Higher-order Multidimensional Programming

John Plaice Jarryd P. Beck
{plaice,jarrydb}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-1215
August 2012

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

Abstract

We present a higher-order functional language in which variables define arbitrary-dimensional entities, where any atomic value may be used as a dimension, 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. At the core of the design of functions is the intension abstraction, a parameterless function whose body is evaluated with respect to the context in which it is used and to part of the context in which it is created.

The multidimensional space can be used for both programming and implementation purposes. At the programming level, the informal presentation of the language gives many examples showing the utility of describing common computing entities as infinite multidimensional data structures. At the implementation level, the main technical part of the paper demonstrates that the higher-order functions over infinite data structures—even ones that are curried—can be statically transformed into equivalent functions directly manipulating the context, thereby replacing closures over parts of the environment by closures over parts of the context.

1 Introduction

We introduce TransLucid (TL), a language with higher-order functions in which variables define arbitrary-dimensional arrays of arbitrary extent. The focus of this paper is to present the design and denotational semantics of TL, and to demonstrate how this semantics can be transformed into one which is readily amenable to implementation. As a result, the solution to a number of semantic and implementation problems for languages descending from Wadge and Ashcroft's Lucid [1], some dating back to the 1970s, are given here. The TL language presented in this paper is untyped, in order to simplify the presentation; all results should generalize to the typed situation.

In many areas of computer science, multidimensional data are common, and play an increasingly important rôle. In TL, a variable X is understood to vary, conceptually, in *all* possible dimensions; this means, say, that if X is defined using two dimensions d_1 and d_2, and Y is defined using two dimensions d_2 and d_3, then both can be considered to vary in all three dimensions: the "variance" of X in d_3 is constant, as is the "variance" of Y in d_1. As for their sum, $X + Y$, 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 used to define the semantics of relational databases [7].

In TL, the evaluation of variables is done in a lazy manner; they are indexed by a runtime multidimensional context, an unordered set of (*dimension*, *ordinate*) pairs, in which any atomic value may play the rôle of dimension. To define the variance of variables with respect to the context, the latter can be perturbed by changing some of the (*dimension*, *ordinate*) pairs defining it. The runtime context permeates an entire program, and dimensions are queried using dynamic binding.

The difficulty in designing this language, and one of the core aspects of this paper, lies in defining the semantics for functions, which must manipulate these multidimensional arrays. If functions are to be first-class values in TL, then it should be possible to build multidimensional arrays of functions, and also that the possible contexts in which a function might be used will in general be different from the context in which the function is created.

To illustrate this problem, we program in TL an example from the end of Wadge and Ashcroft's 1986 book, *Lucid, the Dataflow Programming Language* [13], in which they presented a hypothetical language called *Lambda Lucid*. The function *pow.n*, defined below, returns a function, namely the n-th–power function, i.e., *pow.n.m* calculates the value m^n.

$$
\begin{array}{l}
\texttt{fun}\ \mathit{pow}.n = P \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ P = \mathit{fby}.d\ (\lambda^{\mathrm{b}}\ m \to 1)\ (\lambda^{\mathrm{b}}\ \{d\}\ m \to m \times P.m) \\
\quad \texttt{fun}\ \mathit{fby}.d\ X\ Y = \texttt{if}\ \#.d \equiv 0\ \texttt{then}\ X \\
\qquad\qquad\qquad\qquad \texttt{else}\ Y\ \texttt{@}\ [d \leftarrow \#.d - 1]\ \texttt{fi} \\
\texttt{end}
\end{array}
$$

The definition of *pow* is made with a *dimensionally abstract* `where` *clause*, in which a local dimension d, a variable P and a function *fby* are declared. The stream P, varying in dimension d, is defined to be a sequence of functions, where the i-th element is the i-th–power function, as follows:

	dim d $\to$			
P	0	1	2	$\cdots$
	$\lambda^{\mathrm{b}}\ m \to m^0$	$\lambda^{\mathrm{b}}\ m \to m^1$	$\lambda^{\mathrm{b}}\ m \to m^2$	$\cdots$

The line `dim` $d \leftarrow n$ not only declares the local dimension d, but also states that its initial ordinate should be n. The net result is to extract the n-th element of P, i.e., the n-th–power function.

The function *fby*, taking three parameters, is used to define a stream from two others. Should $A = \langle a_0, a_1, a_2, \ldots \rangle$ and $B = \langle b_0, b_1, b_2, \ldots \rangle$ be two streams varying in dimension s, then $\mathit{fby}.s\ A\ B = \langle a_0, b_0, b_1, b_2, \ldots \rangle$ is also a stream varying in dimension s. The actual parameter s is passed by

value, while actual parameters A and B are passed by name. As for *fby*, it could be defined using λ^{b} (base) and λ^{n} (call-by-name) abstractions:

$$\begin{aligned}\texttt{var}\ \mathit{fby} = \lambda^{\mathrm{b}}\ d \to \lambda^{\mathrm{n}}\ X \to \lambda^{\mathrm{n}}\ Y \to \\ \texttt{if}\ \#.d \equiv 0\ \texttt{then}\ X\ \texttt{else}\ Y\ @\ [d \leftarrow \#.d - 1]\ \texttt{fi}\end{aligned}$$

The body of a base function f is only evaluated with respect to the context in which f is created, and the only dimensions of relevance from this context are the ones explicitly written down. If we consider the first abstraction ($\lambda^{\mathrm{b}}\ m \to 1$), the evaluation of the body 1 is not subject to any context, while in the second abstraction ($\lambda^{\mathrm{b}}\ \{d\}\ m \to m \times P.m$), the evaluation of the body $m \times P.m$ is made with respect to the d-ordinate of the context upon creation; as a result, the evaluation of P takes into account and freezes the d-ordinate upon creation of the abstraction.

The body of a call-by-name function g, on the other hand, is evaluated in the context in which g is applied. In the definition of *fby*, the parameters X and Y are evaluated with respect to the context of application. The expression $\#.d$ queries the d-ordinate from $\#$, which is the current context. The expression $Y\,@[d \leftarrow \#.d-1]$ means decrementing the d-ordinate of the current context before evaluating Y.

The fact that a variable, conceptually, varies in all dimensions forces the programmer to think in an *intensional* manner, as it is in general not possible, nor desirable, to enumerate all of these dimensions. In logic, the *intension* of an utterance is a function from the *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 [3, 12] for writings on Montague's intensional logic.) For example, the phrase "*Five degrees less than yesterday's temperature*" could be written as the expression

$$(\mathit{temperature} - 5)\ @\ [\mathit{date} \leftarrow \#.\mathit{date} - 1]$$

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 according to many other dimensions, such as *latitude*, *longitude*, *altitude*, *timeOfDay*, and so forth.

This idea was first proposed by Faustini and Wadge, in their paper "Intensional Programming" [4], and again in the collective work *Multidimensional Programming* [2] (p.26) (see §2 for further discussion). However, they state, "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."

For the *programmer*, this approach is correct. Nevertheless, the denotational semantics for TL, given in §4, is done precisely in this manner. The semantics of an expression E is purely extensional, supposing an interpretation of the constant symbols, an environment mapping identifiers to context-to-value functions, a large set of unused dimensions, and the runtime context.

This denotational semantics, however, is not effective, as the mapping of dimension identifiers to dimensions in the evaluation of `where` clauses is done by randomly choosing from the large set of unused dimensions. Furthermore, the manipulation of the environment and the manipulation of the context in the semantic rules resemble each other, which seems to indicate a simpler implementation is possible.

In §6, we show how function abstractions and applications, as well as the generation of local dimensions, can be rewritten through the manipulation of the runtime context with *hidden* dimensions, not available to the original programmer, using a new, *contextual* semantics, which *is* directly implementable.

To illustrate this idea, consider the application

$$\big(\lambda^{\mathrm{b}} x \to \lambda^{\mathrm{b}} y \to (x + y)\big).a.b$$

In the translation, the formal parameters x and y are replaced by dimensions ϕ_x and ϕ_y in the context, which is perturbed during the application. As a result the overall application is equivalent to

$$(\#.\phi_x + \#.\phi_y)\ @\ [\phi_x \leftarrow a, \phi_y \leftarrow b]$$

Once the functions have been transformed so that they no longer manipulate the environment, the `where` clauses defining variable identifiers can be collapsed into a single `where` clause, meaning that an implementation only needs to manipulate the context, not the environment.

The TL language has been implemented, and a full Web-enabled interpreter, along with documentation and examples, can be found at `translucid.web.cse.unsw.edu.au`. It is the first descendant of Lucid which fully supports higher-order functions.

2 Background

The origins of the work presented in this paper go back to the mid-1970s, to the Lucid programming language developed by Wadge and Ashcroft [1]. (See reference [8] for a detailed history of Lucid and its successors.) In Lucid, a variable is defined to be an infinite stream, by giving its first element and then the rules for creating subsequent elements from previous elements. For example,

$$X = 0 \; \mathit{fby} \; (X + 1)$$

defines the sequence $\langle x_0, x_1, x_2, \ldots \rangle$ given by

$$\begin{aligned} x_0 &= 0 \\ x_{i+1} &= x_i + 1 \end{aligned}$$

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

A Lucid stream is not a physical data structure, but a conceptual one, so one can truly talk about an infinite stream, as it does not need to 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.

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 this paper. 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 [8], and memoization techniques for its implementation have been presented in [9]. However, up to now, TransLucid has had no user-defined functions.

The TL language—TransLucid with functions—presented in this paper solves all of these problems. Its syntax and denotational semantics have been adapted so that all atomic values can be used as dimensions, and functions can be higher-order and curried with partial application. The solution uses completely standard notions from programming-language theory: call-by-name and call-by-value, and lexical binding and dynamic binding.

Clearly, passing a potentially infinite, multidimensional data structure to a function can only be done lazily. It is therefore normal that these structures be passed by name. However, to actually probe one of these data structures, one needs to *fix* which dimensions to manipulate. It follows that these parameters *must* be evaluated eagerly and be passed by value.

As for binding, since TL is a descendant of ISWIM [5], all of the identifiers that are created in `where` 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. The binding of dimensions in the runtime context is therefore dynamic.

The tension between call-by-name and call-by-value has been extensively discussed and studied. Lewis *et al.* [6] did develop a language in which certain identifiers could be scoped dynamically, but these identifiers could not be passed as first-class values.

Call-by-value is generally associated with efficiency, and call-by-name is assumed to require the use of closures for implementation. For example, in [14], Wadler states that "lazy, or call-by-need

[memoized version of call-by-name], languages schedule work dynamically by building closures". In TL, the closures encapsulate an expression to be evaluated and a subset of the current context, and are used to build intension and function abstractions.

The origins of the implementation presented here date back to Yaghi's thesis [15], in which he showed 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 by Rondogiannis and Wadge [10]. The latter two 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 [11]. 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.

The solution we provide here will use a separate dimension for each functional abstraction. Typing of the functions is not necessary, and closures of the environment are not at all needed, nor are the dimensions list-valued. The closures of the context needed are for holding on to the existing actual parameters of a partially applied function, as well as the dimensions made explicit in intension abstractions.

3 Higher-order Multidimensional Programming

Like in all languages derived from ISWIM, a program in TL is an expression. All TL expressions 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.

3.1 Constants

The simplest expression in TL is the *constant*. If we consider expression '42', its value is 42, whatever the context. Below, we show the value of expression '42' if we allow dimension 0 to vary in $\mathbb{N}$.

