

We explain all of the different cases below. Note that for each subexpression, the context is perturbed by changing the ordinate for dimension ρ to keep track of the current path from the root of the expression tree to the current node.

1. A variable identifier x is looked up in environment ζ, and the resulting intension is applied to κ to produce a value.

2. An m-ary constant symbol ${}^m c$ is looked up in interpretation ι, returning an atomic value if $m = 0$, otherwise an m-ary atomic function if $m > 0$.

3. The current context is returned when # appears.

4. The context constructor creates a function whose domain is the set of the results of the left-hand sides and whose range is the set of the results of the right-hand sides.

5. The λ creates a function whose body is only sensitive to the ρ-ordinate.

6. In function application, the function and the arguments are all built in context κ, then the function is applied to the arguments, also in κ.

7. Condition E_1 is evaluated in context κ, then, depending on the returned value, one of the choices E_2 or E_3 is evaluated, also in κ.

8. Expression E_1 is evaluated to a context, used to perturb the current context κ to produce a new running context for the evaluation of E_2.

9. Each dimension identifier x_i is mapped in the new environment $\zeta_\sqcup$ to $\widehat{d_i}$, where d_i is a χ dimension whose ordinate is initially the value of expression E_i in context κ.

10. A sequence of environments ζ_α, $\alpha \in \mathbb{N}$, is defined by creating the initial environment $\zeta_0 = \zeta[x_i/\emptyset]_{i=1..m}$, then applying the meaning of the individual equations, mapping variable identifier x_i to the meaning of defining expression E_i to produce $\zeta_{\alpha+1}$ from ζ_α. The expression E is then evaluated in the least-fixed-point environment $\zeta_\sqcup$ resulting from the sequence of the ζ_α.

4 Syntactic Extensions

The syntactic extensions introduced in this section all allow the creation of abstractions whose body, once evaluated, is sensitive to the context at the time of application, and may also be sensitive to a named set of dimensions of the context at the time of abstraction.

4.1 Intension Abstraction

The $\uparrow$ operator allows an entire intension to be wrapped up into a single value, and the $\downarrow$ operator decapsulates an encapsulated intension, as can be seen with the examples of A and B. The rank of A is $\emptyset$: it is a zero-dimensional array whose value is the encapsulated intension, while $\downarrow A$ has rank $\{x, y\}$.

$$\texttt{var } A =\uparrow (\#.x + \#.y + 2)$$

‘A’

	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	2	3	4	5	$\cdots$
1	3	4	5	6	$\cdots$
2	4	5	6	7	$\cdots$
3	5	6	7	8	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

‘$\downarrow A$’

	0	1	2	3	$\overset{\#.x}{\rightarrow}$
0	2	3	4	5	$\cdots$
1	3	4	5	6	$\cdots$
2	4	5	6	7	$\cdots$
3	5	6	7	8	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

As seen above, an encapsulated intension, when decapsulated, gives an array that is identical to the one that was encapsulated. However, there are situations in which one does not wish to encapsulate the entire array, but instead to filter out a particular row in a certain direction. This is done by explicitly stating which dimensions are to be filtered. For example, here B filters out the y direction. As a result, the rank of B is $\{y\}$, and there is a different intension for each

y-ordinate. If we wish a particular one of these intensions, we can specify which one with @, as seen here with C, and its decapsulation $\downarrow C$.

$\texttt{var}\ B = \uparrow \{y\}\ (\#.x + \#.y)$ $\qquad$ $\texttt{var}\ C = B\ @\ [y \leftarrow 1]$

‘B’

					$\overset{\#.x}{\rightarrow}$
0	2	3	4	5	$\cdots$
1	3	4	5	6	$\cdots$
$\#.y\downarrow$	$\vdots$				

‘C’

					$\overset{\#.x}{\rightarrow}$
	3	4	5	6	$\cdots$

‘$\downarrow C$’

					$\overset{\#.x}{\rightarrow}$
	3	4	5	6	$\cdots$

As described in §3, an intension is a mapping from contexts to atomic values. So to encode that in primitive TransLucid, an intension as first-class object is a function that takes a context—which we will always refer to as an *encapsulated intension*, in order to avoid confusion with the intensions that an expression defines—whose body may or may not use that context. Without considering the freezing of dimensions, the expression ‘$\uparrow E$’ is simply syntactic sugar for ‘$\lambda\kappa \rightarrow E\ @\ \kappa$’; while ‘$\downarrow E$’ is simply ‘$E.\#$’, the application to the current context of the function to which E evaluates.

