TransLucid: From theory to implementation

Jarryd P. Beck

A thesis in fulfilment of the requirements for the degree of

Doctor of Philosophy

The School of Computer Science and Engineering

The Faculty of Engineering

March 2015

Abstract

This thesis presents the programming language TransLucid, from its denotational semantics to its concrete implementation. In TransLucid, a variable denotes an intension, which is an array of arbitrary rank and infinite extent, indexed by a multidimensional context. TransLucid is descended from Lucid, whose development since 1974 left several open problems, all of which are solved in this thesis. These open problems are: 1) the semantics and implementation of higher-order functions over intensions; 2) the semantics and implementation of dimensions as first-class values, atomic values as dimensions, and contexts as first-class values; and 3) the implementation of a cache-based evaluator. In addition, this thesis presents a type inference algorithm for TransLucid, and the concrete TransLucid system, which is a synchronous reactive programming environment.

Acknowledgements

I would like to take this chance to thank all of the people who have supported me whilst I undertook the research described in this thesis. Firstly, I would like to thank The School of Computer Science and Engineering at The University of New South Wales for the use of their resources, and Manuel Chakravarty and Éric Martin for sitting on my annual review panel each year. Secondly, thanks to my friends and family who continually encouraged me, and celebrated with me in the achievements I made along the way.

Thank you to my wife, who was there for me all the time, and continually pushed me to keep going whenever something did not work, or when I could not find the right words. Her constant support and encouragement got me through the last few years. I would also like to thank my son, for the joy he has brought into my life in the last six months of writing this thesis. With him around there was never a dull day.

Finally, I would like to thank my supervisor, John Plaice, and his family, for their friendship, hospitality, encouragement, support, and general willingness to chat about life when the ideas were not flowing and everyone's brains needed a rest. In particular, I could not have completed this thesis without my supervisor, who was always willing to help, even well into his evenings when he moved to the other side of the world. His knowledge of mathematics, English grammar, LaTeX, and the world in general far exceeds mine. He always pushed me to aim for nothing but perfection, and put up with my work when it fell short. During my candidature he has helped me to improve my writing skills by orders of magnitude, and taught me to be rigorous in everything that I do. For that I extend to him the greatest thanks.

Contents

List of Figures

Chapter 1

Introduction

This dissertation presents the programming language TransLucid, from its denotational semantics to its concrete implementation. TransLucid is an *intensional* programming language, meaning that a variable denotes an *intension*, which is an array of arbitrary rank and infinite extent. Programming with infinite arrays allows the data flow of a computation to be made explicit, with every subcomputation explicitly indexable. The most immediate application of such a language is in parallel programming; by writing down equations describing this data flow, the programmer is not concerned with the specific implementation details of synchronisation, locking, shared memory, and so on, but with expressing the problem in a manner that makes the available parallelism obvious. The interpreter presented in Chapters 8 and 9 is currently available at `http://translucid.web.cse.unsw.edu.au`. The entire presentation is given in a bottom-up manner, building on simple concepts at each step, since the concepts are too new for a top-down presentation.

TransLucid is a descendant of the dataflow language Lucid, first presented by Wadge and Ashcroft in 1976 [7]. In order to understand the contributions this thesis makes to TransLucid, it is necessary to examine the historical context, mainly the development of Lucid, which left open several problems, all of which are solved in the following chapters.

We begin in 1974, with the mathematician William (Bill) Wadge, and the computer scientist Edward (Ed) Ashcroft. Wadge made the observation that it is typical to write statements like the following in a program:

```
I = 0
while (I < N)
{
  ...
  I = I + 1
}
```

Mathematically, these statements are nonsensical, in particular `I = I + 1`. To a mathematician, a variable is not assigned to, but is a mathematical object defined by an equation, or set of equations, and is typically parameterised by one (or more) dependent variable(s). Wadge realised that the variable `I` could be described by the following equations:

```
first I = 0
next  I = I + 1
```

Initially, when Wadge took this idea to Ashcroft, the latter responded positively, but with the question, "What is the semantics?" The first answer provided by Wadge was to give an operational explanation of what was going on—which is not semantics. Wadge, having studied the Baire space (irrationals with infinite decimal expansion) while doing his PhD, soon after realised that the semantics of the variable `I` is an infinite stream, $\langle 0, 1, 2, \ldots \rangle$, representing the history of `I` as it varies through "time". This infinite stream obeyed the prefix order, which is that a stream s is less defined than another, s', if s is a prefix of s'—in other words, that it was necessary to compute elements in order. It was seen this way exactly because the operational view of `I` suggested the semantics; since `I` was a loop variable, it had to be evaluated sequentially.

This exchange, here between Wadge and Ashcroft, in which several threads of ideas interact, would be repeated as Lucid evolved. The first, and most important, of these threads is the question, "What is its semantics?" The second thread, which we have also seen, is how the operational view of computation interacts with the semantics, and is also frequently recurring. There are other significant threads of ideas, each is introduced as it occurs; each is solved completely in this thesis, but their presentation is deferred, because the solutions only become clear in their entirety.

Lucid was first presented to other researchers in 1975 at the *Symposium on Proving and Improving Programs*, hosted by Gilles Kahn and Gérard Huet in Arc-et-Senans (France). There, it quickly became clear that Kahn's work on dataflow networks, and Wadge and Ashcroft's work on Lucid, were intimately related. In 1977, Kahn wrote [23] about this connection:

> The style of programming also recalls `LUCID`, which has a similar semantics. The obvious pay off will be in easier correctness proofs. Note also that this programming language is just what is needed to compute over real numbers with unlimited accuracy.

Lucid streams were soon understood as dataflow networks, and a new operator, `fby` (*followed by*) was introduced. As a result, the two declarations for `I`, as above, could be replaced by one:

```
I = 0 fby I + 1
```

Another luminary of the same period, Alan Kay, designer of Smalltalk, also noticed Lucid. For Kay, a Smalltalk object was a function transforming a sequence of messages received into a sequence of messages sent. In a 2004 interview [21], Alan Kay said:

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

> insight: which was that tail-recursive loops and Lisp are so clean because you're generating the right-hand side of all the assignment statements before you do any rebinding. So you're automatically forced to use only old values. You cannot rebind, so there are no race conditions on anything.
>
> You just write down all of those things, and then when you do the tail recursion, you rebind all of those variables with these new values. Strachey said, "I can write that down like a sequential program, as a bunch of simultaneous assignment statements, and a loop that makes it easier to think of." That's basically what Lucid did—there is no reason that you have to think recursively for things that are basically iteration, and you can make these iterations as clean as a functional language if you have a better theory about what values are.

The intuition Kay held for Smalltalk objects was that they were filters between Lucid variables.

Although Wadge and Ashcroft wrote about dataflow, they were already aware by 1977 that the primitives of Lucid allowed the creation of programs that could not be interpreted as pure dataflow, but in a demand-driven manner, which they later called *eduction*. The first Lucid interpreters, implemented by Tom Cargill and David May, which were subsequently refined by Calvin Ostrum [26], evaluated elements of streams in this demand-driven manner, meaning that some elements were never even computed. In fact, these interpreters even worked if there were elements defined by nonterminating computations, as long as those elements were never requested. However, this model of evaluation was not consistent with the dataflow semantics, because the prefix order on streams required that a stream be defined sequentially, from zero, in effect, forcing iteration. This brings us back to the second thread of ideas: the way in which the operational view influences the semantic view. We will return to this semantic problem shortly.

The term *eduction* was used around the same time (1978) for exactly the same purpose in a completely different area of computer science. David P. Reed published his PhD thesis *Naming and Synchronization in a Decentralized Computer System* [38], in which a transaction in a distributed system could be defined by supposing an independent transaction on a complete (virtual) copy of the system, whose result would be then integrated back into the (in the mean time) possibly-changed physical system, should there be no contradictions. If there were, then the transaction would need to be restarted. This principle is the basis for *software transactional memory*. The actual implementation of this idea requires a demand-driven strategy to acquire copies of components of the original system, as needed, for the purposes of the transaction; this strategy Reed also called eduction. It is unclear how two different research groups ended up, at the same time, with exactly the same terminology for what was essentially the same process, used in different situations.

In the first interpreters, elements of streams were *cached*. This was fairly trivial, since streams only varied in a single time dimension. A cached evaluator worked as follows: the user would request the value of some stream, say, `I`, defined above, at some time, say, 6. This would then need `I` at time 5, which would need `I` at time 4, down to time 0. Since every stream varied in only the time dimension, it was trivial to remember the value

resulting from each pair of requests. Then, if a request later came along for `I` at some time, for which the value at that time had already been computed, it could just be looked up in the cache. This is in fact the next thread of ideas: it is desirable to cache results of computation, to reduce repeated computation of the same thing. However, as will soon be seen, this became increasingly difficult as the language evolved.

At the end of their 1977 *CACM* article [8], Wadge and Ashcroft gave their vision for Lucid:

> There are, besides iteration, several other features so far not discussed which a programmer would expect to find in a high-level programming language. These include arrays, structured data and user-defined, possibly recursive, functions. Naturally any such extensions must be compatible with the denotational approach; the addition of imperative features would make the rules of inference invalid. Function definitions offer no real difficulty, because, as was noted in Section 2 [of the paper], recursion equations are simply assertions. The addition of arrays is not quite so straightforward, but is possible if we allow the value of a variable to depend on space as well as time parameters (David May's interpreter deals with arrays in this way). Details will be given in a subsequent paper.