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

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

3.2 Variables

Variables are identifiers denoting objects which vary with the context. Suppose that variable X defines a stream $\langle x_0, x_1, \ldots \rangle$ varying in dimension 0, then we can see below how the value of expression 'X' varies if we allow dimension 0 to vary in $\mathbb{N}$.

	dim 0 $\rightarrow$						
X	0	1	2	3	4	5	$\cdots$
	x_0	x_1	x_2	x_3	x_4	x_5	$\cdots$

What this table means is that in context $\{0 \mapsto 0\}$, i.e., where dimension 0 takes on the value of 0, the value of expression 'X' is x_0. In general, in all contexts $\{0 \mapsto i\}$, where $i \in \mathbb{N}$, the value of 'X' is x_i.

3.3 Pointwise operators

As in all languages, TL expressions allow the use of operators such as + for addition, × for multiplication, and so on. In TL, these operators are *base functions*, whose evaluation does not vary with the context; they are applied *pointwise* to their arguments. In other words, in a given context κ, the expression $+.(X, Y)$, yields the sum of X in κ and of Y in κ. If $X = \langle x_0, x_1, \ldots \rangle$ and $Y = \langle y_0, y_1, \ldots \rangle$ are both streams varying in dimension 0, then the variance of '$+.(X, Y)$' with respect to dimension 0 is given below.

	dim 0 →				
$+.(X,Y)$	0	1	2	3	⋯
	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	⋯

In context $\{0 \mapsto i\}$, '$+.(X, Y)$' has value $x_i + y_i$, where $i \in \mathbb{N}$.

From now on, to simplify the presentation, all the standard binary operators will be presented in infix format, i.e., we will write '$X + Y$' instead of '$+.(X, Y)$'.

3.4 Querying the context

In TL, the context is itself a base function, written #. For it to be possible for the context to affect the result of the evaluation of an expression, the context must be *queried*: the expression '#.0' means applying the context to dimension 0 to retrieve the corresponding ordinate. Below we show how the value of '#.0' varies when we let dimension 0 vary:

	dim 0 →						
#.0	0	1	2	3	4	5	⋯
	0	1	2	3	4	5	⋯

In, say, context $\{0 \mapsto 4\}$, the value of '#.0' is 4. In fact, for all contexts $\{0 \mapsto i\}$, where $i \in \mathbb{N}$, the value of '#.0' is i.

Now that the context can be queried, we can write expressions that are dependent on the context. Here is the evaluation of expression '#.0 + #.1' with respect to dimensions 0 and 1.

	dim 0 →						
#.0 + #.1	0	1	2	3	4	5	⋯
dim 1 ↓ 0	0	1	2	3	4	5	⋯
1	1	2	3	4	5	6	⋯
2	2	3	4	5	6	7	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

In context $\{0 \mapsto i, 1 \mapsto j\}$, for all $i, j \in \mathbb{N}$, '#.0 + #.1' has value $i + j$.

3.5 Tuples

In TL, a tuple is a base function, defined as a set of (*dimension*, *value*) pairs. For example, '[0 ← #.0 + 1, 1 ← #.1 + 3].0' varies as follows in dimensions 0 and 1:

	dim 0 →						
[0 ← #.0 + 1, 1 ← #.1 + 3].0	0	1	2	3	4	5	⋯
dim 1 ↓ 0	1	2	3	4	5	6	⋯
1	1	2	3	4	5	6	⋯
2	1	2	3	4	5	6	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

As for '$[0 \leftarrow \texttt{\#}.0 + 1, 1 \leftarrow \texttt{\#}.1 + 3].1$', it varies as follows in dimensions 0 and 1:

$[0 \leftarrow \texttt{\#}.0 + 1, 1 \leftarrow \texttt{\#}.1 + 3].1$	dim 0 → 0	1	2	3	4	5	⋯
dim 1 ↓ 0	3	3	3	3	3	3	⋯
1	4	4	4	4	4	4	⋯
2	5	5	5	5	5	5	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

3.6 Changing the context

If the context can be queried in TL, then it also needs to be changeable. This is done using the '`@`' operator, which takes a tuple as parameter and uses it to change the current context before continuing with the evaluation of the expression. Below, the expression '$\texttt{\#}.0 + \texttt{\#}.1$' is evaluated in a new context, which is created by incrementing the 0-ordinate by 1.

$(\texttt{\#}.0 + \texttt{\#}.1)\ \texttt{@}\ [0 \leftarrow \texttt{\#}.0 + 1]$	dim 0 → 0	1	2	3	4	5	⋯
dim 1 ↓ 0	1	2	3	4	5	6	⋯
1	2	3	4	5	6	7	⋯
2	3	4	5	6	7	8	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Expression '$(\texttt{\#}.0 + \texttt{\#}.1)\ \texttt{@}\ [0 \leftarrow \texttt{\#}.0 + 1]$' therefore evaluates to $i + j + 1$ in context $\{0 \mapsto i, 1 \mapsto j\}$.

3.7 Factorial: version one

TL, of course, needs variables, defined through equations. We introduce these with the factorial function, presented here as a variable varying in dimension 0, whose first few entries can be found below:

fact	dim 0 → 0	1	2	3	4	5	6	7	⋯
	1	1	2	6	24	120	720	5040	⋯

The definition in TL is recursive, with a base case and an inductive case:

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

3.8 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 1 and 0.

ack	dim 0 → 0	1	2	3	4	5	⋯
dim 1 ↓ 0	1	2	3	4	5	6	⋯
1	2	3	4	5	6	7	⋯
2	3	5	7	9	11	13	⋯
3	5	13	29	61	125	253	⋯
4	13	65533	⋯				
5	65533	⋯					
⋮	⋱						

Here is the definition for Ackermann in TL:

$$\begin{aligned}&\texttt{var } ack = \texttt{if \#.1} \equiv 0 \texttt{ then \#.0} + 1\\ &\qquad\texttt{elsif \#.0} \equiv 0 \texttt{ then } ack \texttt{ @ } [1 \leftarrow \texttt{\#.1} - 1, 0 \leftarrow 1]\\ &\qquad\texttt{else } ack \texttt{ @ } \big[1 \leftarrow \texttt{\#.1} - 1, 0 \leftarrow ack \texttt{ @ } [0 \leftarrow \texttt{\#.0} - 1]\big] \texttt{ fi}\end{aligned}$$

In the general (`else`) case, note that the nested context change is only changing the value for dimension 0, since the value for dimension 1 need not be changed. This is similar to what happens with differential equations, in which only the dimensions of relevance are written down.

3.9 Standard functions

A function can take any of three kinds of parameter: *base parameters*, *value parameters* and *named parameters*, respectively introduced by '.', '!' and a space. Below are some standard TL functions.

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

The function *index* takes a single value parameter d, and evaluates the body '`#.`d+1' in the context in which the function is *applied*, not created.

The function *first* takes two parameters, a base parameter d, and a named parameter X. The latter is assumed to vary in the dimension d, and the body is evaluated in the context in which the function is applied, thereby pulling the zeroth element of X in dimension d.

The function *next* also takes a base parameter d and a named parameter X. It shifts all of X one step "to the left".

The function *fby* takes three parameters, one base parameter d, and two named parameters, X and Y. It shifts Y one slot "to the right" and inserts the zeroth element of X.

If $A = \langle a_0, a_1, a_2, \ldots\rangle$ and $B = \langle b_0, b_1, b_2, \ldots\rangle$, then

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

The last standard function is *wvr*, which takes one base parameter d, and two named parameters, X and Y. It 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. If $B = \langle T, F, T, T, F, T, T, F, T, \ldots\rangle$, then

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

3.10 Factorial: version two

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

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

It uses a local dimension d, which is initially set to n. The stream F varies in dimension d. Note that $index!d$ increments the d-ordinate, while the second argument of $fby.d$ decrements the d-ordinate, so the two cancel each other out, yielding $\#.d$.

3.11 Ackermann: version two

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

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

Note the replacement of all but one explicit manipulation of dimensions by the use of relative functions *index*, *next* and *fby*.

3.12 Sieve of Eratosthenes

The sieve of Eratosthenes generates a stream 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.

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

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

3.13 Matrix multiplication

We consider a more elaborate example, of multiplying two matrices, $A_{row:m\times col:p}$ and $B_{row:p\times col:n}$, each varying in dimensions *col* and *row*, where the number of columns of A equals p, as does the

number of rows of B. The expression below defines their multiplication:

```
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
  fun rotate.d_1.d_2 X = X @ [d_1 ← #.d_2]
  fun sum.d_x.n X = Y @ [d_x ← n]
  where
    var Y = fby.d_x 0 (X + Y)
  end
end
```

In the function *multiply*, 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;
- function $rotate.d_1.d_2\ X$ changes variance of X in dimension d_1 to variance in dimension d_2;
- function $sum.d_x.n\ X$ adds up the first n elements in direction d_x of stream X.

3.14 Streams of functions

To facilitate the Taylor-series example below, we re-present the *pow* example from the introduction:

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

The explicit $\{d\}$ in the second abstraction 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	dim d → 0	1	2	⋯
	$\lambda^b\ m \to m^0$	$\lambda^b\ m \to m^1$	$\lambda^b\ m \to m^2$	⋯

3.15 Taylor series expansion

With streams of functions, we can compute the Taylor series expansion for a function f around point a for point x.

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

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

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

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

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

3.16 Explicit intensions

As was described in the background section, an intension is a mapping from contexts to values. In TransLucid, intensions can be written explicitly, in order to freeze the ordinates of certain dimensions from the context at creation, even though the intension will be evaluated in some other context.

For example, suppose we wanted to refer to the temperature in Inuvik, wherever we happened to be. Then we could write:

```
var tempAtLocation = ↑{location} temperature
var tempInuvik = tempAtLocation @ [location ← "Inuvik"]
```

Then whenever variable *tempInuvik* would be written, the temperature would always give the temperature in Inuvik, allowing all dimensions other than *location* to vary freely. Hence

```
(↓ tempInuvik) @ [location ← "Paris", date ← #.date − 1]
```

would give the temperature yesterday, for Inuvik, not Paris.

4 Semantics

The denotational semantics computes least fixed points of systems of equations in a semantic domain where the values of variables are *intensions*. After a presentation of notation, we define the domains and the rules, then we demonstrate the soundness of the rules.

4.1 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 $f, g : A \rightarrowtail B$. The *perturbation of f by g* is defined by:

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

$$
\begin{array}{lll}
E ::= & x & \textit{identifier} \\
| & {}^{m}c & \textit{m-ary constant symbol, } m \in \mathbb{N} \\
| & \texttt{\#} & \textit{context} \\
| & [E \leftarrow E,\ \ldots] & \textit{tuple builder} \\
| & \lambda^{\mathrm{b}}\ \{E, \ldots\}\ (x, \ldots) \to E & \textit{base abstraction} \\
| & E\,.\,(E,\ \ldots) & \textit{base application} \\
| & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E\ \texttt{fi} & \textit{conditional} \\
| & E\ \texttt{@}\ E & \textit{context perturbation} \\
| & \uparrow\{E, \ldots\}\ E & \textit{intension abstraction} \\
| & \downarrow E & \textit{intension application} \\
| & \lambda^{\mathrm{v}}\ \{E, \ldots\}\ x \to E & \textit{call-by-value abstraction} \\
| & E\,!\,E & \textit{call-by-value application} \\
| & \lambda^{\mathrm{n}}\ \{E, \ldots\}\ x \to E & \textit{call-by-name abstraction} \\
| & E\ E & \textit{call-by-name application} \\
| & E\ \texttt{wheredim}\ x \leftarrow E,\ \ldots\ \texttt{end} & \textit{local dimensions} \\
| & E\ \texttt{wherevar}\ x = E,\ \ldots\ \texttt{end} & \textit{local variables}
\end{array}
$$

Figure 1: Syntax of TL expressions

- Let f be a function with finite domain $\{v_1, \ldots, v_m\}$. Then f can be given as its *graph* $\{v_1 \mapsto f(v_1), \ldots, v_m \mapsto f(v_m)\}$.
- Let $f : A \rightarrowtail B$, and let $S \subseteq A$. The *domain restriction of f to S* is defined by