The formal translation from these syntactic constructs into the primitive ones from §3 is as follows. In order to avoid too many parentheses, we write $E^{\mathcal{T}}$ instead of $\mathcal{T}(E)$.

$$\begin{aligned}
\mathcal{T}(\downarrow E) &\Rightarrow E^{\mathcal{T}}.\# \\
\mathcal{T}(\uparrow \{E_1, \ldots, E_m\}\ E_0) &\Rightarrow (\lambda d_1 \rightarrow \cdots \rightarrow \lambda d_m \rightarrow \lambda\kappa \rightarrow \lambda\kappa_a \\
&\quad \rightarrow E_0^{\mathcal{T}}\ @\ [d_i \leftarrow \kappa.d_i]_{i=1..m}\ @\ \kappa_a).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\kappa
\end{aligned}$$

4.2 Freezing the Context for Function Abstractions

The arguments in braces of the intension abstraction operator are used to freeze the ordinates of a specified set of dimensions from the context at abstraction time. For example, the function abstraction A given below creates a different function for each different x-ordinate:

$$\texttt{var}\ A = \lambda\{x\}\ a \rightarrow a + \#.x$$

‘A’

0					1					$\overset{\#.x}{\rightarrow}$
a					a					
	0	1	2	3 $\overset{a}{\rightarrow}$		0	1	2	3 $\overset{a}{\rightarrow}$	
	0	1	2	3 $\cdots$		1	2	3	4 $\cdots$	$\cdots$

If we apply A to a query for the x-ordinate, we get:

‘$A.(\#.x)$’

					$\overset{\#.x}{\rightarrow}$
	0	2	4	6	$\cdots$

This syntactic construct can be easily transformed into the primitive ones from §3.

$$\begin{aligned}
\mathcal{T}(\lambda\{E_1, \ldots, E_m\}\ x \rightarrow E_0) &\Rightarrow (\lambda d_1 \rightarrow \cdots \rightarrow \lambda d_m \rightarrow \lambda\kappa \rightarrow \lambda x \\
&\quad \rightarrow E_0^{\mathcal{T}}\ @\ [d_i \leftarrow \kappa.d_i]_{i=1..m}).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\kappa
\end{aligned}$$

4.3 Call-by-value Context-sensitive Functions

The intension abstraction operator $\uparrow$ allows the construction of expressions which are sensitive to the application context, as well as the abstraction context for named dimensions. We do the same for functions, with the λ^{v} operator. Here, A is defined with respect to the x-ordinate of the *abstraction* context and the y-ordinate of the *application* context.

$$\texttt{var}\ A = \lambda^{\mathrm{v}}\{x\}\ a \to a + \#.x + \#.y$$

‘A’ | 0 | 1 | $\xrightarrow{\#.x}$

a

	0	1	2	3	$\xrightarrow{a}$
0	1	2	3	4	$\cdots$
1	2	3	4	5	$\cdots$
2	3	4	5	6	$\cdots$
3	4	5	6	7	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

a

	0	1	2	3	$\xrightarrow{a}$
1	2	3	4	5	$\cdots$
2	3	4	5	6	$\cdots$
3	4	5	6	7	$\cdots$
4	5	6	7	8	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

$\ldots$

Application of these call-by-value context-sensitive functions is done with the ‘!’ operator. If we apply A to a query for the x-ordinate, we get:

‘$A!(\#.x)$’	0	1	2	3	$\xrightarrow{\#.x}$
0	0	2	4	6	$\cdots$
1	1	3	5	7	$\cdots$
2	2	4	6	8	$\cdots$
3	3	5	7	9	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

As before, these syntactic constructs can be transformed into the primitive ones from §3.

$$\begin{aligned}
\mathcal{T}\big(E_0 ! E_1\big) &\Rightarrow E_0^{\mathcal{T}}.E_1^{\mathcal{T}}.\# \\
\mathcal{T}\big(\lambda^{\mathrm{v}}\{E_1,\ldots,E_m\}\ x \to E_0\big) &\Rightarrow \big(\lambda d_1 \to \cdots \to \lambda d_m \to \lambda\kappa \to \lambda x \to \lambda\kappa_a \\
&\quad \to E_0^{\mathcal{T}}\ @\ [d_i \leftarrow \kappa.d_i]_{i=1..m}\ @\ \kappa_a\big).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\kappa
\end{aligned}$$

4.4 Call-by-name Context-sensitive Functions

Up to now, all of the arguments to functions are fully evaluated before being passed to the functions. However, there are many times in which one wishes to pass an entire, encapsulated, and therefore unevaluated, intension to a function. This is done with the λ^{n} operator. Here, function A takes a dimension d and an intension X as input, and shifts X one “to the left”. We write $X_{\{d\mapsto i\}}$ for the value of X when the current d-ordinate is i.

$$\texttt{var}\ A = \lambda\, d \to \lambda^{\mathrm{n}}\, X \to X\ @\ [d \leftarrow \#.d + 1]$$

‘A’

d, X

	0	1	2	3	$\xrightarrow{\#.d}$
	$X_{\{d\mapsto 1\}}$	$X_{\{d\mapsto 2\}}$	$X_{\{d\mapsto 3\}}$	$X_{\{d\mapsto 4\}}$	$\cdots$

Application of these call-by-name context-sensitive functions is done with the space (‘ ’) operator. If we apply A to a query for the x-ordinate plus one, we get:

‘$A.x\ (\#.x + 1)$’	0	1	2	3	$\xrightarrow{\#.x}$
0	2	3	4	5	$\cdots$

The transformation of these syntactic constructs into the primitive ones from §3 is not as straightforward as before. At the time of application, the argument is explicitly encapsulated. When the argument appears in the body, it must be explicitly decapsulated. This is done in the transformation by syntactic substitution ($E_0[x/\mathcal{T}(\downarrow x)]$):

$$\begin{array}{rcl}
\mathcal{T}\big(E_0\ E_1\big) & \Rightarrow & E_0.(\mathcal{T}(\uparrow E_1)).\# \\
\mathcal{T}\big(\lambda^{\mathrm{n}}\{E_1,\ldots,E_m\}\ x \to E_0\big) & \Rightarrow & \big(\lambda d_1 \to \cdots \to \lambda d_m \to \lambda\kappa \to \lambda x \to \lambda\kappa_a \\
 & & \to E_0^{\mathcal{T}}[x/\mathcal{T}(\downarrow x)]\ \texttt{@}\ [d_i \leftarrow \kappa.d_i]_{i=1..m}\ \texttt{@}\ \kappa_a\big). \\
 & & E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\kappa
\end{array}$$

The definition of A could be rewritten, using syntactic sugar, as the function *next*:

$$\texttt{fun}\ \mathit{next}.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d + 1]$$

where '.', '!', and ' ' introduce, respectively, in the `fun` declarations, call-by-value parameters for ordinary functions, call-by-value parameters for context-sensitive functions, and call-by-name parameters for context-sensitive functions. Then along with the definition of the function *index*:

$$\texttt{fun}\ \mathit{index}!d = \#.d + 1$$

the expression $\mathit{next}.x\ (\mathit{index}!x)$ is equivalent to the above expression $A.x\ (\#.x + 1)$.