Solving these problems took a lot longer than expected, precisely because of the interplay between methodology, semantics and implementation. As a problem was being solved, it was also generalised, which forced advances in other aspects of the language. The above problems are all solved in this thesis, and due to the history of the language, not always in the manner envisioned by Wadge and Ashcroft, but using a more general approach where applicable. In particular, the introduction of intensions and higher-order functions over intensions was non-trivial, and forms the key contribution of this thesis.

By the time that Wadge and Ashcroft wrote their first book on Lucid in 1985, *Lucid: the Dataflow Programming Language* [43], streams were written with a single definition, using operators that transformed streams. Their vision was that a Lucid function was essentially a filter—as was envisioned by Kay—which transformed streams of data. At the start of their book, Wadge and Ashcroft presented an example program that computes the root mean square of an input stream. With the appropriate function definitions, the expression `sqroot(avg(square(a)))` is a filter producing a stream, whose elements are the root mean square of the elements of `a`, up to that point. They likened this to Unix pipes, whereby one could write something like `(sqroot | avg | square) < a` to achieve the same effect.

With the initial intuition that a variable is an infinite stream, and the subsequent realisation that problems can be described as data flowing through a network, Wadge and Ashcroft saw the potential for exploiting the maximum parallelism in a problem, by simply writing down the problem declaratively, rather than a sequential solution to that problem. In their 1985 book *Lucid: the Dataflow Programming Language*, they wrote [43, p.31]:

> Machines exist to *solve problems*; to simulate an airplane in flight, to control a refinery, to compute the gas bill. The role of the programmers in this context

> is to communicate the problem to the machine—the programming language is their tool. The reason that the programmer wants to get *away* from the machine is to get *closer* to the problem.

These three sentences epitomised their vision for Lucid. Wadge and Ashcroft saw the potential for parallel programming, at the time in its infancy, and understood that to solve parallel programming problems, it was imperative that programming move away from the sequential programming model, and towards a language in which parallel problems could naturally be expressed.

Although, by 1977, they recognised that the semantics and implementation were different to pure dataflow, it continued to inspire their vision, which made it difficult for them to look further. By initially looking at a stream such as `I`, which provided a very good model for declarative iteration, Wadge and Ashcroft restricted themselves to thinking only about iteration, i.e., a single stream varying in time, whose elements depended on the value of the previous element. Furthermore, by doing this, they were essentially focusing on task-level parallelism, and never considered data-level parallelism; they had the correct intuition with regards to connecting different filters together, but by forcing everything to be iterative, they lost the potential for every element in an entire stream to be computed in parallel.

The lack of functions in Lucid was problematic, because they were necessary for transforming streams. So by their 1985 book [43], there were user-defined functions in Lucid, but they were not first-class objects. This led to criticism from the functional programming community, who did not consider Lucid to be a functional language because it did not have higher-order functions, even though every single Lucid variable was itself a function (a stream is a function from naturals to values), so any Lucid function was, in fact, inherently higher-order, just not in the sense that the functional programming community wanted. In addition, Wadge wanted an approach to the implementation of functions that was consistent with the eductive implementation of Lucid variables.

At the end of their 1985 book [43, p.233], the authors presented an idea for a hypothetical version of Lucid, in which there could be streams of functions, as well as higher-order functions over streams. They wanted to be able to define a variable like `P` as follows:

```
P = (\x : x) fby (\x : x * P(x))
```

Functions as first-class values is the next thread of ideas in Lucid, beginning with this idea about allowing streams of functions; the idea is picked up again later. The TransLucid solution to the above problem is presented in §4.13.

Also at the end of their 1977 paper [8], Wadge and Ashcroft presented the possibility of manipulating arrays in Lucid, essentially allowing streams of arrays, or explicit multidimensionality, instead of the current single-dimensional model. So the thread of ideas was, "How do we allow more dimensions to be manipulated by the user?" In their 1985 book [43], Wadge and Ashcroft presented several implementations which attempted to answer that question. To achieve this multidimensionality, additional *space* dimensions were made available to the user through additional operators. There were operators such as

`cby` (*continued by*), amongst others, which were "space" analogs of the functions used to manipulate streams varying in "time". These functions had slightly different semantics in each implementation, but the basic idea was that they allowed some limited manipulation of the "space" and "time" dimensions. These ideas were heading in the right direction, but since the user could not explicitly manipulate dimensions as first-class values, the programs that could be written were limited by the operators made available in the language. In addition, the dimensions over which a function operated were preordained, so if the user wrote a function that operated over a stream varying in space dimension s_0, then it could not be made to operate over space dimension s_1, forcing it to be rewritten.

In fact, as shown in this thesis, both higher-order functions and multidimensionality need each other. So it is not surprising that without either concept being completely understood, neither of them was successful. Multidimensionality is not particularly interesting without higher-order functions, because functions are necessary to structure data; and higher-order functions need multidimensionality for implementation purposes.

Meanwhile, the desire to understand the concepts of Lucid denotationally continued. So, now, we return to the problem of the semantics of streams versus the implementation. It was later realised that the correct order for streams is, in fact, not the prefix order, but the *Scott* order, which says that a stream s is less defined than s' if s' defines *at least* the elements that s defines, and that they concur for those elements.

The Scott order of streams led to the idea of the *intension* [15] in 1986. This had the implication that a variable is not a stream in one dimension, but an infinite array varying in arbitrary dimensions (the intension is a generalisation of the Lucid stream, extended to multiple dimensions, so a stream can be seen as an intension varying in one dimension). In logic, an intension is a mapping from *possible worlds* to *extensions*, see the work by Carnap [12] and Montague [14, 42]. In Lucid, an intension is a mapping from possible worlds (multidimensional indices) to extensions (atomic values).

From this point onwards, the questions driving the development of Lucid were about allowing the user to explicitly manipulate all of the concepts referred to in the semantics. So in continuing the thread about the user manipulating dimensions, the next natural question was, "What if the user can explicitly manipulate dimensions?" The answer to this question was Indexical Lucid, introduced by Faustini and Jagannathan in 1991 [16, 17], and later presented in the 1995 book *Multidimensional Declarative Programming* [9]. In Indexical Lucid, a function `rotate`, which takes as input an intension varying in dimension `d1` and rotates it into dimension `d2`, i.e., returning an intension varying in dimension `d2`, can be defined as follows:

```
rotate.d1,d2 X = X @.d2 #.d1
```

If `a` and `b` were actual dimensions, and `A` were an intension varying in dimension `a`, then the function application:

```
rotate.a,b A
```

would rotate `A` into dimension `b`. In Indexical Lucid, functions take *dimensional* parameters, and *intensional* parameters, and cannot be partially applied. So here, `a` and `b` are not

passed as values as in ML-style languages, but are passed as the dimension names a and b to rotate. Indexical Lucid made huge progress in the way that programs were written. However, it always came back to the question, "What is its semantics?" In *Multidimensional Declarative Programming* [9, Chapter 3], the authors gave an intuitive description of the semantics, but no rigorous mathematical semantics. In addition, the cached evaluator previously used did not work with the huge number of dimensions now used in a program; it was not clear how to tag the result of a computation, nor how to later look it up, since not all dimensions being used at any given time were relevant to the value of any particular expression. The implementation of a cached evaluator became an open problem, and is solved in Chapter 6.

Continuing along the thread of introducing first-class functions, the next question was, "Can we introduce higher-order functions?" In 1999, Rondogiannis and Wadge presented a system for transforming higher-order functional programs into multidimensional intensional programs with no functions [39]. However, their system only worked for a restricted class of functional programs, in which functions could take other functions as argument, but could not return functions—so partially applied functions were not allowed either. This was a step in the right direction, but did not produce a Lucid language with functions, only a means to transform other programs to Lucid with no functions.

To return to the thread about explicitly manipulating dimensions, the next question was, "Do dimensions necessarily have to be special parameters to functions?". In 1999 and 2000, Paquet and Plaice suggested with Tensor Lucid [27, 29], that dimensions could be first-class values. Then in 2000, Plaice proposed Multidimensional Lucid [31], in which any atomic object could be used as a dimension. If dimensions could become first-class values, it was only natural that contexts should become first-class values. In 2001, in developing his GIPSY project [28], Paquet suggested that the context change operator become $E' @ E$, meaning that E should evaluate to a context κ, and E' would be evaluated by perturbing the running context with κ. However, the implementation issues became further complicated.

In 2005, Plaice started the TransLucid project, with an idea from Wadge called *lazy eduction*. This was a back-and-forth interaction between the cache and the evaluator, whereby the cache would build up the dimensions used to produce a result by attempting evaluation with only the dimensions known about. If more dimensions were required, the cache would record this, and the process would continue until a result was obtained. All of this work culminated with Plaice's Habilitation thesis in 2010 [32], which attempted to provide a denotational semantics, operational semantics and cache semantics for TransLucid with higher-order functions and first-class dimensions and contexts. Whilst the ideas were heading in the right direction, clearly defining the future work, much was not implemented, and the set of primitives defined therein turned out to be definable by an even smaller set, leading to simpler semantics and implementation.