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

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

$$(f \lhd\!\!\!\cdot\, S)(v) = f \lhd (A - S).$$

4.2 Syntax

Definition 1. *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 for which $ar({}^{m}c) = m$.*

Definition 2. *Let Σ be a signature and let D be a set of* calculable 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 $\iota({}^{m}c) : D^m \rightarrowtail D$. We write* $\mathbf{Interp}(\Sigma, D)$ *for the set of interpretations of Σ over D.*

Definition 3. *Let Σ be a signature and X be a set of identifiers. A* TL expression *over Σ and X is an expression E whose abstract syntax satisfies the grammar given in Figure 1. The* free variables *of E are written $FV(E)$. If $x \notin FV(E')$, then a substitution of E' for the variable x in E is written $E[x/E']$. We write* $\mathbf{Expr}(\Sigma, X)$ *for the set of TL expressions over Σ and X.*

Note that the abstract syntax has no `where` clause, nor `fun` declaration. To get to this abstract syntax, the concrete syntax presented in the previous section must be passed through two textual filters, first $\mathcal{T}_0$, then $\mathcal{T}_1$. Transformation $\mathcal{T}_0$ is given below:

$$
\begin{aligned}
\mathcal{T}_0(\texttt{fun}\ x\ arg_i = E)_{i=1..m} &= \big(\texttt{var}\ x = \mathcal{T}_0'(arg_i)\ \mathcal{T}_0(E)\big) \\
\mathcal{T}_0'(.x_i) &= (\lambda^{\mathrm{b}}\ x_i \to\ ) \\
\mathcal{T}_0'(!x_i) &= (\lambda^{\mathrm{v}}\ x_i \to\ ) \\
\mathcal{T}_0'(\ x_i) &= (\lambda^{\mathrm{n}}\ x_i \to\ )
\end{aligned}
$$

where $\mathcal{T}_0'$ chooses the right kind of λ-abstraction, and all the other constructs are mapped trivially.

Transformation $\mathcal{T}_1$ is given below:

$$\mathcal{T}_1 \begin{pmatrix} E \texttt{ where} \\ \quad \texttt{dim } x_i \leftarrow E_i \\ \quad \texttt{var } x_j' = E_j' \\ \texttt{end}_{i=1..m,\ j=1..n} \end{pmatrix} = \begin{pmatrix} \big(\mathcal{T}_1(E) \texttt{ wherevar } x_j' \leftarrow \mathcal{T}_1(E_j') \texttt{ end}\big) \\ \texttt{wheredim } x_i \leftarrow \mathcal{T}_1(E_i) \texttt{ end} \end{pmatrix}$$

where all the other constructs are mapped trivially.

4.3 Domains

Definition 4. *Let D be an enumerable set of values. The* semantic domain $\mathbf{D}$ derived from D *is the least solution to the equations*

$$\begin{aligned} \mathbf{D} &= D \cup \big(\textstyle\bigcup_{m>0} \mathbf{D}_{\texttt{base},m}\big) \cup \mathbf{D}_{\texttt{play}} \cup \mathbf{D}_{\texttt{value}} \cup \mathbf{D}_{\texttt{name}} \\ \mathbf{D}_{\texttt{base},m} &= D^m \rightarrowtail \mathbf{D}, \qquad \textit{for } m \in \mathbb{N} - \{0\} \\ \mathbf{D}_{\texttt{ctxt}} &= \mathbf{D}_{\texttt{base},1} \\ \mathbf{D}_{\texttt{intens}} &= \mathbf{D}_{\texttt{ctxt}} \rightarrowtail \mathbf{D} \\ \mathbf{D}_{\texttt{play}} &= \mathcal{P}(D) \rightarrowtail \mathbf{D}_{\texttt{intens}} \\ \mathbf{D}_{\texttt{value}} &= \mathbf{D} \rightarrowtail \mathbf{D}_{\texttt{play}} \\ \mathbf{D}_{\texttt{name}} &= \mathbf{D}_{\texttt{play}} \rightarrowtail \mathbf{D}_{\texttt{play}} \end{aligned}$$

where

- *a subset of D is written Δ;*
- *for all $\eta \in \mathbf{D}_{\texttt{intens}}$, if $\kappa \in \mathrm{dom}\ \eta$, then for all κ' such that $\kappa = \kappa' \lhd (\mathrm{dom}\ \kappa)$, we have $\eta(\kappa) = \eta(\kappa')$;*
- *for all $\pi \in \mathbf{D}_{\texttt{play}}$, if $\Delta \in \mathrm{dom}\ \pi$, then for all $d \in \Delta$, we have $d \notin \mathrm{dom}(\pi(\Delta))$.*

We call

- $\mathbf{D}_{\texttt{base},m}$, *$m > 0$, the set of* base functions of arity *m;*
- $\mathbf{D}_{\texttt{ctxt}}$ *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;*
- $\mathbf{D}_{\texttt{play}}$ *the set of* intensional playgrounds*; an intensional playground is written π;*
- $\mathbf{D}_{\texttt{value}}$ *the set of* call-by-value functions*;*
- $\mathbf{D}_{\texttt{name}}$ *the set of* call-by-name functions.

The restriction that for all κ' such that $\kappa = \kappa' \lhd (\mathrm{dom}\ \kappa)$, we have $\eta(\kappa) = \eta(\kappa')$, ensures that the addition of new information cannot put into question previous decisions. 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.

An intensional playground takes as input a set of dimensions Δ and returns as result an intension η, such that the domain of η is disjoint from Δ. The idea is that the dimensions in Δ can be used to compute η, but that they do not show up in the result; they are encapsulated.

Definition 5. *Let D be an enumerable set of values, $\mathbf{D}$ be the semantic domain derived from D, and $\perp \notin \mathbf{D}$. Then we define the order $\sqsubseteq$ over $\mathbf{D}_\perp = \mathbf{D} \cup \{\perp\}$ by:*

- *For all $d \in \mathbf{D}_\perp$, $\perp \sqsubseteq d$.*
- *For all $d \in D$, $d \sqsubseteq d$.*
- *For all $f, f' \in \mathbf{D}_{\mathtt{base},m}$, $f \sqsubseteq f'$ iff $f = f' \lhd (\mathrm{dom}\ f)$.*
- *For all $\eta, \eta' \in \mathbf{D}_{\mathtt{intens}}$, $\eta \sqsubseteq \eta'$ iff $\eta = \eta' \lhd (\mathrm{dom}\ \eta)$.*
- *For all $\pi, \pi' \in \mathbf{D}_{\mathtt{play}}$, $\pi \sqsubseteq \pi'$ iff $\pi = \pi' \lhd (\mathrm{dom}\ \pi)$.*
- *For all $f, f' \in \mathbf{D}_{\mathtt{value}}$, $f \sqsubseteq f'$ iff $f = f' \lhd (\mathrm{dom}\ f)$.*
- *For all $f, f' \in \mathbf{D}_{\mathtt{name}}$, $f \sqsubseteq f'$ iff $f = f' \lhd (\mathrm{dom}\ f)$.*

We write

- *$\perp_{\mathtt{base},m}$ for the least element of $\mathbf{D}_{\mathtt{base},m}$, with empty domain;*
- *$\perp_{\mathtt{intens}}$ for the least element of $\mathbf{D}_{\mathtt{intens}}$, with empty domain;*
- *$\perp_{\mathtt{play}}$ for the least element of $\mathbf{D}_{\mathtt{play}}$, with empty domain;*
- *$\perp_{\mathtt{value}}$ for the least element of $\mathbf{D}_{\mathtt{value}}$, with empty domain;*
- *$\perp_{\mathtt{name}}$ for the least element of $\mathbf{D}_{\mathtt{name}}$, with empty domain.*

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

1. *$(D_\perp, \sqsubseteq)$, where $D_\perp = D \cup \{\perp\}$;*
2. *$(\mathbf{D}_{\mathtt{base},m}, \sqsubseteq)$, $m \in \mathbb{N} - \{0\}$;*
3. *$(\mathbf{D}_{\mathtt{intens}}, \sqsubseteq)$;*
4. *$(\mathbf{D}_{\mathtt{play}}, \sqsubseteq)$;*
5. *$(\mathbf{D}_{\mathtt{value}}, \sqsubseteq)$;*
6. *$(\mathbf{D}_{\mathtt{name}}, \sqsubseteq)$.*

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; furthermore, if $d_j \in \mathbf{D}_{\mathtt{base},m}$ for some $m > 0$, then so are all d_k, for $k \geq j$. We consider them each in turn.

1. Case $(D_\perp, \sqsubseteq)$. This is a flat order, hence a cpo.
2. Case $(\mathbf{D}_{\mathtt{base},m}, \sqsubseteq)$, $m \in \mathbb{N} - \{0\}$. Define $f_i = d_{i+j}$, $i \in \mathbb{N}$. We define the function $f_\sqcup$ as follows:

$$\mathrm{dom}(f_\sqcup) = \bigcup_i \mathrm{dom}\ f_i \quad \text{and, for } d \in \mathrm{dom}(f_\sqcup),$$
$$f_\sqcup(d) = f_{i_d}(d), \quad i_d \text{ is the least } i \text{ s.t. } d \in \mathrm{dom}\ f_i.$$

   Since, for all i, $\mathrm{dom}\ f_i \subseteq D^m$, it follows that $\mathrm{dom}(f_\sqcup) \subseteq D^m$. Hence $f_\sqcup \in \mathbf{D}_{\mathtt{base},m}$. Furthermore, for all i, $f_i \sqsubseteq f_\sqcup$, hence $f_\sqcup$ is an upper bound of the chain of f_i. Now suppose that f_b is an upper bound of the f_i. Then, for each f_i, $\mathrm{dom}\ f_i \subseteq \mathrm{dom}\ f_b$. Hence $\mathrm{dom}(f_\sqcup) \subseteq \mathrm{dom}\ f_b$, and so $f_\sqcup \sqsubseteq f_b$. Hence $f_\sqcup$ is the least upper bound of the chain of f_i. It follows that $(\mathbf{D}_{\mathtt{base},m}, \sqsubseteq)$ is a cpo.

$$
\begin{aligned}
d \setminus \Delta &= d, \qquad d \notin \Delta \\
d \setminus \Delta &= \bot, \qquad d \in \Delta \\
\kappa \setminus \Delta &= \{ d_i \mapsto \kappa(d_i) \setminus \Delta \mid d_i \in \mathrm{dom}\ (\kappa \lhd \Delta) \} \\
\big(\lambda(d_{a_j}).f(d_{a_j})\big) \setminus \Delta &= \lambda(d_{a_j}).\big(f(d_{a_j}) \setminus \Delta\big) \\
\big(\lambda\Delta_a.\lambda\kappa_a.f(\Delta_a,\kappa_a)\big) \setminus \Delta &= \lambda\Delta_a.\lambda\kappa_a.\big(f(\Delta_a,\kappa_a) \setminus \Delta\big) \\
\big(\lambda d_a.\lambda\Delta_a.\lambda\kappa_a.f(d_a,\Delta_a,\kappa_a)\big) \setminus \Delta &= \lambda d_a.\lambda\Delta_a.\lambda\kappa_a.\big(f(d_a,\Delta_a,\kappa_a) \setminus \Delta\big) \\
\big(\lambda\pi_a.\lambda\Delta_a.\lambda\kappa_a.f(\pi_a,\Delta_a,\kappa_a)\big) \setminus \Delta &= \lambda\pi_a.\lambda\Delta_a.\lambda\kappa_a.\big(f(\pi_a,\Delta_a,\kappa_a) \setminus \Delta\big)
\end{aligned}
$$

Figure 2: Definition of $\setminus$ for denotational semantics

3. Case $(\mathbf{D}_{\mathtt{intens}}, \sqsubseteq)$. 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, dom $\eta_i \in \mathbf{D}_{\mathtt{ctxt}}$, it follows that 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 $\eta_\sqcup(\kappa') = \eta_\sqcup(\kappa)$. Hence $\eta_\sqcup \in \mathbf{D}_{\mathtt{intens}}$. Now suppose that η_b is an upper bound of the η_i. Then, for each η_i, dom $\eta_i \sqsubseteq$ dom η_b. Hence $\mathrm{dom}(\eta_\sqcup) \sqsubseteq$ dom η_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.