5 Programming Methodology

In this section, we examine certain well-known, simple problems, and show how the multidimensional approach of TL can offer insight. To simplify the presentation, we will use the terms *base parameters* for call-by-value parameters for ordinary functions, *value parameters* for call-by-value parameters for context-sensitive functions, and *named parameters* for call-by-name parameters for context-sensitive functions.

5.1 Standard functions

Below are some standard TL functions.

$\texttt{fun}\ \mathit{index}!d = \#.d + 1$
$\texttt{fun}\ \mathit{first}.d\ X = X\ \texttt{@}\ [d \leftarrow 0]$
$\texttt{fun}\ \mathit{next}.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d + 1]$
$\texttt{fun}\ \mathit{fby}.d\ X\ Y = \texttt{if}\ \#.d \equiv 0\ \texttt{then}\ X\ \texttt{else}\ Y\ \texttt{@}\ [d \leftarrow \#.d - 1]\ \texttt{fi}$
$\texttt{fun}\ \mathit{lPair}.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d \times 2]$
$\texttt{fun}\ \mathit{rPair}.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d \times 2 + 1]$

In the case for *index*, d is a value parameter; in all other cases, d, is a base parameter. As for X and Y, they are named parameters.

Function *index* evaluates the body '$\#.d + 1$' in the context in which the function is *applied*, not created.

Functions *first*, *next* and *fby* ("followed-by") existed in the original Lucid, but only for one, implicit dimension. They are analogous to *hd*, *tl* and *cons* for lists, but apply to infinite structures. The function $\mathit{fby}.d\ X\ Y$ means the first element of X *followed by* the elements of Y starting from zero in the d direction.

Functions *lPair* and *rPair* split datasets in two, for divide-and-conquer algorithms.

If $A = \langle a_0, a_1, a_2, \ldots \rangle$ and $B = \langle b_0, b_1, b_2, \ldots \rangle$ are two intensions varying in dimension d, then

	0	1	2	3	4	5	$\overset{\#.d}{\rightarrow}$
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_6	$\cdots$
fby.d A B	a_0	b_0	b_1	b_2	b_3	b_4	$\cdots$
lPair.d A	a_0	a_2	a_4	a_6	a_8	a_{10}	$\cdots$
rPair.d A	a_1	a_3	a_5	a_7	a_9	a_{11}	$\cdots$

Another standard function from Lucid is *wvr* ("whenever"), which is a *filter* in the d dimension.

$$\begin{aligned}\texttt{fun}\ \mathit{wvr}.d\ X\ Y = {}& \texttt{if}\ \mathit{first}.d\ Y \\ & \texttt{then}\ \mathit{fby}.d\ X\ \big(\mathit{wvr}.d\ (\mathit{next}.d\ X)\ (\mathit{next}.d\ Y)\big) \\ & \texttt{else}\ \mathit{wvr}.d\ (\mathit{next}.d\ X)\ (\mathit{next}.d\ Y)\ \texttt{fi}\end{aligned}$$

It defines an intension varying in the d dimension that retains elements of the X input *whenever* the corresponding Y element is true. If $B = \langle T, F, T, T, F, T, T, F, T, \ldots \rangle$, then

	0	1	2	3	4	5	$\overset{\#.d}{\rightarrow}$
wvr.d A B	a_0	a_2	a_3	a_5	a_6	a_8	$\cdots$

Last, we present three more standard functions.

$$\begin{aligned}&\texttt{fun}\ \mathit{default}.d.m.n\ X\ Y = \texttt{if}\ \#.d \geq m\ \texttt{\&\&}\ \#.d \leq n\ \texttt{then}\ X\ \texttt{else}\ Y\ \texttt{fi} \\ &\texttt{fun}\ \mathit{rotate}.d_1.d_2\ X = X\ \texttt{@}\ [d_1 \leftarrow \#.d_2] \\ &\texttt{fun}\ \mathit{sum}.d_x.n\ X = Y\ \texttt{@}\ [d_x \leftarrow n] \\ &\texttt{where} \\ &\quad \texttt{var}\ Y = \mathit{fby}.d_x\ 0\ (X + Y) \\ &\texttt{end}\end{aligned}$$

Function *default* creates an intension varying in dimension d using elements from X for entries m to n, and elements from Y elsewhere.