It is at this point that the present thesis steps in and draws all of these threads together. This thesis provides an answer to each of the following questions from the history of Lucid: "What is its semantics?", "Can we have first-class higher-order functions?", "Can we

implement a cached evaluator?", and "Can contexts and dimensions be first-class values, to be manipulated as the user desires?"

Chapter 2 presents the semantics of Core TransLucid, which is a basic functional programming language, with only four more syntactic elements than a standard functional language, in which an expression with no free variables denotes an intension. Furthermore, intensions, functions, contexts and dimensions are all part of the semantic domain (Definition 1), meaning that they are all first-class values.

After the denotational semantics, much of the work in this thesis exists to produce a concrete language that is usable and implementable. In order to move from Core TransLucid to Concrete TransLucid, presented in Chapter 8, several different syntaxes are presented, with complete syntactic transformations and semantics. To illustrate the syntax and transformations, the remainder of this Chapter will use the example of a simple matrix multiplication, presenting it in each different syntax. Matrix multiplication is presented in (canonical) TransLucid, and explained in detail in §4.7.

Suppose that A and B are two matrices, of sizes 2×3 and 3×2 respectively, both indexed by dimensions a and b. We can visualise these matrices in the following tables:

$`A'$	0	1	2	$\overset{\#.a}{\rightarrow}$
0	1	2	3	
1	4	5	6	
$\#.b\downarrow$				

$`B'$	0	1	$\overset{\#.a}{\rightarrow}$
0	9	8	
1	7	6	
2	5	4	
$\#.b\downarrow$			

In fact, in Core TransLucid, A and B are two-dimensional intensions, of infinite size, but we only need concern ourselves with the relevant entries. With this example it should be clear why we refer to this style of programming as *Cartesian Programming*, because all variables are arrays sitting in a Cartesian coordinate space, indexed by as many dimensions as necessary. In Core TransLucid the multiplication of A and B is defined by the expression:

$$
\begin{array}{l}
W \texttt{ wherevar} \\
\quad A' = A \texttt{ @ } [a \leftarrow \#.d] \\
\quad B' = B \texttt{ @ } [b \leftarrow \#.d] \\
\quad Z = A' \times B' \\
\quad W = \texttt{if } \#.d \equiv 0 \texttt{ then } Z \texttt{ else } Z + W \texttt{ fi} \\
\texttt{ end} \\
\texttt{wheredim} \\
\quad d \leftarrow 2 \\
\texttt{end}
\end{array}
$$

Here, we create a new dimension d, and initialise its ordinate to 2, resulting in the `wherevar` clause being evaluated in the context $\{d \mapsto 2\}$. The variables A' and B' are the result of rotating A into dimension d along dimension a and B into dimension d along dimension b. Then, Z is a $2 \times 3 \times 2$ parallelepiped, formed as the result of multiplying every corresponding element of A' and B'. The variable S is the result of summing the elements of Z in the

d direction. Since the size of the common dimension of the matrices is 3, we need to pull out the third entry of the sum (counting from zero), hence initialising the d-ordinate to 2. The variables used to produce the result can be visualised as follows:

‘A’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	1	2	3	
1	4	5	6	
$\#.b\downarrow$				

‘B’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	9	7	5	
1	8	6	4	
$\#.a\downarrow$				

‘Z @ $[a \leftarrow 0]$’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	9	14	15	
1	36	35	30	
$\#.b\downarrow$				

‘Z @ $[a \leftarrow 1]$’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	8	12	12	
1	32	30	24	
$\#.b\downarrow$				

For the variable W which sums the elements of Z, we write dimension d to the right, which is the direction of the summation, and the boxes represent the values making up the final result.

‘W @ $[a \leftarrow 0]$’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	9	23	38	
1	36	71	101	
$\#.b\downarrow$				

‘W @ $[a \leftarrow 1]$’	0	1	2	$\overset{\#.d}{\rightarrow}$
0	8	20	32	
1	32	62	86	
$\#.b\downarrow$				

The final result is reproduced below using a single table, rotating the table so that dimension a is to the right again.

‘W @ $[d \leftarrow 2]$’	0	1	$\overset{\#.a}{\rightarrow}$
0	38	32	
1	101	86	
$\#.b\downarrow$			

In Core TransLucid, we can write down higher-order functions over intensions, although it is a little verbose. Therefore, Chapter 3 presents (canonical) TransLucid, which provides several function abstractions and applications, implemented as syntactic sugar over Core TransLucid, and demonstrates the canonical way that the user would write higher-order functions. These function abstractions are:

1. An *intension* abstraction, which promotes the intension to a first-class object;
2. A *base* abstraction, whose body must define a single atomic value;
3. A *call-by-value* abstraction, whose parameter is evaluated when applied, and whose body defines an intension;
4. A *call-by-name* abstraction, whose parameter is evaluated when used, and whose body defines an intension.

In addition, the `wherevar` and `wheredim` clauses are combined into a single `where` clause. Using the syntactic sugar of TransLucid, we define a function *matrix_multiply* that takes as input two matrices, the dimensions in which they vary and the size of their common dimension, this example corresponds to the one in §4.7:

$$
\begin{array}{l}
\texttt{fun}\ \mathit{matrix\_multiply}.d_c.d_r.k\ X\ Y = W \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow k \\
\quad \texttt{var}\ X' = \mathit{rotate}.d_c.d\ X \\
\quad \texttt{var}\ Y' = \mathit{rotate}.d_r.d\ Y \\
\quad \texttt{var}\ Z = X' \times Y' \\
\quad \texttt{var}\ W = \mathit{foldl}.d.\mathit{plus}.0\ Z \\
\texttt{end}
\end{array}
$$

Then the expression $\mathit{matrix\_multiply}.a.b.3\ A\ B$ gives the matrix multiplication of A and B. The `fun` declaration is also presented with TransLucid, and it is syntactic sugar for a variable declaration, which defines a function. In this case it is the equivalent to:

$$\texttt{var}\ \mathit{matrix\_multiply} = \lambda^b d_c \to \lambda^b d_r \to \lambda^b k \to \lambda^n X \to \lambda^n Y \to W\ \texttt{where}\ \cdots\ \texttt{end}$$

Here, X' and Y' are defined using the function $\mathit{rotate}.d.d'\ X$ (§4.5), which rotates dimension d of its argument X into d', and variable S is defined using $\mathit{foldl}.d.f.z\ X$ (§4.6), which computes the left fold of its argument X along direction d using the function f and with z as the initial element.

After presenting TransLucid, Chapter 4 gives an introduction into the methodology of programming in TransLucid, by presenting how TransLucid solves several standard programming problems, with particular reference to the geometric view of an intension. This is a key chapter, because it is through this geometric view that it can be seen how the flattening of the data involved in a computation allows data dependencies to be made explicit, with the goal of drawing out maximal parallellism. This is in contrast to a language such as Haskell, which requires the programmer to explicitly program the parallelism, using extensions such as Control Parallel [4], Concurrent Haskell [3] and Data Parallel Haskell [1], the result being that if an opportunity for parallelism is missed by the programmer, then it cannot be recovered by a compiler.

In moving TransLucid towards an implementation, Chapter 5 presents Operational TransLucid, which moves manipulations of the environment into manipulations of the context—another key contribution of this thesis. It is necessary to do this because a direct implementation of the denotational semantics would be horribly inefficient. To achieve this, all uses of the environment in function abstractions are transformed to uses of the context, and a function abstraction must hold on to the ordinate of any dimension used as a parameter by a surrounding abstraction. In addition, call-by-name abstractions are transformed to call-by-value abstractions that take an intension as parameter, call-by-name applications are transformed to call-by-value applications that wrap the right-hand

side in an intension abstraction, and the `where` clause is split into `wheredim` and `wherevar`. The matrix multiplication example is transformed to the following (note that the user does not see this syntax, it is purely for implementation purposes):

$$
\begin{aligned}
&\texttt{var } \mathit{matrix\_multiply} = \lambda^{\mathrm{b}}_{\circ}\ \phi_{d_c} \rightarrow \\
&\qquad\qquad\qquad\qquad \lambda^{\mathrm{b}}_{\circ}\ \{\phi_{d_c}\}\ \phi_{d_r} \rightarrow \\
&\qquad\qquad\qquad\qquad \lambda^{\mathrm{b}}_{\circ}\ \{\phi_{d_c}, \phi_{d_r}\}\ \phi_k \rightarrow \\
&\qquad\qquad\qquad\qquad \lambda^{\mathrm{v}}_{\circ}\ \{\phi_{d_c}, \phi_{d_r}, \phi_k\}\ \phi_X \rightarrow \\
&\qquad\qquad\qquad\qquad \lambda^{\mathrm{v}}_{\circ}\ \{\phi_{d_c}, \phi_{d_r}, \phi_k, \phi_X\}\ \phi_Y \rightarrow W \\
&\texttt{wherevar} \\
&\quad \texttt{var } X' = \mathit{rotate}.\underline{\phi_{d_c}}.\underline{\phi_d}\ (\downarrow \underline{\phi_X}) \\
&\quad \texttt{var } Y' = \mathit{rotate}.\underline{\phi_{d_r}}.\underline{\phi_d}\ (\downarrow \underline{\phi_Y}) \\
&\quad \texttt{var } Z = X' \times Y' \\
&\quad \texttt{var } W = \mathit{foldl.d.plus}.0\,!\,(\uparrow \{\phi_{d_c}, \phi_{d_r}, \phi_k, \phi_X, \phi_Y\}\ Z) \\
&\texttt{end} \\
&\texttt{wheredim}_{\circ} \\
&\quad \texttt{dim } \phi_d \leftarrow k \\
&\texttt{end}
\end{aligned}
$$

The question of the cached evaluator is solved in Chapter 6. The evaluator discovers the dimensions of relevance using a back-and-forth interaction with the cache. Initially, an attempt is made to evaluate with no dimensions; should a dimension be required, the cache records the dimension and its ordinate and tries again. This continues until an answer is produced, at which point the cache has built up a tree describing the dimensions necessary to evaluate an intension.