4. Case $(\mathbf{D}_{\mathtt{play}}, \sqsubseteq)$. Similar to point (2).

5. Case $(\mathbf{D}_{\mathtt{value}}, \sqsubseteq)$. Similar to point (2).

6. Case $(\mathbf{D}_{\mathtt{name}}, \sqsubseteq)$. Similar to point (3).

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

Definition 6. *Let D be an enumerable set of values and X be a set of variables. Then an* environment *over X and D is a mapping $\xi : X \rightarrowtail \mathbf{D}_{\mathtt{play}}$. A substitution of π for the value of x in ξ is written $\xi[x/\pi]$. The perturbation of ξ by ξ' is written $\xi \dagger \xi'$. We extend the order $\sqsubseteq$ to environments: $\xi \sqsubseteq \xi'$ iff $\forall x \in \mathrm{dom}\ \xi \cup \mathrm{dom}\ \xi'$, $\xi(x) \sqsubseteq \xi'(x)$. A set $\Delta \subset D$ is* irrelevant to ξ *iff $\forall x \in \mathrm{dom}\ \xi, \forall \kappa \in \mathbf{D}_{\mathtt{ctxt}}$, $\xi(x)(\Delta)(\kappa) = \xi(x)(\Delta)(\kappa \lhd \Delta)$. We write $\mathbf{Env}(X, D)$ for the set of environments over X and D.*

4.4 Notation for semantic rules

- Let $d \in \mathbf{D}$. The *constant intension* for d is defined by:

$$\overline{d} = \lambda\kappa.d$$

- Let $\eta \in \mathbf{D}_{\mathtt{intens}}$. The *constant intensional playground* for η is defined by:

$$\widehat{\eta} = \lambda\Delta.\eta$$

- The masking of a value d by a set Δ, written $d \setminus \Delta$, is defined in Figure 2.

$$[\![x]\!]\iota\xi\Delta\kappa = \xi(x)(\Delta)(\kappa) \tag{1}$$

$$[\![{}^m c]\!]\iota\xi\Delta\kappa = \iota({}^m c) \tag{2}$$

$$[\![\texttt{\#}]\!]\iota\xi\Delta\kappa = \kappa \tag{3}$$

$$[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]\iota\xi\Delta\kappa = \big\{[\![E_{i0}]\!]\iota\xi\Delta_{i0}\kappa \mapsto [\![E_{i1}]\!]\iota\xi\Delta_{i1}\kappa\big\} \qquad \textstyle\bigcup_i(\Delta_{i0} \cup \Delta_{i1}) \subseteq \Delta \tag{4}$$

$$\begin{aligned}[\![\lambda^{\mathrm{b}}\ \{E_i\}_{i=1..m}\ (x_j)_{j=1..n} \to E_0]\!]\iota\xi\Delta\kappa = {} & \text{let } d_i = [\![E_i]\!]\iota\xi\Delta_i\kappa && \Delta_0 \cup \textstyle\bigcup_i \Delta_i \subseteq \Delta \\ & \text{in } \lambda(d_{a_j}).[\![E_0]\!]\iota\big(\xi[x_j/\widehat{d_{a_j}}]\big)\Delta_0(\kappa \lhd \{d_i\})\end{aligned} \tag{5}$$

$$[\![E_0\ .\ (E_i)_{i=1..m}]\!]\iota\xi\Delta\kappa = \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big([\![E_i]\!]\iota\xi\Delta_i\kappa\big) \qquad \Delta_0 \cup \textstyle\bigcup_i \Delta_i \subseteq \Delta \tag{6}$$

$$\begin{aligned}[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\xi\Delta\kappa = {} & \text{let } d_0 = [\![E_0]\!]\iota\xi\Delta_0\kappa && \Delta_0 \cup \Delta_1 \cup \Delta_2 \subseteq \Delta \\ & \text{in } \begin{cases}[\![E_1]\!]\iota\xi\Delta_1\kappa, & d_0 \equiv \mathit{true} \\ [\![E_2]\!]\iota\xi\Delta_2\kappa, & d_0 \equiv \mathit{false}\end{cases}\end{aligned} \tag{7}$$

$$[\![E_0\ \texttt{@}\ E_1]\!]\iota\xi\Delta\kappa = [\![E_0]\!]\iota\xi\Delta_0\big(\kappa \dagger [\![E_1]\!]\iota\xi\Delta_1\kappa\big) \qquad \Delta_0 \cup \Delta_1 \subseteq \Delta \tag{8}$$

$$\begin{aligned}[\![\uparrow\{E_i\}_{i=1..m}\ E_0]\!]\iota\xi\Delta\kappa = {} & \text{let } d_i = [\![E_i]\!]\iota\xi\Delta_i\kappa && \textstyle\bigcup_i \Delta_i \subseteq \Delta \\ & \text{in } \lambda\Delta_a.\lambda\kappa_a.[\![E_0]\!]\iota\xi\Delta_a\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\end{aligned} \tag{9}$$

$$[\![\downarrow E_0]\!]\iota\xi\Delta\kappa = ([\![E_0]\!]\iota\xi\Delta_0\kappa)\Delta_1\kappa \qquad \Delta_0 \cup \Delta_1 \subseteq \Delta \tag{10}$$

$$\begin{aligned}[\![\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \to E_0]\!]\iota\xi\Delta\kappa = {} & \text{let } d_i = [\![E_i]\!]\iota\xi\Delta_i\kappa && \textstyle\bigcup_i \Delta_i \subseteq \Delta \\ & \text{in } \lambda d_a.\lambda\Delta_a.\lambda\kappa_a. \\ & \quad [\![E_0]\!]\iota\big(\xi[x/\widehat{d_a}]\big)\Delta_a\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\end{aligned} \tag{11}$$

$$[\![E_0\ !\ E_1]\!]\iota\xi\Delta\kappa = \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big([\![E_1]\!]\iota\xi\Delta_1\kappa\big)\Delta_2\kappa \qquad \Delta_0 \cup \Delta_1 \cup \Delta_2 \subseteq \Delta \tag{12}$$

$$\begin{aligned}[\![\lambda^{\mathrm{n}}\ \{E_i\}_{i=1..m}\ x \to E_0]\!]\iota\xi\Delta\kappa = {} & \text{let } d_i = [\![E_i]\!]\iota\xi\Delta_i\kappa && \textstyle\bigcup_i \Delta_i \subseteq \Delta \\ & \text{in } \lambda\pi_a.\lambda\Delta_a.\lambda\kappa_a. \\ & \quad [\![E_0]\!]\iota\big(\xi[x/\pi_a]\big)\Delta_a\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\end{aligned} \tag{13}$$

$$[\![E_0\ E_1]\!]\iota\xi\Delta\kappa = \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big([\![E_1]\!]\iota\xi\big)\Delta_1\kappa \qquad \Delta_0 \cup \Delta_1 \subseteq \Delta \tag{14}$$

$$\begin{aligned}[\![E_0\ \texttt{wheredim}\ x_i \leftarrow E_i\ \texttt{end}_{i=1..m}]\!]\iota\xi\Delta\kappa = {} & \text{let } \ d_i \in \Delta,\ i_1 \neq i_2 \Rightarrow d_{i_1} \neq d_{i_2} && \Delta'_0 \cup \textstyle\bigcup_i \Delta'_i \subseteq \Delta' \\ & \quad \Delta' = \Delta - \{d_i\} \\ & \quad \xi_a\ = \xi[x_i/\widehat{d_i}] \\ & \quad \kappa_a = \big\{d_i \mapsto [\![E_i]\!]\iota\xi\Delta'_i\kappa\big\} \\ & \text{in } [\![E_0]\!]\iota\xi_a\Delta'_0(\kappa \dagger \kappa_a) \setminus \{d_i\}\end{aligned} \tag{15}$$

$$\begin{aligned}[\![E_0\ \texttt{wherevar}\ x_i = E_i\ \texttt{end}_{i=1..m}]\!]\iota\xi\Delta\kappa = {} & \text{let } \ \xi_0\ \ \ = \xi[x_i/\perp_{\texttt{play}}] \\ & \quad \xi_{\alpha+1} = \xi_\alpha\big[x_i/[\![E_i]\!]\iota\xi_\alpha\big] \\ & \quad \xi_\sqcup\ \ \ = \mathrm{lfp}\ \xi_\alpha \\ & \text{in } [\![E_0]\!]\iota\xi_\sqcup\Delta\kappa\end{aligned} \tag{16}$$

Figure 3: Denotational semantics of TL expressions

4.5 Semantic rules

Definition 7. *Let $D = \Delta_S \cup \Delta_H$ be an enumerable set of values, where Δ_S ($\supset \{true, false\}$) is a set of calculable values, and Δ_H is an infinite set of hidden dimensions such that $\Delta_S \cap \Delta_H = \emptyset$; Σ be a signature; X be a set of variables; $E \in \mathbf{Expr}(\Sigma, X)$; $\iota \in \mathbf{Interp}(\Sigma, \Delta_S)$; $\xi \in \mathbf{Env}(X, D)$, such that Δ_H is irrelevant to ξ; and κ be a context, such that $\Delta_H \cap \mathrm{dom}\ \kappa = \emptyset$. Then the*

semantics for E with respect to ι, ξ, Δ_H and κ is given by

$$[\![E]\!]\iota\xi\Delta_H\kappa,$$

where the rules for $[\![\cdot]\!]$ are given in Figure 3.

In the rules of Figure 3, if an expression E has several subexpressions E_1, E_2, ..., an infinite set Δ of dimensions is split into several sets Δ_1, Δ_2, ..., each infinite, such that $\Delta_1 \cup \Delta_2 \cup \cdots \subseteq \Delta$ and that $\Delta_i \cap \Delta_j = \emptyset$ for $i \neq j$.

4.5.1 Identifiers

Equation (1): A variable identifier x is looked up in environment ξ, and the resulting intensional playground is applied to Δ and κ to produce a value.

4.5.2 Base functions

Equations (2)–(6): Base functions are functions which are only sensitive to the context upon creation. They take as arguments a tuple, and cannot be curried. There are three ways of creating base functions:

- Equation (2): the interpretation of an m-ary constant symbol, $m > 0$;
- Equation (3): the current context `#` itself;
- Equation (4): the creation of an m-tuple;
- Equation (5): an explicit base function.

Equation (6) applies a base function to its arguments. The base function is determined in the current context, and all of its arguments are computed in the same context; the base function is then applied to those arguments to return a value.

4.5.3 Conditional expressions

Equation (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.

4.5.4 Context perturbation

Equation (8): Expression E_1 is evaluated to a base function, used to perturb the current context κ to produce a new running context for the evaluation of E_2.

4.5.5 Intension abstraction

Equation (9): An intension abstraction freezes, in expression E, the ordinates of the dimensions in κ designated by the E_i expressions. The value returned is an intensional playground. When this value is used in another context κ_a, the body is evaluated with respect to $\kappa_a \dagger (\kappa \lhd \{d_i\})$, i.e., with respect to κ_a, except for the ordinates of the dimensions designated by the E_i, which are fixed at their κ settings.