Function *rotate*.d_1.d_2 X changes variance in dimension d_1 of X to dimension d_2.

Function *sum*.d_x.n X adds up the first n elements in direction d_x of the encapsulated intension X. The local variable Y holds the running sums of the elements of X.

5.2 Ackermann

The Ackermann function is one of the first recursive functions discovered that is not primitive recursive. It grows so fast that in practice it cannot be computed once its first argument is greater than 3, except for the values shown. Here it is presented as a variable varying in dimensions d_m and d_n.

A	0	1	2	3	4	5	$\overset{\#.d_n}{\rightarrow}$
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	$\cdots$					
$\#.d_m\downarrow$	$\vdots$	$\ddots$					

In TL, Ackermann takes two base parameters, and is defined using two local dimensions.

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

Note that there is one explicit manipulation of dimension, all other dimension manipulations use the relative functions *index*, *next* and *fby*. Written explicitly, A would be:

$$
\begin{array}{l}
\texttt{var}\ A = \texttt{if}\ \#.d_m \equiv 0\ \texttt{then}\ \#.d_n + 1 \\
\qquad \texttt{elsif}\ \#.d_n \equiv 0\ \texttt{then}\ A\ \texttt{@}\ [d_m \leftarrow \#.d_m - 1,\ d_n \leftarrow 1] \\
\qquad \texttt{else}\ A\ \texttt{@}\ \big[d_m \leftarrow \#.d_m - 1,\ d_n \leftarrow A\ \texttt{@}\ [d_n \leftarrow \#.d_n - 1]\big]\ \texttt{fi}
\end{array}
$$

5.3 Factorial by divide-and-conquer

The recursive or iterative factorial taught in elementary programming classes does not work well with large numbers. For those situations, divide-and-conquer is better suited.

$F_{n=8}$	0	1	2	3	4	5	6	7	8	9	$\overset{\#.d}{\rightarrow}$
0	1	1	2	3	4	5	6	7	8	1	$\cdots$
1	1	6	20	42	8	1	1	1	1	1	$\cdots$
3	6	840	8	1	1	1	1	1	1	1	$\cdots$
3	5040	8	1	1	1	1	1	1	1	1	$\cdots$
4	40320	1	1	1	1	1	1	1	1	1	$\cdots$
$\#.d' \downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

To compute the factorial of n (in the above example, $n = 8$), we embed the numbers from 1 to n in a sea of 1s in the d-direction, then multiply pairwise in the d'-direction until we reach one number:

- $1 \times 1 = 1,\ 2 \times 3 = 6,\ 4 \times 5 = 20,\ 6 \times 7 = 42,\ 8 \times 1 = 8,\ \ldots$
- $1 \times 6 = 6,\ 20 \times 42 = 840,\ 8 \times 1 = 8,\ \ldots$
- $6 \times 840 = 5040,\ 8 \times 1 = 8,\ \ldots$
- $5040 \times 8 = 40320,\ \ldots$

$$
\begin{array}{l}
\texttt{fun}\ fact.n = F \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow 0 \\
\quad \texttt{dim}\ d' \leftarrow ilog.n \\
\quad \texttt{var}\ F = fby.d'\ \big(default.d.1.n\ (\#.d)\ 1\big)\ \big(lPair.d\ F \times rPair.d\ F\big) \\
\texttt{end}
\end{array}
$$

The *ilog* function is the integer base-2 logarithm: $ilog.n = \lceil \log_2(n+1) \rceil$.

5.4 Sieve of Eratosthenes

The sieve of Eratosthenes creates an intension varying in dimension d of the prime numbers. It is built using a local dimension d', and presented below as a two-dimensional table. The zeroth

row is the naturals ≥ 2, and each subsequent row is the previous row without the multiples of the zeroth element of the previous row. The sequence of primes is formed by the zeroth column.