Consider an evaluation of the Fibonacci numbers. We would like to compute the value of the following expression:

$$
\begin{aligned}
&\mathit{fib} \\
&\texttt{where} \\
&\quad \texttt{dim } d \leftarrow 3 \\
&\quad \texttt{var } \mathit{fib} = \texttt{if } \#.d \equiv 0 \texttt{ then } 1 \\
&\qquad\qquad\qquad \texttt{elsif } \#.d \equiv 1 \texttt{ then } 1 \\
&\qquad\qquad\qquad \texttt{else } \big(\mathit{fib}\ \texttt{@}\ [d \leftarrow \#.d - 1]\big) + \big(\mathit{fib}\ \texttt{@}\ [d \leftarrow \#.d - 2]\big)\ \texttt{fi} \\
&\texttt{end}
\end{aligned}
$$

Without the cache, the evaluator requests the pair $(\mathit{fib}, \{d \mapsto 3\})$, which requests the pairs $(\mathit{fib}, \{d \mapsto 2\})$ and $(\mathit{fib}, \{d \mapsto 1\})$, the latter gives the result 1, and the former requests $(\mathit{fib}, \{d \mapsto 1\})$ and $(\mathit{fib}, \{d \mapsto 0\})$ which both produce 1. Even with the third Fibonacci number there are two requests for the first Fibonacci number.

Evaluating with the cache is slightly different. Initially we ask for $(\mathit{fib}, \{d \mapsto 3\})$, and the cache starts the computation with the empty context $\emptyset$. When the evaluator reaches `#`$.d$ inside the condition of the `if` expression, it returns the fact that it needs dimension d. The cache records this, and then attempts to evaluate again with the context $\{d \mapsto 3\}$.

This time, the `else` branch is taken, and a request is made at $\{d \mapsto 2\}$. This whole process is repeated, except that this time the cache knows that the d-ordinate is required, so it starts evaluating immediately, and again comes to the `else` branch, which requests $(\mathit{fib}, \{d \mapsto 1\})$. Again this is repeated, but this time the result 1 is returned, so the cache remembers that $(\mathit{fib}, \{d \mapsto 1\}) = 1$. As evaluation continues, the same process is carried out for the values at 0 and 2. Finally, evaluation returns to the right-hand side of the addition when the context is $\{d \mapsto 3\}$, which requests $(\mathit{fib}, \{d \mapsto 1\})$. This time, that result is already stored in the cache, so it does not attempt to recompute it, and simply returns the result.

As a final note, the cache scheme is so effective, that the plethora of dimensions resulting from the transformation to Operational TransLucid causes no problems for it.

In addition, this thesis seeks to answer two additional questions: "How far can we take static analysis of TransLucid programs?" and, "Can we produce not just an expression evaluator, but a complete programming environment with multidimensional input and output, that remains completely declarative?"

Chapter 7 presents a type inference algorithm for TransLucid, which starts with the idea that the principal type of an object is itself. To support this idea, types are inferred in a framework which generates subtyping constraints. For example, the type of the number 42 is not **intmp**, but α, with the constraint $(42 \leq \alpha)$. The types of larger programs are built up by composing the constraints generated by subexpressions, and placing them in a constraint graph, if that graph is closed then it has a solution, and the program is well-typed.

An example type that can be inferred in this system is the Y-combinator, whose definition is:

$$\lambda^{\mathrm{n}} f \to \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)\ \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)$$

and whose type is inferred as:

$$\forall\alpha.(\uparrow (\uparrow \alpha \xrightarrow{v} \alpha) \xrightarrow{v} \alpha)$$

which is something that unification-based type-inference algorithms cannot do, such as is used by Haskell.

Then finally, Chapter 8 presents Concrete TransLucid, along with the TransLucid system, which is a synchronous reactive programming environment, supported by the TransLucid standard library presented in Chapter 9. Concrete TransLucid provides concrete syntax for expressions and atomic objects, so that the user can write down real programs, using real atomic objects available in the host environment. The TransLucid system evaluates demands across several instants, taking finite multidimensional arrays as input and producing finite multidimensional arrays as output. Additionally, the behaviour of the system can be modified, by the programmer adding the appropriate declarations at each instant. All of this is completely declarative, with the demands for computation being carried out each instant, and the results written to the required outputs. This solves the problem that Simon Peyton Jones referred to when he said that "Haskell is useless" [2].

By providing a semantics for side effects to occur in discrete time instants, this thesis has produced a completely declarative system, which is capable of affecting the outside world.

To finish with the matrix multiplication example in Concrete TransLucid, with the system evaluating demands at each instant, and taking inputs and producing real output, we can create a system that evaluates matrix multiplications every instant, with new inputs each time. We declare the variables A and B to be inputs, and variable C to be an output:

$$\begin{aligned}
&\texttt{indim}\ a\\
&\texttt{indim}\ b\\
&\texttt{invar}\ A \leftarrow \mathit{URL}\\
&\texttt{invar}\ B \leftarrow \mathit{URL}\\
&\texttt{outvar}\ C\ \texttt{:=}\ \mathit{matrix_multiply.a.b.3}\ A\ B\\
&\texttt{outvardest}\ C \rightarrow \mathit{URL}
\end{aligned}$$

Each of the three *URL*s would be set to point to the appropriate location, and as long the inputs provide a different matrix at each instant, the request for assignment into variable C will compute a different matrix, upon each instant, until the end of time.

The open problems left by Lucid in the 1970s and 1980s are closed by this thesis, but many new questions are raised, which research teams could spend many fruitful years working on. Chapter 10 examines the different problems raised, and points in the direction of their solution, with much of the future work being in static analysis, so that efficient code with maximal parallelism, as appropriate for the target architecture, can be generated from the same program.

Chapter 2

Core TransLucid

This chapter presents the key concepts and semantics for TransLucid. The language presented here is *Core TransLucid*, a higher-order functional language in which a variable denotes an *intension*, which is a mapping from contexts to ordinary values, where a context is a set of dimension–ordinate pairs. Core TransLucid is a standard functional language, to which are added four syntactic constructs: the tuple, the context, the context change, and the `wheredim` clause. Core TransLucid's semantics requires the definition of domains for contexts and intensions, in addition to ordinary functions.

The chapter begins with an extensive presentation (§2.1) of the concept of intension, the very basis for the language, and the hardest to grasp. This intuition leads naturally to the definition of the domains needed to define Core TransLucid's semantics (§2.2). In this semantics, the evaluation of an expression takes place with respect to the interpretation of the constant symbols in the expression, to an environment mapping identifiers in the expression to intensions, and to a current context. During this evaluation, new dimensions may be allocated, for which two approaches, one non-deterministic, one deterministic, are presented.

The full language used to program, simply called TransLucid, will be presented in the next chapter. It uses the same semantic domains, but requires syntactic extensions, all directly translatable to Core TransLucid.

2.1 Intensions, contexts and functions

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

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

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	$\overset{\texttt{\#}.x}{\rightarrow}$
	0	1	2	3	⋯

‘#.y’	
0	0
1	1
2	2
3	3
$\texttt{\#}.y\downarrow$	⋮

‘ + ’	
	+

‘ ∗ ’	
	×

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 TransLucid: 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 ‘$2 * \texttt{\#}.x$’, 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	$\overset{\texttt{\#}.x}{\rightarrow}$
	2	2	2	⋯

‘ ∗ ’	0	1	2	$\overset{\texttt{\#}.x}{\rightarrow}$
	×	×	×	⋯

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

For expression ‘$42 + (2 * \texttt{\#}.x) + \texttt{\#}.y$’, the subexpressions ‘42’ and ‘ + ’ (both rank $\emptyset$), ‘$2 * \texttt{\#}.x$’ (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 one”	“zero”	“one”	“two”	$\cdots$
`ES`	$\cdots$	“menos dos”	“menos uno”	“cero”	“uno”	“dos”	$\cdots$
`FR`	$\cdots$	“moins deux”	“moins un”	“zéro”	“un”	“deux”	$\cdots$
`#.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: Faustini and Wadge [15] state, with respect to an example about temperature, “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 §2.2 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 TransLucid 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	$\overset{a}{\to}$
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	$\overset{\#.x}{\to}$
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
            end
         wheredim
            d ← n
         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	$\overset{\#.\chi^1_{\#.\rho}}{\to}$
	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:

$$
\begin{array}{l}
\texttt{fun}\ fact.n = F \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ F = \texttt{if}\ \#.d \equiv 0\ \texttt{then}\ 1\ \texttt{else}\ \#.d \times \big(F\ \texttt{@}\ [d \leftarrow \#.d - 1]\big)\ \texttt{fi} \\
\texttt{end}
\end{array}
$$

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. These ideas will be developed in detail in the next chapter.