4.5.6 Intension application

Equation (10): If $\downarrow$ is placed in front of an expression E that evaluates to an intension, then that intension is evaluated in the current context, which is not necessarily the same as the context in which the intension was created.

4.5.7 Call-by-value functions

Equations (11)–(12): The semantics of a function abstraction is similar to that of an intension abstraction, but where the environment is perturbed by $\widehat{\widehat{d_a}}$, an intensional playground generated from the actual parameter d_a, which is fully evaluated before the function body is evaluated.

4.5.8 Call-by-name functions

Equations (13)–(14): In a call-by-name function, the argument is passed by name, i.e., the argument is evaluated dynamically when it is encountered during evaluation of the function body.

4.5.9 Local dimensions

Equation (15): Each dimension identifier x_i is mapped in the new environment $\xi_\sqcup$ to $\widehat{\widehat{d_i}}$, where d_i is an unused dimension in Δ and its ordinate is initially (in κ') the value of expression E_i in context κ. Note that when the d_i are selected from Δ, the choice is made randomly. Note also that the use of the $\setminus$ operator ensures that the d_i cannot be accessed outside of the `wheredim` clause. It follows that any other choice would have led to the same result.

4.5.10 Local variables

Equation (16): The semantics of a `wherevar` clause is given using least fixed points. A sequence of environments ξ_α, $\alpha \in \mathbb{N}$, is defined by initializing $\xi_0 = \xi[x_i/\bot_{\texttt{play}}]_{i=1..m}$, then applying the meaning of the individual equations, mapping variable identifier x_i to the meaning of defining expression E_i to produce $\xi_{\alpha+1}$ from ξ_α. The expression E is then evaluated in the the least-fixed-point environment $\xi_\sqcup$ resulting from the sequence of the ξ_α.

5 Properties

5.1 Soundness

The four propositions below ensure that the semantics given in Figure 3 is sound.

Proposition 2. *Let E be an expression, ι be an interpretation, Δ be a set of hidden dimensions, ξ be an environment and κ and κ' be contexts such that $\kappa \sqsubseteq \kappa'$. Then $[\![E]\!]\iota\xi\Delta\kappa \sqsubseteq [\![E]\!]\iota\xi\Delta\kappa'$.*

Proof. By induction on the structure of E.

Case x: Because of the compatibility constraint on intensions, $\xi(x)(\Delta)(\kappa) \sqsubseteq \xi(x)(\Delta)(\kappa')$.

Case #: $\kappa \sqsubseteq \kappa'$.

The remaining cases use standard inductive arguments. □

Proposition 3. *Let E be an expression, ι be an interpretation, Δ be a set of hidden dimensions, ξ and ξ' be environments such that $\xi \sqsubseteq \xi'$, and κ be a context. Then $[\![E]\!]\iota\xi\Delta\kappa \sqsubseteq [\![E]\!]\iota\xi'\Delta\kappa$.*

Proof. By induction on the structure of E.

Case x: From the definition of $\sqsubseteq$, $\xi(x)(\Delta)(\kappa) \sqsubseteq \xi'(x)(\Delta)(\kappa)$.

The remaining cases use standard inductive arguments. □

Proposition 4. *Let $\alpha \in \mathbb{N}$, and let ξ_α be as in the definition of $[\![E$ `wherevar` $\cdots$ `end`$]\!]$ in Equation (16) of Figure 3. Then $\xi_\alpha \sqsubseteq \xi_{\alpha+1}$.*

Proof. Note that the only differences that might occur between ξ_α and $\xi_{\alpha+1}$ will be for the variable identifiers x_i. We therefore prove by induction on α that $\xi_\alpha(x_i)(\Delta)(\kappa) \sqsubseteq \xi_{\alpha+1}(x_i)(\Delta)(\kappa)$:

Case $\alpha = 0$**:**

$$\begin{aligned}\xi_0(x_i)(\Delta)(\kappa) &= \bot \\ &\sqsubseteq \xi_1(x_i)(\Delta)(\kappa)\end{aligned}$$

Case $\alpha = N > 0$**:** Assume that $\xi_{N-1}(x_i)(\Delta)(\kappa) \sqsubseteq \xi_N(x_i)(\Delta)(\kappa)$.

$$\begin{aligned}\xi_N(x_i)(\Delta)(\kappa) &= [\![E_i]\!]\iota\xi_{N-1}\Delta\kappa \\ &\sqsubseteq [\![E_i]\!]\iota\xi_N\Delta\kappa \\ &\qquad \text{Induction hypothesis and Proposition 3} \\ &= \xi_{N+1}(x_i)(\Delta)(\kappa)\end{aligned}$$

□

Proposition 5. *Let ξ' be as in the definition of $[\![E$ `wherevar` $\cdots$ `end`$]\!]$ in Equation (16) of Figure 3. Then ξ' is in fact a least fixed point.*

Proof. By Proposition 4, we have $\xi_0 \sqsubseteq \cdots \sqsubseteq \xi_\alpha \sqsubseteq \cdots$. Since $\mathbf{D}_{\texttt{play}}$ is a complete partial order, the least fixed point exists, and equals $\xi_\sqcup$ by definition. □

5.2 Intensions and abstractions

With the intension as first-class value, it turns out that call-by-name functions can be simulated with call-by-value functions. Nevertheless, call-by-name functions are useful to the programmer; without these, TL programs would be full of ↑ and ↓.

The following propositions define standard equivalences, the last one holding for all functional programs.

Proposition 6. *Let E be an expression, Δ and Δ' be two infinite sets of dimensions such that $\Delta' \subset \Delta$, ξ be an environment and κ be a context. Then $[\![E]\!]\iota\xi\Delta\kappa = [\![E]\!]\iota\xi\Delta'\kappa$.*

Proof. Since the choice of dimensions in Δ is random, it can always be made within the subset Δ' without affecting the result. □

Proposition 7. *Let E_0 be an expression, Δ be a set of dimensions, ξ be an environment and κ be a context. Then $[\![\downarrow(\uparrow\emptyset\ E_0)]\!]\iota\xi\Delta\kappa = [\![E_0]\!]\iota\xi\Delta\kappa$*

Proof. Let $\Delta_0 \cup \Delta_1 \subseteq \Delta$.

$$\begin{aligned}[\![\downarrow(\uparrow\emptyset\ E_0)]\!]\iota\xi\Delta\kappa &= \big([\![\uparrow\emptyset\ E_0]\!]\iota\xi\Delta_0\kappa\big)\Delta_1\kappa \\ &= \big(\lambda\Delta_a.\lambda\kappa_a.[\![E_0]\!]\iota\xi\Delta_a\kappa_a\big)\Delta_1\kappa \\ &= [\![E_0]\!]\iota\xi\Delta_1\kappa \\ &= [\![E_0]\!]\iota\xi\Delta\kappa\end{aligned}$$

□

Proposition 8. *Let E_0, E_1 be expressions, Δ be a set of dimensions, ξ be an environment and κ be a context. Then $[\![E_0\ E_1]\!]\iota\xi\Delta\kappa = [\![E_0\ !\ (\uparrow\emptyset\ E_1)]\!]\iota\xi\Delta\kappa$*

Proof. Let $\Delta_0 \cup \Delta_1 \cup \Delta_2 \subseteq \Delta$.

$$\begin{aligned}[\![E_0\ E_1]\!]\iota\xi\Delta\kappa &= [\![E_0\ E_1]\!]\iota\xi(\Delta_0 \cup \Delta_2)\kappa \\ &= \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big([\![E_1]\!]\iota\xi\big)\Delta_2\kappa \\ &= \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big(\lambda\Delta_a.\lambda\kappa_a.[\![E_1]\!]\iota\xi\Delta_a\kappa_a\big)\Delta_2\kappa \\ &= \big([\![E_0]\!]\iota\xi\Delta_0\kappa\big)\big([\![\uparrow\emptyset\ E_1]\!]\iota\xi\Delta_1\kappa\big)\Delta_2\kappa \\ &= [\![E_0\ !\ (\uparrow\emptyset\ E_1)]\!]\iota\xi\Delta\kappa\end{aligned}$$

□

Proposition 9. *Let x be an identifier, $E_0, \ldots, E_m$ be expressions, Δ be a set of dimensions, ξ be an environment and κ be a context. Then*

$$\begin{aligned}&\llbracket \lambda^{\mathrm{n}}\ \{E_i\}_{i=1..m}\ x \to E_0 \rrbracket \iota\xi\Delta\kappa \\ =\ &\llbracket \lambda^{\mathrm{v}}\ \{E_i\}\ x \to E_0[x/{\downarrow}x] \rrbracket \iota\xi\Delta\kappa\end{aligned}$$

Proof. Let $\bigcup_i \Delta_i \subseteq \Delta$.

$$\begin{aligned}&\llbracket \lambda^{\mathrm{n}}\ \{E_i\}_{i=1..m}\ x \to E_0 \rrbracket \iota\xi\Delta\kappa \\ =\ &\text{let } d_i = \llbracket E_i \rrbracket \iota\xi\Delta_i\kappa \text{ in} \\ &\lambda\pi_a.\lambda\Delta_a.\lambda\kappa_a. \\ &\llbracket E_0 \rrbracket \iota\big(\xi[x/\pi_a]\big)\Delta_a\big(\kappa_a\ \dagger\ (\kappa \lhd \{d_i\})\big) \\ =\ &\text{let } d_i = \llbracket E_i \rrbracket \iota\xi\Delta_i\kappa \text{ in} \\ &\lambda d_a.\lambda\Delta_a.\lambda\kappa_a. \\ &\llbracket E_0 \rrbracket \iota\big(\xi[x/d_a]\big)\Delta_a\big(\kappa_a\ \dagger\ (\kappa \lhd \{d_i\})\big) \\ =\ &\text{let } d_i = \llbracket E_i \rrbracket \iota\xi\Delta_i\kappa \text{ in} \\ &\lambda d_a.\lambda\Delta_a.\lambda\kappa_a. \\ &\llbracket E_0[x/{\downarrow}x] \rrbracket \iota\big(\xi[x/\widehat{d_a}]\big)\Delta_a\big(\kappa_a\ \dagger\ (\kappa \lhd \{d_i\})\big) \\ =\ &\llbracket \lambda^{\mathrm{v}}\ \{E_i\}\ x \to E_0[x/{\downarrow}x] \rrbracket \iota\xi\Delta\kappa\end{aligned}$$

□

Proposition 10. *Let x be an identifier, E_0, E_1 be expressions, Δ be a set of dimensions, ξ be an environment and κ be a context. Then*

$$\llbracket E_0\ \texttt{wherevar}\ x = E_1\ \texttt{end} \rrbracket \xi\Delta\kappa = \llbracket (\lambda^{\mathrm{n}}\ \emptyset\ x \to E_0)\ E_1 \rrbracket \iota\xi\Delta\kappa$$

Proof. Let $\Delta_0 \cup \Delta_1 \subseteq \Delta$.

$$\begin{aligned}&\llbracket E_0\ \texttt{wherevar}\ x = E_1\ \texttt{end} \rrbracket \iota\xi\Delta\kappa \\ =\ &\text{let } \xi_0 = \xi[x/\bot_{\texttt{play}}] \\ &\quad\ \xi_{\alpha+1} = \xi_\alpha[x/\llbracket E_1 \rrbracket \xi_\alpha] \\ &\quad\ \xi_\sqcup = \mathrm{lfp}\ \xi_\alpha \\ &\text{in } \llbracket E_0 \rrbracket \iota\xi_\sqcup\Delta\kappa \\ =\ &\llbracket E_0 \rrbracket \iota\big(\xi[x/\llbracket E_1 \rrbracket \iota\xi]\big)\Delta\kappa \\ =\ &\llbracket E_0 \rrbracket \iota\big(\xi[x/\llbracket E_1 \rrbracket \iota\xi]\big)\Delta_1\kappa \\ =\ &\big(\lambda\pi_a.\lambda\Delta_a.\lambda\kappa_a.\llbracket E_0 \rrbracket \iota(\xi[x/\pi_a])\Delta_a\kappa_a\big)\big(\llbracket E_1 \rrbracket \iota\xi\big)\Delta_1\kappa \\ =\ &\big(\llbracket \lambda^{\mathrm{n}}\ \emptyset\ x \to E_0 \rrbracket \iota\xi\Delta_0\kappa\big)\big(\llbracket E_1 \rrbracket \iota\xi\big)\Delta_1\kappa \\ =\ &\llbracket (\lambda^{\mathrm{n}}\ \emptyset\ x \to E_0)\ E_1 \rrbracket \iota\xi\Delta\kappa\end{aligned}$$

□

6 The contextual semantics

The denotational semantic rules $\llbracket\cdot\rrbracket$ presented in Figure 3 are not effective, as the choice of hidden dimensions from the parameter Δ is made in a random manner. Furthermore, the semantics requires the manipulation of both the environment and the context, despite the fact that the manipulation of the two resemble each other. In this section, we show how this semantics can be transformed into an effective one, leading to a natural implementation.