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

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

5.5 Matrix multiplication

Suppose we wish to multiply two matrices, $A_{row:m\times col:p}$ and $B_{row:p\times col:n}$, each varying in dimensions *row* and *col*, where the number of columns of A equals p, as does the number of rows of B. Then we would want to write *multiply.row.col.p A B* for their multiplication.

Here is the definition of *multiply*:

```
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
```

The formal parameters X and Y are assumed to vary with respect to formal parameters d_r (row) and d_c (column), while formal parameter k corresponds to the number of columns in X and the number of rows in Y. Here is the meaning of the other identifiers:

- d is an additional, temporary dimension;
- X' corresponds to changing variance in the d_r and d_c dimensions in X to variance in the d_r and d dimensions;
- Y' corresponds to changing variance in the d_r and d_c dimensions in Y to variance in the d and d_c dimensions;
- 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;

5.6 Arrays of functions

The final example, which motivated much of the work presented in this paper, comes from the end of the book, *Lucid, the Dataflow Programming Language* [18], in which a hypothetical language

called *Lambda Lucid*, allowing streams of functions, is presented. The function $pow.n$, defined below, returns a function, namely the n-th–power function, such that $pow.n.m$ calculates the value m^n.

```
fun pow.n = P
where
  dim d ← n
  var P = fby.d (λ m → 1) (λ {d} m → m × P.m)
end
```

The explicit $\{d\}$ in the second λ ensures that the d-ordinate needed to evaluate P within the abstraction is frozen at the time of creation of the abstraction. Here is the table for P:

P	0	1	2	$\overset{\#.d}{\rightarrow}$
	$\lambda m \rightarrow m^0$	$\lambda m \rightarrow m^1$	$\lambda m \rightarrow m^2$	$\cdots$

A divide-and-conquer version of P could also be defined.

6 Conclusions

In this paper, we have presented the TL programming language, its denotational semantics, and extended the core language to a set of syntactic constructs allowing the programming of a wide range of problems from a multidimensional, intensional perspective. The significance of these results is manifold.

Since its inception, Lucid and its descendants have not been considered to be full-fledged higher-order functional languages, because of their lack of higher-order functions. This problem is now solved with TL, since we can now build multidimensional variables of higher-order functions, as well as higher-order functions over multidimensional variables.

Specifically, in this paper, we have solved four key longstanding problems:

1. design, semantics and implementation of higher-order functions over Lucid streams [1];
2. design, semantics and implementation of hypothetical Lambda Lucid, with streams of functions [18];
3. denotational semantics of Indexical Lucid [2];
4. indexical implementation of higher-order functions [14].

Historically, the main difficulty for adding higher-order functions to Lucid was at the implementation level. The standard implementation was a demand-driven interpreter, using memoization (with cache) of previously computed (*variable*, *context*) pairs.

The origins for function implementations date back to [19], in which it was shown how first-order functions for Lucid could be implemented. A dimension was introduced, whose ordinate was a list encoding the actual parameters for all of the currently active functions. This idea was formalized and proven correct in [13]. The latter two authors subsequently generalized their solution to a limited class of higher-order functions, which could take other functions as parameters, but could not return functions, nor be partially applied [14]. This latter solution required a separate dimension for every order of function, therefore necessitating that a type inference algorithm be applied to the function definitions and applications before this transformation could take place.

Since TL is a descendant of ISWIM [8], all of the identifiers that are created in `wheredim` and `wherevar` clauses are lexically scoped. However, the context is dynamically bound; it *permeates* the entire program, as would distributed, global variables in an imperative language, or as do the environment variables of Unix processes. The ordinate of a dimension d can be changed at one point and affect the evaluation of an expression passed by name to the body of a function. It

is therefore necessary to develop a full programming methodology, a few examples of which were given in §5.

The semantics presented in §3 manipulates in the context a special dimension ρ, whose ordinate is a list encoding the path from the root of the evaluation tree of the expression being evaluated. Using the ρ-ordinate upon entry to a `wheredim` clause guarantees that there is no possibility of dimension clash when allocating new dimensions.