All of the primitives of TransLucid have now been presented informally; their formalization follows in the next section.

2.2 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 (Figure 2.1, p.23), then give the rules. This presentation order is chosen to be consistent with the bottom-up nature of the dissertation: the definitions for domains are key.

2.2.1 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:

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

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

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

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

$$(f \triangleleft S)(v) \quad = \quad \left\{ \; f(v), \quad v \notin S. \right.$$

2.2.2 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*

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

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

We call

- *D the set of* atomic values*; an atomic value is written δ; a subset of D is written Δ;*
- *$\mathbf{D}_{\mathtt{atomic},m}$ the set of* atomic functions *of arity m, such as arithmetic and Boolean operators; an atomic function is written op;*
- *$\mathbf{D}_{\mathtt{ctxt}}$ 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*;*
- *$\mathbf{D}_{\mathtt{intens}}$ the set of* intensions*, mapping contexts to values; an intension is written η;*
- *$\mathbf{D}_{\mathtt{func}}$ the set of* functions*; a function is written f.*

Note that $\mathbf{D}_{\mathtt{atomic},1}$, $\mathbf{D}_{\mathtt{ctxt}}$ and $\mathbf{D}_{\mathtt{intens}}$ are all subsets of $\mathbf{D}_{\mathtt{func}}$. 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 $\kappa = \kappa' \lhd (\mathrm{dom}\ \kappa)$, we have $\eta(\kappa) = \eta(\kappa')$, ensures that a TransLucid 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 $\delta \in D$, $\delta \sqsubseteq \delta$.*
- *For all $op, op' \in \mathbf{D}_{\mathtt{atomic},m}$, $op \sqsubseteq op'$ iff $op = op' \lhd (\mathrm{dom}\ op)$.*

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

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. $(\mathbf{D}_{\texttt{atomic},m}, \sqsubseteq)$;
3. $(\mathbf{D}_{\texttt{ctxt}}, \sqsubseteq)$;
4. $(\mathbf{D}_{\texttt{intens}}, \sqsubseteq)$;
5. $(\mathbf{D}_{\texttt{func}}, \sqsubseteq)$.

Proof. Mostly standard. See §2.4.1, p.27. □

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

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

The rank of an intension can be infinite, although in practice, in TransLucid, as we shall see, the ranks of intensions in programs we write have very low dimensionality.

The semantics of TransLucid relies on the dynamic allocation of dimensions, something similar to the dynamic allocation of memory in imperative languages. A key point for the semantics to have meaning is that it does not matter which dimensions are allocated. We therefore need to define a notion of *equivalence* (Definition 7), which requires first definitions of *occurrence* and *substitution* of atomic values in a value (Definitions 4–6).

Definition 4. *Let* $d \in \mathbf{D}$ *be a value, and* $\delta \in D$ *be an atomic value. We say that* δ occurs in d *iff*

$$\begin{cases} \delta = d, & d \in D \\ \exists (\delta_1, \ldots, \delta_m) \in \mathrm{dom}(op) \text{ s.t.} & \\ \qquad (\exists i \text{ s.t. } \delta \text{ occurs in } \delta_i) \vee (\delta \text{ occurs in } op(\delta_1, \ldots, \delta_m)), & d = op \in \mathbf{D}_{\mathrm{atomic,m}} \\ \delta \in \mathrm{dom}(\kappa) \vee \exists \delta' \in \mathrm{dom}(\kappa) \text{ s.t. } \delta \text{ occurs in } \kappa(\delta'), & d = \kappa \in \mathbf{D}_{\mathrm{ctxt}} \\ \delta \in \mathrm{rank}(\eta) & \\ \qquad \vee \exists \kappa \in \mathrm{dom}(\eta) \text{ s.t. } \delta \text{ occurs in } \kappa \lhd \mathrm{rank}(\eta) & \\ \qquad \vee \exists \kappa \in \mathrm{dom}(\eta) \text{ s.t. } \delta \text{ occurs in } \eta(\kappa), & d = \eta \in \mathbf{D}_{\mathrm{intens}} \\ \delta \text{ occurs in } \mathrm{dom}(f) \vee \exists d' \text{ s.t. } \delta \text{ occurs in } f(d'), & d = f \in \mathbf{D}_{\mathrm{func}}. \end{cases}$$

Definition 5. *Let $d \in \mathbf{D}$ be a value, and $\delta, \delta' \in D$ be atomic values, and suppose that δ' does not occur in d. Then the* substitution of δ' for δ in d*, written $\theta(\delta, \delta', d)$, replaces each occurrence of δ in d by δ', and is defined as follows:*

$$\begin{cases} \begin{cases} \delta', & \delta = d \\ d, & \textit{otherwise}, \end{cases} & d \in D \\ \big\{\big(\theta(\delta,\delta',d_1),...,\theta(\delta,\delta',d_m)\big) \mapsto \theta\big(\delta,\delta',op(d_1,\ldots,d_m)\big) \mid \\ \qquad (d_1,\ldots,d_m) \in \mathrm{dom}(op)\big\}, & d = op \in \mathbf{D}_{\text{atomic,m}} \\ \big\{\big(\theta(\delta,\delta',d')\big) \mapsto \theta\big(\delta,\delta',\kappa(d')\big) \mid d' \in \mathrm{dom}(\kappa)\big\}, & d = \kappa \in \mathbf{D}_{\text{ctxt}} \\ \big\{\big(\theta(\delta,\delta',d')\big) \mapsto \theta\big(\delta,\delta',f(d')\big) \mid d' \in \mathrm{dom}(\eta)\big\}, & d = \eta \in \mathbf{D}_{\text{intens}} \\ \big\{\big(\theta(\delta,\delta',d')\big) \mapsto \theta\big(\delta,\delta',f(d')\big) \mid d' \in \mathrm{dom}(f)\big\}, & d = f \in \mathbf{D}_{\text{func}}. \end{cases}$$

Definition 6. *Let $d \in \mathbf{D}$ be a value, and $\Lambda = \langle \delta_1, \ldots \delta_m \rangle$ and $\Lambda' = \langle \delta'_1 \ldots \delta'_m \rangle$ be disjoint sequences of atomic values in D, and suppose that none of the δ'_i occur in d. Then the* substitution *of Λ' for Λ in d, written $\Theta(\Lambda, \Lambda', d)$, replaces each occurrence of δ_i in d by the corresponding δ'_i, and is defined as follows:*

$$\Theta(\Lambda, \Lambda', d) = \theta\big(\delta_1, \delta'_1, \ldots, \theta(\delta_m, \delta'_m, d)\ldots\big).$$

Definition 7. *Let $d, d' \in \mathbf{D}$. Then we call d and d'* equivalent*, written $d \equiv d'$, when there exist disjoint sequences $\Lambda = \langle \delta_1, \ldots, \delta_m \rangle$ and $\Lambda' = \langle \delta'_1, \ldots, \delta'_m \rangle$ such that $d' = \Theta(\Lambda, \Lambda', d)$.*

2.2.3 Signatures, interpretations, environments and syntax

Definition 8. *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 9. *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, which will be provided by a *host environment* for the concrete language (see Chapter 8).

Definition 10. *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 $\sqsubseteq$ and $\equiv$ to environments and define the extended rank* (erank) *and extended range* (eran)*:*

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

Definition 11. *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* TransLucid expressions *over Σ and X. The* free variables *of a TransLucid expression E are written $FV(E)$. The* abstract syntax *for TransLucid expressions is given in Figure 2.1.*

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

Figure 2.1: Syntax of Core TransLucid expressions

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

2.2.4 Nondeterministic semantic rules

We will present two sets of semantic rules: one allocates dimensions non-deterministically, the other deterministically. The non-deterministic semantics was proposed in the *Multi-dimensional Programming* book [9, Chapter 3], and involves passing around an infinite set of dimensions from which dimensions can be allocated, which is split into smaller, but still infinite, sets at each syntax branch. The deterministic rules, presented in §2.2.6, lead naturally to an implementation.

The structure of the rules for the two semantics, and, in fact, all but one rule—for the `wheredim` clause—, are shared. The syntax for both semantics is, in fact, the same, and the only different rule is for the `wheredim` clause. Therefore, we only present the properties and rules once, and present the minor differences for the second semantics.

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

- Let d be a value. Then $\widehat{d}$ is a constant intension, defined by $\widehat{d} = \lambda\kappa.d$.
- Suppose Δ is an infinite set of values, and let $n \in \mathbb{N}$. We suppose that Δ can be partitioned into ${}^{0}\Delta, {}^{1}\Delta, \ldots, {}^{n}\Delta$ such that they are all mutually disjoint and all infinite.

Preamble 1. *Let X be a set of variables; $D = \Delta_S \cup \Delta_H \cup \Delta_O$, where Δ_S is a set of atomic values, and Δ_H is a second set of atomic values, called* hidden dimensions*; Δ_O is a third set of atomic values, called* other 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*

$$\begin{aligned} \Delta_O \cap \Delta_S = \Delta_O \cap \Delta_H = \Delta_S \cap \Delta_H &= \emptyset \\ \{true, false\} &\subseteq \Delta_S \\ \mathrm{erank}(\zeta) \cup \mathrm{eran}(\zeta) &\subseteq \Delta_S \\ \mathrm{dom}(\kappa) \cup \mathrm{ran}(\kappa) &\subseteq \Delta_S. \end{aligned}$$

Preamble 2. *Let* $\rho \in \Delta_O$*, and* κ *be a context such that* $\Delta = \kappa(\rho)$ *is an infinite set of values. We define*

$$_i\kappa = \kappa \dagger \{\rho \mapsto {}^i\Delta\}.$$

The notation $_i\kappa$ appears repeatedly in the semantic rules, and it is used to designate κ, where just the ρ-ordinate has been changed. For the non-deterministic rules, a smaller, but still infinite, set of dimensions is created.