6.1 Deterministic choice of dimensions

The choice of dimensions in the denotational semantics is defined nondeterministically; however, this choice can be made deterministic with a simple set of adjustments. We begin by defining Δ_H to be the set

$$\Delta_H = \{\chi^i_\nu \mid \nu \in \mathbb{N}^*, i \in \mathbb{N}\}.$$

The idea is that the list of integers ν encodes unambiguously the position in the evaluation tree of an expression, and that the infinite set Δ_H can be subdivided as many times as needed.

We continue by defining subsets of Δ_H of the form ${}^\nu\mu$, where $\nu \in \mathbb{N}^*$. The set ${}^\nu\mu$ contains the hidden dimensions that can be found once position ν has been reached in the evaluation tree. It is given by:

$${}^\nu\mu = \{\chi^i_{\nu\cdot\nu'} \mid \nu' \in \mathbb{N}^*, i \in \mathbb{N}\},$$

where $\nu \cdot \nu'$ is the concatenation of lists ν and ν'.

The semantics is adjusted as follows. When the value for Δ is ${}^\nu\mu$, then we let:

$$\Delta' = \{\chi^i_{\nu\cdot\nu'} \mid \nu' \in \mathbb{N}^+, i \in \mathbb{N}\} \tag{17}$$

$$\Delta_j = \{\chi^i_{\nu:j\cdot\nu'} \mid \nu' \in \mathbb{N}^+, i \in \mathbb{N}\}, \tag{18}$$

where $\nu : j$ is the appending of list ν by integer j. As for the selection of the d_i in equation (15), we simply let $d_i = \chi^i_\nu$. Finally, we note that $\Delta_H = {}^\epsilon\mu$, where ϵ is the empty list.

Since the setting of ν is different for every subexpression, it follows that the dimensions allocated for the dimension identifiers when two `wheredim` clauses are encountered in different subexpressions will be distinct. Hence, with these adjustments, the semantic rules of Figure 3 become deterministic.

6.2 Using the context

In the denotational semantic rules given in Figure 3, the construct $\widehat{\overline{d}}$, where

$$\widehat{\overline{d}} = \lambda\Delta.\lambda\kappa.d,$$

appears four times: for $\uparrow$ (Equation 9), λ^{b} (Equation 5), λ^{v} (Equation 11) and `wheredim` (Equation 15).

In each of these four situations, a value d must be transformed into a constant intensional playground whose range is always a constant intension, so that the playground can be added into the environment as the value of some identifier. Although technically correct, this does not naturally lead to an efficient implementation.

In the *contextual semantics* $[\![\cdot]\!]^\circ$ given in this section, the explicit manipulation of the environment in the denotational semantics for these situations is replaced by explicit manipulation of the context with appropriate additional hidden dimensions, so that there is no need to create objects of the form $\widehat{\overline{d}}$.

In order for this contextual semantics to work, a TL expression E must be rewritten into a TL$^\circ$ expression E° with a different syntax. We suppose here that E is of the following form:

- each identifier in E that is not a free variable is declared exactly once, be it a dimension identifier, a variable identifier, or a formal parameter; this can be done simply by renaming all identifiers as needed;
- the expression E does not contain any call-by-name abstractions or applications; these can be rewritten into call-by-value and intension abstractions using Propositions 8 and 9.

$$
\begin{array}{rll}
E ::= & d & \textit{constant value} \\
| & \underline{d} & \textit{constant dimension} \\
| & x & \textit{identifier} \\
| & {}^{m}c & \textit{m-ary constant symbol, } m \in \mathbb{N} \\
| & \texttt{\#} & \textit{context} \\
| & [E \leftarrow E,\ \ldots] & \textit{tuple builder} \\
| & \lambda^{\mathrm{b}}_{\circ}\ \{E, \ldots\}\ (\phi_x, \ldots) \rightarrow E & \textit{base abstraction} \\
| & E\,.\,(E,\ \ldots) & \textit{base application} \\
| & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E\ \texttt{fi} & \textit{conditional} \\
| & E\ \texttt{@}\ E & \textit{context perturbation} \\
| & \uparrow\{E, \ldots\}\ E & \textit{intension abstraction} \\
| & \downarrow E & \textit{intension application} \\
| & \lambda^{\mathrm{v}}_{\circ}\ \{E, \ldots\}\ \phi_x \rightarrow E & \textit{call-by-value abstraction} \\
| & E\,!\,E & \textit{call-by-value application} \\
| & E\ \texttt{wheredim}^{\circ}\ \phi_x \leftarrow E,\ \ldots\ \texttt{end} & \textit{local dimensions} \\
| & E\ \texttt{wherevar}\ x = E,\ \ldots\ \texttt{end} & \textit{local variables}
\end{array}
$$

Figure 4: Syntax of TL$^\circ$ expressions

The rewriting of expression E into E° is done by a transformation $\mathcal{T}$, which replaces each λ^{b}-abstraction by a $\lambda^{\mathrm{b}}_{\circ}$-abstraction, each λ^{v}-abstraction by a $\lambda^{\mathrm{v}}_{\circ}$-abstraction, and each `wheredim` clause by a `wheredim`$^\circ$ clause, in which every dimension identifier or formal parameter x is replaced by dimension ϕ_x.

The contextual semantics $[\![\cdot]\!]^\circ$ is defined over the transformed expression. If we look at Equations (5), (11) and (15) from Figure 3, respectively for λ^{b}, λ^{v} and `wheredim`, we see that in all three situations, the body E_0 is evaluated with respect to a modification of the abstraction environment ξ. Therefore, when this information is moved to the context κ, then all of the abstraction objects created during the evaluation of the body E_0, namely for $\uparrow$, λ^{b} and λ^{v} subexpressions, the new ϕ_x dimensions must be added to the set of dimensions saved from the creation context κ as opposed to the evaluation context κ_a. As a result, the only manipulations of the environment take place in `wherevar` clauses.

Finally, the set Δ no longer needs to be passed around in the semantics. Since each Δ is of the form ${}^{\nu}\mu$, only the ν needs to be passed around. In the contextual semantics, the parameter ν becomes the ordinate for dimension ρ in the context κ.

6.3 Syntax

We begin with the new syntax.

Definition 8. *Let Σ be a signature and X ($\ni x$) be a set of identifiers. Then* **Expr**$^\circ$ *($\ni E$) is the set of TL$^\circ$ expressions over Σ and X. The* free variables *of E are written $FV(E)$. If $x \notin FV(E')$, then a substitution of E' for the variable x in E is written $E[x/E']$. The* abstract syntax *for TL$^\circ$ expressions is given in Figure 4.*

Note that the new syntax includes entries for d and $\underline{d}$, where $d \in D$. For both, the d refers to a dimension that will be introduced with the transformation. Entry d evaluates to itself, without passing through the interpretation ι, while entry $\underline{d}$ evaluates to $\kappa(d)$, the ordinate of d in the current context.

6.4 Transformation

We define transformation $\mathcal{T}(E, \Delta)$, where $E \in$ **Expr** is an expression and Δ is the set of hidden dimensions allocated from all of the λ^{v}-abstractions and the dimension identifiers defined in `wheredim` clauses for which E is a subexpression; the result is an expression $E^\circ \in$ **Expr**$^\circ$.

Should E be the outermost expression, the transformation is $\mathcal{T}(E, \emptyset)$, as there are no surrounding λ^{v}-abstractions or `wheredim` clauses.

The transformation $\mathcal{T}$ recursively traverses the structure of E. For most subexpressions, the definition for $\mathcal{T}$ is straightforward. There are three interesting cases: `wheredim` clauses, intension and function abstractions.

For `wheredim` clauses, suppose that we have the expression

$$E = (E_0 \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}) \tag{19}$$

Then

$$\mathcal{T}(E, \Delta) = \begin{pmatrix} \mathcal{T}\big(E_0[x_i/\underline{\phi_{x_i}}], \Delta \cup \{\phi_{x_i}\}\big) \\ \texttt{wheredim}^{\circ}\ \phi_{x_i} \leftarrow \mathcal{T}(E_i, \Delta)\ \texttt{end} \end{pmatrix} \tag{20}$$

The dimension identifiers (x_i) in expression E_0 in the original clause are mapped to dimension queries $(\underline{\phi_{x_i}})$ after the transformation. Each query looks up in the current context the actual χ^i_ν dimension that was allocated on this entry into this `wheredim` clause. By mapping x_i to $\underline{\phi_{x_i}}$ instead of $\#.\phi_{x_i}$, the evaluation trees before and after the transformation have the same structure, thereby simplifying the proof of their equivalence.

For intension abstractions, suppose that we have the expression

$$E = \big(\uparrow\{E_i\}_{i=1..m}\ E_0\big) \tag{21}$$

Then

$$\mathcal{T}(E, \Delta) = \Big(\uparrow\big(\Delta \cup \big\{\mathcal{T}(E_i, \Delta)\big\}\big)\ \mathcal{T}(E_0, \Delta)\Big) \tag{22}$$

Each hidden dimension in Δ passed down the transformation is added to the set of dimensions that are frozen in the intensions.

When the transformation is being applied to an intension, if the set Δ is nonempty, then that intension is within the body of one or more λ^{v}-abstractions or `wheredim` clauses, which means that the body will already have had the formal parameters of those abstractions and `wheredim` clauses substituted with dimension queries. It follows that the ordinates of those dimensions have to be added to the set of frozen dimensions.

For base abstractions, suppose that we have the expression

$$E = \big(\lambda^{\mathrm{b}}\ \{E_i\}_{i=1..m}\ (x_j)_{j=1..n} \rightarrow E_0\big) \tag{23}$$

Then

$$\mathcal{T}(E, \Delta) = \begin{pmatrix} \lambda^{\mathrm{b}}_{\circ}\ \big(\Delta \cup \{\rho\} \cup \big\{\mathcal{T}(E_i, \Delta)\big\}\big)\ (\phi_{x_j}) \\ \rightarrow \mathcal{T}\big(E_0[x_j/\underline{\phi_{x_j}}],\ \Delta \cup \{\phi_{x_j}\}\big) \end{pmatrix} \tag{24}$$

The formal parameter (x) in expression E_0 in the original abstraction becomes a dimension query $(\underline{\phi_x})$ after the transformation. As for intensions, each hidden dimension in Δ passed down through the transformation is added to the set of dimensions that are frozen. Finally, since the body of a base abstraction is not sensitive to the context upon application, the ρ dimension must be saved.