This approach ensures a deterministic approach to dimension allocation. The original semantics, not presented here, uses an infinite set Δ of dimensions from which one can be picked in a nondeterministic manner upon entry to a `wheredim` clause. When an expression has n subexpressions, then the set Δ is split into n sets, Δ_1 through Δ_n, all still infinite. The two semantics are equivalent, but the one presented in this paper has the advantage of being more directly implementable.

In fact, this semantics is the basis for the current implementation, which is available online at `translucid.web.cse.unsw.edu.au` , along with documentation and examples. The implementation handles λ-abstractions and `wheredim` clauses slightly differently, by replacing manipulation of the environment defining the variables by manipulation of the context. A new, hidden dimension is introduced for each formal parameter for each λ-abstraction, and for each dimension identifier in a `wheredim` clause. Because the context uses dynamic binding, while the environment uses lexical binding, the new abstract syntax resulting from these changes requires keeping track, at each λ-abstraction, of the ordinates of dimensions from encompassing λ-abstractions and `wheredim` clauses.

This implementation can optionally run with a cache, as outlined in [3]. However, the cache supposes that a dimension identifier in a `wheredim` clause can always be mapped to a single dimension. We have found that this assumption is valid for all reasonable programs we have been able to write. A proper static semantics is currently under development, including type inference, constant folding and rank analysis.

In the implementation, there is a special dimension called `time`, and all variables in a program vary with respect to this dimension, which is *causal*: the value of variable A for a context κ such that $\kappa(\texttt{time}) = T$ cannot depend on the value of any variable, including itself, for a context κ' such that $\kappa'(\texttt{time}) = T'$, where $T' > T$.

The introduction of the `time` dimension makes it possible for a TL program to vary over time: not just the values of variables, but even their very definitions. As a result, the semantics of a whole TL program — as opposed to that of a TL expression, as given in this paper — is similar to that of a LUSTRE program [4]. Formalizing this idea will require the introduction of an *intension η clocked by a dimension t*, meaning that intension η is constrained so that the restriction of the domain of η to a time slice $t \in [0, T]$, $T \in \mathbb{N}$, remains an intension. The work on coalgebras and comonads for timed streams, such as [16, 17], will be useful.

Finally, it should be stated that as the key semantic and implementation issues of TransLucid are resolved, we are focusing more of our attention on methodological issues. We believe that programming with multidimensional, infinite data structures in an intensional manner allows us to view programming from a completely new perspective. This intuition will only be confirmed through experimentation with real problems, such as with multidimensional databases and simulations.

References

[1] E. A. Ashcroft and W. W. Wadge. Lucid, A Nonprocedural Language with Iteration. *Comm. of the ACM*, 20(7):519–526, July 1977.

[2] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming.* Oxford University Press, New York, 1995.

[3] Jarryd P. Beck, John Plaice, and William W. Wadge. Multidimensional infinite data in the language Lucid. *Mathematical Structures in Computer Science*, 2013. In press.

[4] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. A declarative language for programming synchronous systems. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1987*, pages 178–188, Munich, January 1987.

[5] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In Robert Harper and Richard L. Wexelblat, editors, *ICFP*, pages 226–238. ACM, 1996.

[6] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics*. D. Reidel, Dordrecht, Holland, 1981.

[7] A. A. Faustini and W. W. Wadge. Intensional programming. In J. C. Boudreaux, B. W. Hamil, and R. Jenigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier North-Holland, 1987.

[8] Peter J. Landin. The next 700 programming languages. *Comm. of the ACM*, 9(3):157–166, 1966.

[9] Daniel Lemire. Data Warehousing and OLAP: A Research-Oriented Bibliography, 2010. `lemire.me/OLAP`.

[10] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[11] John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Mathematics in Computer Science*, 2(1):37–61, 2008.

[12] John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential demand-driven evaluation of Eager TransLucid. In *COMPSAC*, pages 1266–1271. IEEE Computer Society, 2008.