Definition 12. *Suppose Preamble 1 holds, let* Δ_H *be a large infinite set of hidden dimensions, and suppose* $_i\kappa$ *is defined as in Preamble 2. Then the non-deterministic semantics for* E *with respect to* ι*,* ζ *and* κ *is given by*

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

where the rules for $[\![\cdot]\!]$ *are given in Figure 2.2, and the* `wheredim` *rule is given in Figure 2.3.*

$$\begin{aligned}
[\![x]\!]\iota\zeta\kappa &= \zeta(x)(\kappa) && (2.1)\\
[\![{}^m c]\!]\iota\zeta\kappa &= \iota({}^m c) && (2.2)\\
[\![\texttt{\#}]\!]\iota\zeta\kappa &= \kappa && (2.3)\\
[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]\iota\zeta\kappa &= \big\{[\![E_{i0}]\!]\iota\zeta({}_i\kappa) \;\mapsto\; [\![E_{i1}]\!]\iota\zeta({}_{(i+m)}\kappa)\big\} && (2.4)\\
[\![\lambda x \to E_0]\!]\iota\zeta\kappa &= \lambda d_a.[\![E_0]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) && (2.5)\\
[\![E_0.(E_i)_{i=1..m}]\!]\iota\zeta\kappa &= \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_i]\!]\iota\zeta({}_i\kappa)\big) && (2.6)\\
[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\zeta\kappa &= \text{let } d_0 = [\![E_0]\!]\iota\zeta({}_0\kappa) && (2.7)\\
&\quad \text{in } \begin{cases} [\![E_1]\!]\iota\zeta({}_1\kappa), & d_0 \equiv true \\ [\![E_2]\!]\iota\zeta({}_2\kappa), & d_0 \equiv false \end{cases} \\
[\![E_0 \mathbin{@} E_1]\!]\iota\zeta\kappa &= [\![E_0]\!]\iota\zeta\big({}_0\kappa \dagger [\![E_1]\!]\iota\zeta({}_1\kappa)\big) && (2.8)\\
[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa &= \text{let } \zeta_0 = \zeta \dagger \{x_i \mapsto \emptyset\} && (2.9)\\
&\quad\; \zeta_{\alpha+1} = \zeta_\alpha \dagger \big\{x_i \mapsto [\![E_i]\!]\iota\zeta_\alpha\big\} \\
&\quad\; \zeta_\sqcup = \mathrm{lfp}\ \zeta_\alpha \\
&\quad \text{in } [\![E_0]\!]\iota(\zeta_\sqcup)\kappa
\end{aligned}$$

Figure 2.2: Semantics of Core TransLucid expressions

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 set of unused dimensions which may be used for dimension allocation.

$$\begin{aligned}[\![E \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa \quad = \quad & \text{let } \delta_i \in \kappa(\rho) \\ & \quad \kappa' = \kappa \dagger \{\rho \mapsto (\kappa(\rho) - \{\delta_1, \ldots, \delta_m\})\} \\ & \quad d_i = [\![E_i]\!]\iota\zeta({}_i\kappa') \\ & \text{in } [\![E_0]\!]\iota(\zeta \dagger \{x_i \mapsto \widehat{\delta_i}\})({}_0\kappa' \dagger \{\delta_i \mapsto d_i\})\end{aligned} \tag{2.10}$$

Figure 2.3: Non-deterministic `wheredim`

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

(2.2) 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 $m > 0$.

(2.3) The current context is returned when `#` appears.

(2.4) 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.

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

(2.6) 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.

(2.7) 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.

(2.8) 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.

(2.9) A sequence of environments ζ_α, $\alpha \in \mathbb{N}$, is defined by creating the initial environment $\zeta_0 = \zeta \dagger \{x_i \mapsto \emptyset\}_{i=1..m}$, 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.

(2.10) Each dimension identifier x_i is mapped in the new environment to $\widehat{\delta_i}$, where δ_i is a dimension non-deterministically chosen from the infinite set of hidden dimensions $\kappa(\rho)$; the δ_i-ordinate is initially the value of expression E_i in context κ.

2.2.5 Soundness of semantics

In the semantics, the allocation of dimensions in the `wheredim` clause is non-deterministic, as is the splitting in each subexpression of the available dimensions from which to allocate. For the semantics to be correct, it is necessary that the meaning of a program not depend on the actual sets of dimensions chosen when `wheredim` clauses are encountered.

Proposition 2. *Let E be an expression, ι be an interpretation, ζ, ζ' be environments, and κ, κ' be contexts, such that $\zeta \equiv \zeta'$ and $(\kappa \triangleleft \{\rho\}) \equiv (\kappa' \triangleleft \{\rho\})$. Then*

$$[\![E]\!]\iota\zeta\kappa \equiv [\![E]\!]\iota\zeta'\kappa'.$$

Proof. By induction over the structure of E. See §2.4.2, p.28. □

2.2.6 Deterministic semantic rules

Since non-deterministic behaviour does not easily lead to an implementation, we present a deterministic method for allocating dimensions, in which a list representing the evaluation tree is passed through the semantics. Since the actual choice of dimensions in a `wheredim` clause is irrelevant to the meaning of a program, we should be able to implement that choice of dimension in any way that we want. Therefore, we present a deterministic means to allocate dimensions to bring TransLucid one step closer to an implementation.

Since we make use of a list in the deterministic semantics, we use the notation below to manipulate lists, and then change the meaning of ${}_i\kappa$ to suit the new semantics.

- We write ν for a list of natural numbers, where $\nu \in \mathbb{N}^*$. The empty list is written ϵ, and the consing of element $i \in \mathbb{N}$ onto list ν is written $i : \nu$.
- We suppose that there is an infinite set of hidden dimensions of the form χ^i_ν, where $i \in \mathbb{N}$ and $\nu \in \mathbb{N}^*$.

Preamble 3. *Let $\rho \in \Delta_O$, and κ be a context such that $\nu = \kappa(\rho)$ is a list of natural numbers. We define*

$${}_i\kappa = \kappa \dagger \{\rho \mapsto i : \nu\}.$$

Definition 13. *Suppose Preamble 1 holds, let*

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

and suppose ${}_i\kappa$ is defined as in Preamble 3. Then the deterministic semantics for E with respect to ι, ζ and κ is given by

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

where the rules for $[\![\cdot]\!]$ are given in Figure 2.2, with the `wheredim` *rule given in Figure 2.4.*

$$\begin{aligned}[\![E \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa \quad = \quad & \text{let } \delta_i = \chi^i_{\kappa(\rho)} \\ & \quad\; d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \\ & \text{in } [\![E_0]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{\delta_i}\}\big)\big({}_0\kappa \dagger \{\delta_i \mapsto d_i\}\big)\end{aligned} \tag{2.11}$$

Figure 2.4: Deterministic `wheredim`

Proposition 2 holds for the semantics in Definition 13, since this is simply a special case of Definition 12, where the choice of hidden dimensions is made deterministically.

2.3 Conclusions

In this chapter, we have presented the Core TransLucid programming language and its denotational semantics. The difficulties solved by this chapter were two-fold: First, what was the exact choice of primitives? Initially, it was thought that the full TransLucid language, with the different kinds of abstraction, presented in the next chapter, was primitive. Then, it was discovered that in fact, so long as the current context was a first-class value, that only a single functional abstraction was necessary.

The second difficulty arose in the allocation of dimensions in `wheredim` clauses, so that each entry is guaranteed to allocate a new dimension. The semantics presented in §2.2 manipulates in the context a special dimension ρ, whose ordinate is an infinite set of dimensions from which the `wheredim` clause can take dimensions. Using the ρ-ordinate upon entry to a `wheredim` clause guarantees that there is no possibility of dimension clash when allocating new dimensions.

However, this is not effective for implementation, since it is non-deterministic, so the second method sets the ρ ordinate to a list encoding the path from the root of the evaluation tree of the expression being evaluated. This approach ensures a deterministic approach to dimension allocation, which is one step closer to an implementation.

Despite presenting a semantics for TransLucid, this chapter makes no attempt at solving implementation issues such as concrete syntax, concrete objects and operations over those objects. Rather, it supposes a set D of atomic types, and a function ι which provides an interpretation of constants. Neither does this chapter give more than a basic presentation of how problems can be solved using TransLucid.

These issues are resolved in later chapters, with Chapter 4 giving a presentation of programming in TransLucid, with a focus on viewing problems from a geometric perspective. Then Chapter 5 presents the infrastructure necessary to move the semantics towards an implementation, by changing manipulations of the environment to manipulations of the context, with Chapter 8 presenting the complete concrete TransLucid system, with syntax and data types available for the user to create and manipulate.