A similar transformation is used for call-by-value abstractions, except that dimension ρ is not saved. Suppose that we have the expression:

$$E = \big(\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \rightarrow E_0\big) \tag{25}$$

Then

$$\mathcal{T}(E, \Delta) = \begin{pmatrix} \lambda^{\mathrm{v}}_{\circ}\ \big(\Delta \cup \big\{\mathcal{T}(E_i, \Delta)\big\}\big)\ \phi_x \\ \rightarrow \mathcal{T}\big(E_0[x/\underline{\phi_x}],\ \Delta \cup \{\phi_x\}\big) \end{pmatrix} \tag{26}$$

6.5 Domains

Because the $[\![\cdot]\!]^\circ$ semantics does not use intensional playgrounds, we must define new domains.

Definition 9. *Let D be an enumerable set of values. The* semantic domain $\mathbf{D}^\circ$ derived from D *is the least solution to the equations*

$$\begin{aligned}
\mathbf{D}^\circ &= D \cup \left(\bigcup_{m>0} \mathbf{D}^\circ_{\mathtt{base},m}\right) \cup \mathbf{D}^\circ_{\mathtt{intens}} \cup \mathbf{D}^\circ_{\mathtt{value}} \\
\mathbf{D}^\circ_{\mathtt{base},m} &= D^m \rightarrowtail \mathbf{D}^\circ, \qquad \text{for } m \in \mathbb{N} - \{0\} \\
\mathbf{D}^\circ_{\mathtt{ctxt}} &= \mathbf{D}^\circ_{\mathtt{base},1} \\
\mathbf{D}^\circ_{\mathtt{intens}} &= \mathbf{D}^\circ_{\mathtt{ctxt}} \rightarrowtail \mathbf{D}^\circ \\
\mathbf{D}^\circ_{\mathtt{value}} &= \mathbf{D}^\circ \rightarrowtail \mathbf{D}^\circ_{\mathtt{intens}}
\end{aligned}$$

Definition 10. *Let D be an enumerable set of values, $\mathbf{D}^\circ$ be the semantic domain derived from D, and $\bot \notin \mathbf{D}^\circ$. Then we define the order $\sqsubseteq^\circ$ over $\mathbf{D}^\circ_\bot = \mathbf{D}^\circ \cup \{\bot\}$ by:*

- *For all $d \in \mathbf{D}^\circ_\bot$, $\bot \sqsubseteq^\circ d$.*
- *For all $d \in D$, $d \sqsubseteq^\circ d$.*
- *For all $f, f' \in \mathbf{D}^\circ_{\mathtt{base},m}$, $f \sqsubseteq^\circ f'$ iff $f = f' \lhd (\mathrm{dom}\ f)$.*
- *For all $\eta, \eta' \in \mathbf{D}^\circ_{\mathtt{intens}}$, $\eta \sqsubseteq^\circ \eta'$ iff $\eta = \eta' \lhd (\mathrm{dom}\ \eta)$.*
- *For all $f, f' \in \mathbf{D}^\circ_{\mathtt{value}}$, $f \sqsubseteq^\circ f'$ iff $f = f' \lhd (\mathrm{dom}\ f)$.*

We write

- *$\bot_{\mathtt{base},m}$ for the least element of $\mathbf{D}^\circ_{\mathtt{base},m}$, with empty domain;*
- *$\bot_{\mathtt{intens}}$ for the least element of $\mathbf{D}^\circ_{\mathtt{intens}}$, with empty domain;*
- *$\bot_{\mathtt{value}}$ for the least element of $\mathbf{D}^\circ_{\mathtt{value}}$, with empty domain.*

Proposition 11. *The pair $(\mathbf{D}^\circ_\bot, \sqsubseteq^\circ)$ is a complete partial order, such that the following are also cpos:*

1. *$(D_\bot, \sqsubseteq^\circ)$, where $D_\bot = D \cup \{\bot\}$;*
2. *$(\mathbf{D}^\circ_{\mathtt{base},m}, \sqsubseteq^\circ)$, $m \in \mathbb{N} - \{0\}$;*
3. *$(\mathbf{D}^\circ_{\mathtt{intens}}, \sqsubseteq^\circ)$;*
4. *$(\mathbf{D}^\circ_{\mathtt{value}}, \sqsubseteq^\circ)$.*

Proof. Analogous to the proof of Proposition 1. □

Definition 11. *Let D be an enumerable set of values and X be a set of variables. Then a* contextual environment *over X and D is a mapping $\zeta : X \rightarrowtail \mathbf{D}^\circ_{\mathtt{intens}}$. A substitution of κ for the value of x in ζ is written $\zeta[x/\kappa]$. The perturbation of ζ by ζ' is written $\zeta \dagger \zeta'$. We extend the order $\sqsubseteq^\circ$ to contextual environments: $\zeta \sqsubseteq^\circ \zeta'$ iff $\forall x \in \mathrm{dom}\ \zeta \cup \mathrm{dom}\ \zeta'$, $\zeta(x) \sqsubseteq^\circ \zeta'(x)$. A set $\Delta \subset D$ is* irrelevant to ζ *iff $\forall x \in \mathrm{dom}\ \zeta, \forall \kappa \in \mathbf{D}_{\mathtt{ctxt}}$, $\xi(x)(\kappa) = \xi(x)(\kappa \lhd\!\lhd \Delta)$. We write $\mathbf{Env}^\circ(X, D)$ for the set of contextual environments over X and D.*

6.6 Notation

- Let $n \in \mathbb{N}$ and κ be a context. Then

$$_n\kappa = \kappa \dagger \left\{\rho \mapsto \kappa(\rho) : n\right\}$$

- The masking of a value d by a set Δ, written $d \setminus \Delta$, is defined in Figure 5.

$$
\begin{aligned}
d \setminus \Delta &= d, \qquad d \notin \Delta \\
d \setminus \Delta &= \bot, \qquad d \in \Delta \\
\kappa \setminus \Delta &= \big\{ d_i \mapsto \kappa(d_i) \setminus \Delta \mid d_i \in \mathrm{dom}\,(\kappa \lhd \Delta) \big\} \\
\big(\lambda(d_{a_j}).f(d_{a_j})\big) \setminus \Delta &= \lambda(d_{a_j}).\big(f(d_{a_j}) \setminus \Delta\big) \\
\big(\lambda\kappa_a.f(\kappa_a)\big) \setminus \Delta &= \lambda\kappa_a.\big(f(\kappa_a) \setminus \Delta\big) \\
\big(\lambda d_a.\lambda\kappa_a.f(d_a,\kappa_a)\big) \setminus \Delta &= \lambda d_a.\lambda\kappa_a.\big(f(d_a,\kappa_a) \setminus \Delta\big)
\end{aligned}
$$

Figure 5: Definition of $\setminus$ for contextual semantics

$$
\begin{aligned}
[\![d]\!]^\circ \iota\zeta\kappa &= d && (27)\\
[\![\underline{d}]\!]^\circ \iota\zeta\kappa &= \kappa(d) && (28)\\
[\![x]\!]^\circ \iota\zeta\kappa &= \zeta(x)(\kappa) && (29)\\
[\![{}^m c]\!]^\circ \iota\zeta\kappa &= \iota({}^m c) && (30)\\
[\![\mathtt{\#}]\!]^\circ \iota\zeta\kappa &= \kappa && (31)\\
[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]^\circ \iota\zeta\kappa &= \big\{ [\![E_{i0}]\!]^\circ \iota\zeta({}_{(i*2)}\kappa) \;\mapsto\; [\![E_{i1}]\!]^\circ \iota\zeta({}_{(i*2+1)}\kappa) \big\} && (32)\\
[\![\lambda^{\mathrm{b}}_\circ \{E_i\}_{i=1..m}\ (\phi_{x_j})_{j=1..n} \to E_0]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ d_i = [\![E_i]\!]^\circ \iota\zeta({}_i\kappa) && (33)\\
&\quad \mathrm{in}\ \lambda(d_{a_j}).[\![E_0]\!]^\circ \iota\zeta\big(({}_0\kappa \lhd \{\rho, d_i\}) \dagger \{\phi_{x_j} \mapsto d_{a_j}\}\big) \\
[\![E_0\ .\ (E_i)_{i=1..m}]\!]^\circ \iota\zeta\kappa &= \big([\![E_0]\!]^\circ \iota\zeta({}_0\kappa)\big)\big([\![E_i]\!]^\circ \iota\zeta({}_i\kappa)\big) && (34)\\
[\![\mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2\ \mathtt{fi}]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ d_0 = [\![E_1]\!]^\circ \iota\zeta({}_0\kappa) && (35)\\
&\quad \mathrm{in} \begin{cases} [\![E_1]\!]^\circ \iota\zeta({}_1\kappa), & d_0 \equiv \mathit{true} \\ [\![E_2]\!]^\circ \iota\zeta({}_2\kappa), & d_0 \equiv \mathit{false} \end{cases} \\
[\![E_0\ \mathtt{@}\ E_1]\!]^\circ \iota\zeta\kappa &= [\![E_0]\!]^\circ \iota\zeta\big(({}_0\kappa) \dagger [\![E_1]\!]^\circ \zeta({}_1\kappa)\big) && (36)\\
[\![\uparrow\{E_i\}_{i=1..m}\ E_0]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ d_i = [\![E_i]\!]^\circ \iota\zeta({}_i\kappa) && (37)\\
&\quad \mathrm{in}\ \lambda\kappa_a.[\![E_0]\!]^\circ \iota\zeta\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
[\![\downarrow E_0]\!]^\circ \iota\zeta\kappa &= \big([\![E_0]\!]^\circ \iota\zeta({}_0\kappa)\big)({}_1\kappa) && (38)\\
[\![\lambda^{\mathrm{v}}_\circ \{E_i\}_{i=1..m}\ \phi_x \to E_0]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ d_i = [\![E_i]\!]^\circ \iota\zeta({}_i\kappa) && (39)\\
&\quad \mathrm{in}\ \lambda d_a.\lambda\kappa_a.[\![E_0]\!]^\circ \iota\zeta\big(\kappa_a \dagger (\kappa \lhd \{d_i\}) \dagger \{\phi_x \mapsto d_a\}\big) \\
[\![E_0\ !\ E_1]\!]^\circ \iota\zeta\kappa &= \big([\![E_0]\!]^\circ \iota\zeta({}_0\kappa)\big)\big([\![E_1]\!]^\circ \iota\zeta({}_1\kappa)\big)({}_2\kappa) && (40)\\
[\![E_0\ \mathtt{wheredim}^\circ\ \phi_{x_i} \leftarrow E_i\ \mathtt{end}_{i=1..m}]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ \ d_i = \chi^i_{\kappa(\rho)} && (41)\\
&\qquad\quad d'_i = [\![E_i]\!]^\circ \iota\zeta({}_i\kappa) \\
&\quad \mathrm{in}\ [\![E_0]\!]^\circ \iota\zeta\big(({}_0\kappa) \dagger \{\phi_{x_i} \mapsto d_i,\ d_i \mapsto d'_i\}\big) \setminus \{d_i\} \\
[\![E_0\ \mathtt{wherevar}\ x_i = E_i\ \mathtt{end}_{i=1..m}]\!]^\circ \iota\zeta\kappa &= \mathrm{let}\ \ \zeta_0 \quad = \zeta[x_i/\bot_{\mathtt{intens}}] && (42)\\
&\qquad\quad \zeta_{\alpha+1} = \zeta_\alpha\big[x_i/[\![E_i]\!]^\circ \iota\zeta_\alpha\big] \\
&\qquad\quad \zeta_\sqcup \quad = \mathrm{lfp}\ \zeta_\alpha \\
&\quad \mathrm{in}\ [\![E_0]\!]^\circ \iota\zeta_\sqcup\kappa
\end{aligned}
$$

Figure 6: Contextual semantics of TL expressions

6.7 Semantic rules

Definition 12. *Let X be a set of variables; $D = \Delta_S \cup \Delta_H^\circ$ be an enumerable set of values, where Δ_S ($\supset \{true, false\}$) is a set of calculable values, and*

$$\Delta_H^\circ = \{\chi_\nu^i \mid \nu \in \mathbb{N}^*, i \in \mathbb{N}\} \cup \{\rho\} \cup \{\phi_x \mid x \in X\}$$

is a set of hidden dimensions such that $\Delta_S \cap \Delta_H^\circ = \emptyset$; Σ be a signature; $E \in \mathbf{Expr}^\circ(\Sigma, X)$; $\iota \in \mathbf{Interp}(\Sigma, \Delta_S)$; $\zeta \in \mathbf{Env}^\circ(X, D)$, such that Δ_H° is irrelevant to ζ; and κ be a context. Then the contextual semantics for E with respect to ι, ζ and κ is given by

$$[\![E]\!]^\circ \iota \zeta \big(\kappa \dagger \{\rho \mapsto \epsilon\}\big),$$

where the rules for $[\![\cdot]\!]^\circ$ are given in Figure 6.