[13] Panos Rondogiannis and William W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

[14] Panos Rondogiannis and William W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, May 1999.

[15] Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[16] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

[17] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electr. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.

[18] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[19] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

A Proof of Proposition 1

Proof. Suppose $(d_i)_{i\in\mathbb{N}}$ is an $\sqsubseteq$-increasing chain in $\mathbf{D}_\perp$. Then, unless all the $d_i = \perp$, there exists a j such that for all $k \geq j$, d_k will belong to one of the above enumerated cases. We consider them each in turn.

1. Case $(D_\perp, \sqsubseteq)$. This is a flat order, hence a cpo.

2. Case $(\mathbf{D}_{\mathtt{atomic},m}, \sqsubseteq)$. This is the standard order on the set of partial functions from D^m to D, hence a cpo.

3. Case $(\mathbf{D}_{\mathtt{ctxt}}, \sqsubseteq)$. This is the standard order on the set of partial functions from D to $\mathbf{D}$, hence a cpo.

4. Case $(\mathbf{D}_{\mathtt{intens}}, \sqsubseteq)$. This is the non-standard case. Define $\eta_i = d_{i+j}$, $i \in \mathbb{N}$. We define the function $\eta_\sqcup$ as follows:

$$\begin{aligned} \mathrm{dom}(\eta_\sqcup) &= \bigsqcup_i \mathrm{dom}\ \eta_i \quad \text{and, for } \kappa \in \mathrm{dom}(\eta_\sqcup), \\ \eta_\sqcup(\kappa) &= \eta_{i_\kappa}(\kappa), \quad i_\kappa \text{ is the least } i \text{ s.t. } \kappa \in \mathrm{dom}\ \eta_i. \end{aligned}$$

   Since, for all i, $\mathrm{dom}\ \eta_i \in \mathbf{D}_{\mathtt{ctxt}}$, it follows that $\mathrm{dom}\ (\eta_\sqcup) \in \mathbf{D}_{\mathtt{ctxt}}$.

   Now suppose that $\kappa \in \mathrm{dom}\ \eta_\sqcup$. Then there exists i_κ such that $\kappa \in \mathrm{dom}\ \eta_{i_\kappa}$. But since $\eta_{i_\kappa} \in \mathbf{D}_{\mathtt{intens}}$, it follows that for all κ' such that $\kappa = \kappa' \lhd \mathrm{dom}\ \kappa$, that $\eta_{i_\kappa}(\kappa') = \eta_{i_\kappa}(\kappa)$, hence $\eta_\sqcup(\kappa') = \eta_\sqcup(\kappa)$. Because of the intension property, $i_{\kappa'} \leq i_\kappa$. But should $i_{\kappa'} < i_\kappa$, because the η_i form an increasing chain, it follows that $\eta_{i_{\kappa'}}(\kappa') = \eta_{i_\kappa}(\kappa') = \eta_\sqcup(\kappa')$. Since κ was chosen arbitrarily, it follows that $\eta_\sqcup \in \mathbf{D}_{\mathtt{intens}}$.

   Now suppose that η_b is an upper bound of the η_i. Then, for each η_i, $\mathrm{dom}\ \eta_i \sqsubseteq \mathrm{dom}\ \eta_b$. Hence $\mathrm{dom}(\eta_\sqcup) \sqsubseteq \mathrm{dom}\ \eta_b$, and so $\eta_\sqcup \sqsubseteq \eta_b$. Hence $\eta_\sqcup$ is the least upper bound of the chain of η_i. It follows that $(\mathbf{D}_{\mathtt{intens}}, \sqsubseteq)$ is a cpo.

5. Case $(\mathbf{D}_{\mathtt{func}}, \sqsubseteq)$. This is the standard order on the set of partial functions from $\mathbf{D}$ to $\mathbf{D}$, hence a cpo.

Therefore $(\mathbf{D}_\perp, \sqsubseteq)$ is a cpo.