2.4 Proofs of propositions

2.4.1 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. □

2.4.2 Proof of Proposition 2

Proof. Proof by induction on the structure of E. There are three base cases.

$$\begin{aligned} & [\![x]\!]\iota\zeta\kappa \\ &= \zeta(x)(\kappa) \\ &\equiv \zeta'(x)(\kappa') \\ &= [\![x]\!]\iota\zeta'\kappa' \end{aligned}$$

$$\begin{aligned} & [\![{}^m c]\!]\iota\zeta\kappa \\ &= \iota({}^m c) \\ &\equiv \iota({}^m c) \\ &= [\![{}^m c]\!]\iota\zeta'\kappa' \end{aligned}$$

$$
\begin{aligned}
&[\![\texttt{\#}]\!]\iota\zeta\kappa \\
&= \kappa \\
&\equiv \kappa' \\
&= [\![\texttt{\#}]\!]\iota\zeta'\kappa'
\end{aligned}
$$

There are seven inductive cases. We let $n = 2M$, where M is the maximum m occurring in an instance of "$[\cdots]$", ".", "`wheredim`" or "`wherevar`", and suppose that $\kappa(\rho) = \Delta$ and $\kappa'(\rho) = \Delta'$. Then, by assumption, we can split Δ into distinct, infinite sets ${}^0\Delta, \ldots, {}^n\Delta$ and, similarly, Δ' into ${}^0\Delta', \ldots, {}^n\Delta'$. The inductive hypothesis is that for any $j \in 0..n$, $[\![E_j]\!]\iota\zeta({}_j\kappa) \equiv [\![E_j]\!]\iota\zeta'({}_j\kappa')$.

$$
\begin{aligned}
&\left[\!\left[[E_{i0} \leftarrow E_{i1}]_{i=1..m}\right]\!\right]\iota\zeta\kappa \\
&= \left\{[\![E_{i0}]\!]\iota\zeta({}_i\kappa) \;\mapsto\; [\![E_{i1}]\!]\iota\zeta({}_{(i+m)}\kappa)\right\} \\
&\equiv \left\{[\![E_{i0}]\!]\iota\zeta'({}_i\kappa') \;\mapsto\; [\![E_{i1}]\!]\iota\zeta'({}_{(i+m)}\kappa')\right\} \\
&= \left[\!\left[[E_{i0} \leftarrow E_{i1}]_{i=1..m}\right]\!\right]\iota\zeta'\kappa'
\end{aligned}
$$

$$
\begin{aligned}
&[\![\lambda x \to E_0]\!]\iota\zeta\kappa \\
&= \lambda d_a.[\![E_0]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) \\
&\equiv \lambda d_a.[\![E_0]\!]\iota\big(\zeta' \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa' \lhd \{\rho\}\big) \\
&= [\![\lambda x \to E_0]\!]\iota\zeta'\kappa'
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0.(E_i)_{i=1..m}]\!]\iota\zeta\kappa \\
&= \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_i]\!]\iota\zeta({}_i\kappa)\big) \\
&\equiv \big([\![E_0]\!]\iota\zeta'({}_0\kappa')\big)\big([\![E_i]\!]\iota\zeta'({}_i\kappa')\big) \\
&= [\![E_0.(E_i)_{i=1..m}]\!]\iota\zeta'\kappa'
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\zeta\kappa \\
=\ &\text{let } d_0 = [\![E_0]\!]\iota\zeta({}_0\kappa) \\
&\text{in } \begin{cases} [\![E_1]\!]\iota\zeta({}_1\kappa), & d_0 \equiv \mathit{true} \\ [\![E_2]\!]\iota\zeta({}_2\kappa), & d_0 \equiv \mathit{false} \end{cases} \\
\equiv\ &\text{let } d_0 = [\![E_0]\!]\iota\zeta'({}_0\kappa') \\
&\text{in } \begin{cases} [\![E_1]\!]\iota\zeta'({}_1\kappa'), & d_0 \equiv \mathit{true} \\ [\![E_2]\!]\iota\zeta'({}_2\kappa'), & d_0 \equiv \mathit{false} \end{cases} \\
=\ &[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\zeta'\kappa'
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \texttt{ @ } E_1]\!]\iota\zeta\kappa \\
=\ &[\![E_0]\!]\iota\zeta\big({}_0\kappa \dagger [\![E_1]\!]\zeta({}_1\kappa)\big) \\
\equiv\ &[\![E_0]\!]\iota\zeta'\big({}_0\kappa' \dagger [\![E_1]\!]\zeta'({}_1\kappa')\big) \\
=\ &[\![E_0 \texttt{ @ } E_1]\!]\iota\zeta'\kappa'
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa \\
=\ &\text{let } \begin{aligned} \zeta_0 &= \zeta \dagger \{x_i \mapsto \emptyset\} \\ \zeta_{\alpha+1} &= \zeta_\alpha \dagger \{x_i \mapsto [\![E_i]\!]\iota\zeta_\alpha\} \\ \zeta_\sqcup &= \text{lfp } \zeta_\alpha \end{aligned} \\
&\text{in } [\![E_0]\!]\iota(\zeta_\sqcup)\kappa \\
\equiv\ &\text{let } \begin{aligned} \zeta'_0 &= \zeta' \dagger \{x_i \mapsto \emptyset\} \\ \zeta'_{\alpha+1} &= \zeta'_\alpha \dagger \{x_i \mapsto [\![E_i]\!]\iota\zeta'_\alpha\} \\ \zeta'_\sqcup &= \text{lfp } \zeta'_\alpha \end{aligned} \\
&\text{in } [\![E_0]\!]\iota(\zeta'_\sqcup)\kappa' \\
=\ &[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta'\kappa'
\end{aligned}
$$

The `wheredim` case is the only non-trivial one.

$$
\begin{aligned}
&[\![E_0 \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa \\
=\ &\text{let } \delta_i \in \kappa(\rho), i = 1..m \\
&\qquad \kappa'' = \kappa \dagger \{\rho \mapsto (\kappa(\rho) - \{\delta_1, \ldots, \delta_m\})\} \\
&\qquad d_i = [\![E_i]\!]\iota\zeta({}_i\kappa'') \\
&\quad \text{in } [\![E_0]\!]\iota(\zeta \dagger \{x_i \mapsto \widehat{\delta_i}\})({}_0\kappa'' \dagger \{\delta_i \mapsto d_i\}) \\
\equiv\ &\text{let } \delta'_i \in \kappa'(\rho), i = 1..m \\
&\qquad \kappa'' = \kappa' \dagger \{\rho \mapsto (\kappa'(\rho) - \{\delta'_1, \ldots, \delta'_m\})\} \\
&\qquad d_i = [\![E_i]\!]\iota\zeta({}_i\kappa'') \\
&\quad \text{in } [\![E_0]\!]\iota(\zeta' \dagger \{x_i \mapsto \widehat{\delta'_i}\})({}_0\kappa'' \dagger \{\delta'_i \mapsto d_i\}) \\
=\ &[\![E_0 \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta'\kappa'
\end{aligned}
$$

The two middle lines are valid because if ζ and ζ' are equivalent, then $\zeta \dagger \{x_i \mapsto \widehat{\delta_i}\} \equiv \zeta' \dagger \{x_i \mapsto \widehat{\delta'_i}\}$, by definition. Similarly for the κ'' case. Therefore, the induction hypothesis applies.

We have therefore proven that should $\zeta \equiv \zeta'$ and $\kappa \equiv \kappa'$, then $[\![E]\!]\iota\zeta\kappa \equiv [\![E]\!]\iota\zeta'\kappa'$. □

Chapter 3

TransLucid

This chapter presents *TransLucid*, which is the language that is seen by the user, and is defined using a set of syntactic extensions to Core TransLucid. The new constructs include a `where` clause combining the `wheredim` and `wherevar` clauses; as well as abstractions whose body, once evaluated, is sensitive to the context at the time of application, and possibly also sensitive to a named set of dimensions of the context at the time of abstraction.

The extensions are of two kinds. First are the *intension*, *function* and *call-by-value abstractions* (and corresponding applications); these, together with Core TransLucid, define *TransLucid Lite*, and are given a semantics, along with a proof of the validity of their transformation to Core TransLucid. Second are the *call-by-name abstraction* and the `where` clause, which are simply defined as syntactic sugar over TransLucid Lite.

3.1 Syntax

The syntax of TransLucid is given in Figure 3.1, where $\cdots$ is the syntax of Core TransLucid. Each of the new syntactic elements is presented in one of the following sections, and the detailed syntax for the `where` clause is given in §3.6.

$$
\begin{array}{rll}
E ::= & \cdots & \\
\mid & \uparrow\{E, \ldots\}\ E & \textit{intension abstraction} \\
\mid & \downarrow E & \textit{intension application} \\
\mid & \lambda^{\mathrm{b}}\ \{E, \ldots\}\ x \to E & \textit{function abstraction} \\
\mid & \lambda^{\mathrm{v}}\ \{E, \ldots\}\ x \to E & \textit{call-by-value abstraction} \\
\mid & E\,!\,E & \textit{call-by-value application} \\
\mid & \lambda^{\mathrm{n}}\ \{E, \ldots\}\ x \to E & \textit{call-by-name abstraction} \\
\mid & E\ E & \textit{call-by-name application} \\
\mid & E\ \texttt{where}\ \cdots\ \texttt{end} & \texttt{where}\ \textit{clause}
\end{array}
$$

Figure 3.1: Syntax of TransLucid extensions

3.2 Intension abstraction

The ↑ operator allows an entire intension to be wrapped up into a single value, and the ↓ 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 $\{x, y\}$.