6.8 Validity of semantics

In order to prove the validity between the denotational and deterministic semantics, we need to define an equivalence $\approx$ between the objects created by the two semantics.

- All simple values are the same in the two semantics:

$$d \approx d, \quad d \in \Delta_S \cup \Delta_H.$$

- The base abstractions correspond in the two semantics:

$$\begin{aligned} & \lambda(d_{a_j}).[\![E_0]\!]\iota(\xi[x_j/\widehat{d_{a_j}}])\Delta_0(\kappa \lhd \{d_i\}) \\ \approx\ & \lambda(d_{a_j}).[\![E_0^\circ]\!]^\circ \iota\zeta((_0\kappa^\circ \lhd \{d_i\}) \dagger \{\phi_{x_j} \mapsto d_{a_j}\}) \\ & \text{iff } (\xi, \Delta, \kappa) \approx (\zeta, \kappa^\circ). \end{aligned}$$

- The intension abstractions correspond in the two semantics:

$$\begin{aligned} & \lambda\Delta_a.\lambda\kappa_a.[\![E_0]\!]\iota(\xi[x/\widehat{d_a}])\Delta_a(\kappa_a \dagger (\kappa \lhd \{d_i\})) \\ \approx\ & \lambda\kappa_a.[\![E_0^\circ]\!]^\circ \iota\zeta(\kappa_a \dagger (\kappa^\circ \lhd \{d_i\})) \\ & \text{iff } \exists\Delta\ (\xi, \Delta, \kappa) \approx (\zeta, \kappa^\circ). \end{aligned}$$

- The call-by-value abstractions correspond in the two semantics:

$$\begin{aligned} & \lambda d_a.\lambda\Delta_a.\lambda\kappa_a.[\![E_0]\!]\iota(\xi[x/\widehat{d_a}])\Delta_a(\kappa_a \dagger (\kappa \lhd \{d_i\})) \\ \approx\ & \lambda d_a.\lambda\kappa_a.[\![E_0^\circ]\!]^\circ \iota\zeta(\kappa_a \dagger (\kappa^\circ \lhd \{d_i\}) \dagger \{\phi_x \mapsto d_a\}) \\ & \text{iff } \exists\Delta\ (\xi, \Delta, \kappa) \approx (\zeta, \kappa^\circ). \end{aligned}$$

- The contexts correspond in the two semantics:

$$\kappa \approx \kappa^\circ$$

  iff $\mathrm{dom}\ \kappa = \mathrm{dom}\ \kappa^\circ \lhd \Delta_H$ and

$$\forall d_i \in \mathrm{dom}\ \kappa,\ \kappa(d_i) \approx \kappa^\circ(d_i).$$

- The respective parameters of $[\![\cdot]\!]$ and $[\![\cdot]\!]^\circ$ correspond in the two semantics:

$$(\xi, \Delta, \kappa) \approx (\zeta, \kappa^\circ)$$

  iff $\exists\nu \in \mathbb{N}^*$ such that $\Delta = {}^\nu\mu$ and

$$\begin{aligned} \nu &= \kappa^\circ(\rho) \\ \kappa &\approx \kappa^\circ \lhd \Delta_H \\ \xi(x)(\Delta)(\kappa) &\approx \zeta(x)(\kappa^\circ), && x \text{ is a variable identifier} \\ \xi(x)(\Delta)(\kappa) &\approx \kappa^\circ(\phi_x), && x \text{ is another identifier.} \end{aligned}$$

Proposition 12. *Let X be a set of variables; $D = \Delta_S \cup \Delta_H$ and $D^\circ = \Delta_S \cup \Delta_H^\circ$ be two enumerable sets of values, where Δ_S ($\supset \{true, false\}$) is a set of calculable values, $\Delta_H = \{\chi_\nu^i \mid i \in \mathbb{N}, \nu \in \mathbb{N}^*\}$ and $\Delta_H^\circ = \Delta_H \cup \{\rho\} \cup \{\phi_x \mid x \in X\}$ are two sets of hidden dimensions such that $\Delta_S \cap \Delta_H^\circ = \emptyset$; Σ be a signature; $E \in \mathbf{Expr}(\Sigma, X)$; $\iota \in \mathbf{Interp}(\Sigma, \Delta_S)$; $\xi \in \mathbf{Env}(X, D)$; $\zeta \in \mathbf{Env}^\circ(X, D^\circ)$, such that $\forall x \in FV(E)$, $\forall \Delta \forall \kappa, \xi(x)(\Delta)(\kappa) = \zeta(x)(\kappa)$; and κ be a context. Then*

$$[\![E]\!]\iota\xi\Delta_H\kappa \approx [\![\mathcal{T}(E, \emptyset)]\!]^\circ\iota\zeta\big(\kappa \dagger \{\rho \mapsto \epsilon\}\big).$$

Proof. For a given subexpression E_s of E, the evaluation using the denotational semantics at a specific point will be of the form $[\![E_s]\!]\iota\xi_s\Delta_s\kappa_s$, while the corresponding evaluation using the deterministic semantics will be $[\![E_s^\circ]\!]^\circ\iota\zeta_s\kappa_s^\circ$. The following invariants hold:

$$(\xi_s, \Delta_s, \kappa_s) \approx (\zeta_s, \kappa_s^\circ)$$
$$[\![E_s]\!]\iota\xi_s\Delta_s\kappa_s \approx [\![E_s^\circ]\!]^\circ\iota\zeta_s\kappa_s^\circ.$$

The proof of the invariants is by induction on the depth of the evaluation tree of E, i.e., on $n = \text{len}(\nu_s)$. For case $n = 0$, we must examine equations (1), (2), (3) and (16) from Figure 3 and the corresponding equations (29), (30), (31) and (42) from Figure 6. For $n = N + 1$, the remaining corresponding pairs must be examined.

The main result follows directly from the last invariant. □

7 Collapsing of `where` clauses

In the $[\![\cdot]\!]^\circ$ semantics, the only manipulation of the environment takes place in `wherevar` clauses. Here, we show that an expression with no local dimension identifiers can be collapsed so that there is only one `wherevar` clause.

Proposition 13. *Let E be the expression*

$$\begin{array}{l} E_0 \texttt{ wherevar} \\ \quad x_i = E_i \\ \quad x_m = E_m \texttt{ wherevar} \\ \qquad\qquad x_j = E_j \\ \qquad\qquad \texttt{end}_{j=m+1..m+n} \\ \texttt{end}_{i=1..m-1} \end{array}$$

with a `wherevar` *clause with an inner* `wherevar` *clause, and all variables are bound only once; and let E' be the expression*

$$\begin{array}{l} E_0 \texttt{ wherevar} \\ \quad x_i = E_i \\ \texttt{end}_{i=1..m+n} \end{array}$$

Let ι be an interpretation, ζ be an environment and κ be a context. Then

$$[\![E]\!]^\circ\iota\zeta\kappa = [\![E']\!]^\circ\iota\zeta\kappa.$$

Proof. The semantics of E requires the computation of the fixed point of the semantics of the outer `wherevar` clause; at each iteration of that clause, each time that x_m is computed, the fixed point of the inner `wherevar` clause must be computed. For E', on the other hand, there is only one `wherevar` clause, hence the computation of only one fixed point is needed. Proving the validity of this proposition requires demonstrating the equivalence of the fixed points.

Because of the way that the order is defined, should the fixed point $\zeta_\sqcup(x)(\kappa)$ be defined in a `wherevar` clause, then there will exist a finite α such that $\zeta_\alpha(x)(\kappa)$ will yield the same result. This leads naturally to the following invariant:

If variable x_i, $i \in 1..m$, evaluates in context κ in iteration α' in E', then that same variable will evaluate in context κ in some combination $(\alpha; \alpha_0, \ldots, \alpha_\alpha)$ of iterations in E, where

$$\alpha' = \alpha + \sum_{\beta=0..\alpha} \alpha_\beta$$

meaning that α iterations of the outer `wherevar` clause have been applied, with, for each iteration β of the outer clause, α_β iterations of the inner clause.

The proof is by induction over α'.

Case $\alpha' = 0$**:** The variable x_i evaluates in context κ without referring to any other variables. The same would hold true in E, i.e., $\alpha = 0$ and $\alpha_0 = 0$.

Case $\alpha' = A + 1$**:** The variable x_i evaluates in context κ in $A+1$ steps, which means it depends on some other variable x in context κ' that evaluates in A steps. The induction hypothesis tells us that x evaluates in context κ' in E in $(\alpha; \alpha_0, \ldots, \alpha_\alpha)$ iterations, where $A = \alpha+\alpha_0+\cdots+\alpha_\alpha$. Then computing variable x_i in context κ will either require an additional iteration of the outer or of the inner clause. In the former case, this occurs in $(\alpha+1; \alpha_0, \ldots, \alpha_\alpha, 0)$ iterations, while in the latter case, it occurs in $(\alpha; \alpha_0, \ldots, \alpha_\alpha+1)$ iterations. In both cases the invariant is maintained.

It follows that the fixed points computed by E and E' yield the same results. □

Corollary 14. *Let E be an expression. If there are no local dimension identifiers, then E can be rewritten into an equivalent expression E' in which there is a single* `wherevar` *clause.*

Proof. By induction on the structure of E, using Proposition 13 for individual steps. □

8 Conclusions

In this paper, we have presented the TL programming language, its denotational semantics, and a means for transforming function applications so that these can be implemented without closures over the environment. The significance of these results is manyfold.

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 three key longstanding problems:

1. design, semantics and implementation of higher-order functions over Lucid streams [1], 1977;
2. design, semantics and implementation of the hypothetical Lambda Lucid, with streams of functions [13], 1986;
3. denotational semantics of Indexical Lucid [2], 1995;
4. full indexical implementation of higher-order functions [11], 1999.

The solutions to the above problems required a recognition that the dualities of call-by-value *vs.* call-by-name and of lexical binding *vs.* dynamic binding are not either-or choices. All of these are used, meaningfully and necessarily, in the TL language.

Finally, we should note that if a TL program makes no explicit use of dimensions or of the runtime context, i.e., 1) free variables do not vary in multiple dimensions, 2) `where` clauses do not define dimension identifiers, and 3) operators `#` and `@` are not used, then that program is an ordinary functional program. This means that the implementation technique presented here for higher-order functions is applicable to *all* functional languages, call-by-value or call-by-name, or both.

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

[4] 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.

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

[6] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In Mark N. Wegman and Thomas W. Reps, editors, *POPL*, pages 108–118. ACM, 2000.

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

[14] Philip Wadler. Lazy versus strict. *ACM Comput. Surv.*, 28(2):318–320, 1996.

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