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

‘A’

	0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
0	2	3	4	5	⋯
1	3	4	5	6	⋯
2	4	5	6	7	⋯
3	5	6	7	8	⋯
$\texttt{\#}.y\downarrow$	⋮	⋮	⋮	⋮	⋱

‘$\downarrow A$’

	0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
0	2	3	4	5	⋯
1	3	4	5	6	⋯
2	4	5	6	7	⋯
3	5	6	7	8	⋯
$\texttt{\#}.y\downarrow$	⋮	⋮	⋮	⋮	⋱

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 $\downarrow C$.

$$\texttt{var}\ B = \uparrow \{y\}\ (\texttt{\#}.x + \texttt{\#}.y + 2) \qquad \texttt{var}\ C = B\ \texttt{@}\ [y \leftarrow 1]$$

‘B’

0

	0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
	2	3	4	5	⋯

1

	0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
	3	4	5	6	⋯

$\texttt{\#}.y\downarrow$ ⋮

‘C’

	0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
	3	4	5	6	⋯

‘$\downarrow C$’

0	1	2	3	$\overset{\texttt{\#}.x}{\rightarrow}$
3	4	5	6	⋯

As described in §2.2, 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 ‘$\uparrow E$’ can be considered to be syntactic sugar for ‘$\lambda\kappa \rightarrow E\ \texttt{@}\ \kappa$’, and ‘$\downarrow E$’ for ‘$E.\texttt{\#}$’, the application to the current context of the function to which E evaluates.

The semantics of the intension abstraction and application are as follows:

$$\begin{aligned} [\![\uparrow\{E_i\}_{i=1..m}\ E_0]\!]\iota\zeta\kappa &= \text{let } d_i = [\![E_i]\!]\iota\zeta_i\kappa \\ &\quad \text{in } \lambda\kappa_a.[\![E_0]\!]\iota\zeta\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \end{aligned} \tag{3.1}$$

$$[\![\downarrow E_0]\!]\iota\zeta\kappa = ([\![E_0]\!]\iota\zeta_0\kappa)_1\kappa \tag{3.2}$$

3.3 Freezing the context for function abstractions

Similarly to the way in which 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 the creation of the abstraction, we can freeze the ordinates of a set of dimensions at the creation of a function abstraction. For example, the function abstraction A given below creates a different function for each different x-ordinate:

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

‘A’	0					1					$\overset{\texttt{\#}.x}{\to}$
	a					a					
		0	1	2	$\overset{a}{\to}$		0	1	2	$\overset{a}{\to}$	$\cdots$
	0	1	2	3	$\cdots$	1	2	3	4	$\cdots$	

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

‘$A.(\texttt{\#}.x)$’	0	1	2	3	$\overset{\texttt{\#}.x}{\to}$
	0	2	4	6	$\cdots$

The semantics of the λ^{b} function abstraction is as follows:

$$\begin{aligned} [\![\lambda^{\mathrm{b}}\ \{E_i\}_{i=1..m}\ x \to E_0]\!]\iota\zeta\kappa &= \text{let } d_i = [\![E_i]\!]\iota\zeta_i\kappa \\ &\quad \text{in } \lambda d_a.[\![E_0]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)({}_0\kappa \lhd \{\rho, d_i\}) \end{aligned} \tag{3.3}$$

The corresponding function application is the same as the function application in Core TransLucid, and has the same semantics.

3.4 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.

$$\texttt{var}\ A = \lambda^{\mathrm{v}}\ \{x\}\ a \to a + \#.x + \#.y + 1$$

‘A’ | 0 | 1 | $\overset{\#.x}{\to}$

a

	0	1	2	3	$\overset{a}{\to}$
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	$\overset{a}{\to}$
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	$\overset{\#.x}{\to}$
0	1	3	5	7	$\cdots$
1	2	4	6	8	$\cdots$
2	3	5	7	9	$\cdots$
3	4	6	8	10	$\cdots$
$\#.y\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The semantics of call-by-value function abstraction and application are as follows:

$$\begin{aligned}
[\![\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \to E_0]\!]\iota\zeta\kappa &= \text{let } d_i = [\![E_i]\!]\iota\zeta_i\kappa \qquad (3.4)\\
&\quad \text{in } \lambda d_a.\lambda\kappa_a.\\
&\qquad [\![E_0]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\\
[\![E_0\,!\,E_1]\!]\iota\zeta\kappa &= \big([\![E_0]\!]\iota\zeta_0\kappa\big)\big([\![E_1]\!]\iota\zeta_1\kappa\big)_2\kappa \qquad (3.5)
\end{aligned}$$

3.5 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.

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

‘A’	
	d, X

	0	1	2	3	$\overset{\texttt{\#}.d}{\to}$
	$X_{\{d \mapsto 1\}}$	$X_{\{d \mapsto 2\}}$	$X_{\{d \mapsto 3\}}$	$X_{\{d \mapsto 4\}}$	$\cdots$

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:

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

With more syntactic sugar, from the next subsection, the definition of A could be rewritten as the function *next*, described in more detail in Chapter 4.

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

3.6 The `where` clause

Some of the examples presented in §2.1 defined functions using the `fun` keyword, and used `where` clauses in which both dimension and variable identifiers were defined. It turns out that the `where` clause, along with the `dim`, `var` and `fun` keywords, are syntactic sugar for constructs in the abstract syntax defined in Figure 2.1.

For example, the following expression computes the factorial of 5:

$\mathit{fact}.5$
`where`
  `fun` $\mathit{fact}.n = F$
    `where`
      `dim` $d \leftarrow n$
      `var` $F =$ `if` `#`$.d \equiv 0$ `then` 1 `else` `#`$.d \times \big(F$ `@` $[d \leftarrow$ `#`$.d - 1]\big)$ `fi`
    `end`
`end`

and that expression is syntactic sugar for the following expression:

$\mathit{fact}.5$ `wherevar`
  $\mathit{fact} = \lambda^{\mathrm{b}} n \to F$ `wherevar`
    $F =$ `if` `#`$.d \equiv 0$ `then` 1 `else` `#`$.d \times \big(F$ `@` $[d \leftarrow$ `#`$.d - 1]\big)$ `fi`
    `end`
  `wheredim`
    $d \leftarrow n$
  `end`
`end`

3.7 Translations

Definition 14. *A* TransLucid Lite *expression E is a TransLucid expression with no* `where` *clauses or λ^{n} expressions. Translation $\mathcal{W}$ (Figure 3.2) translates TransLucid to TransLucid Lite.*

Definition 15. *Translation $\mathcal{T}$ (Figure 3.3) translates TransLucid Lite to Core TransLucid.*

Proposition 3. *Let E be a TransLucid expression, ι be an interpretation, ζ an environment, and κ a context. Then*

$$[\![\mathcal{T}(\mathcal{W}(E))]\!]\iota\zeta\kappa \equiv [\![E]\!]\iota\zeta\kappa.$$

Proof. See §3.8.2, p.40. □

$$\begin{aligned}
\mathcal{W}(x) &= x \\
\mathcal{W}({}^m c) &= {}^m c \\
\mathcal{W}(\mathtt{\#}) &= \mathtt{\#} \\
\mathcal{W}([E_{i0} \leftarrow E_{i1}]_{i=1..m}) &= [E_{i0}^{\mathcal{W}} \leftarrow E_{i1}^{\mathcal{W}}] \\
\mathcal{W}(\lambda x \to E_0) &= \lambda x \to E_0^{\mathcal{W}} \\
\mathcal{W}(E_0\,.\,(E_i)_{i=1..m}) &= E_0^{\mathcal{W}}\,.\,(E_i^{\mathcal{W}}) \\
\mathcal{W}(\mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2\ \mathtt{fi}) &= \mathtt{if}\ E_0^{\mathcal{W}}\ \mathtt{then}\ E_1^{\mathcal{W}}\ \mathtt{else}\ E_2^{\mathcal{W}}\ \mathtt{fi} \\
\mathcal{W}(E_0\ \mathtt{@}\ E_1) &= E_0^{\mathcal{W}}\ \mathtt{@}\ E_1^{\mathcal{W}} \\
\mathcal{W}(\uparrow\{E_i\}_{i=1..m}\ E_0) &= \uparrow\left\{E_i^{\mathcal{W}}\right\}\ E_0^{\mathcal{W}} \\
\mathcal{W}(\downarrow E_0) &= \downarrow E_0^{\mathcal{W}} \\
\mathcal{W}(\lambda^{\mathrm{b}}\ \{E_i\}_{i=1..m}\ x \to E_0) &= \lambda^{\mathrm{b}}\ \left\{E_i^{\mathcal{W}}\right\}\ x \to E_0^{\mathcal{W}} \\
\mathcal{W}(\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \to E_0) &= \lambda^{\mathrm{v}}\ \left\{E_i^{\mathcal{W}}\right\}\ x \to E_0^{\mathcal{W}} \\
\mathcal{W}(E_0\,!\,E_1) &= E_0^{\mathcal{W}}\,!\,E_1^{\mathcal{W}} \\
\mathcal{W}(\lambda^{\mathrm{n}}\ \{E_i\}_{i=1..m}\ x \to E_0) &= \lambda^{\mathrm{v}}\ \left\{E_i^{\mathcal{W}}\right\}\ x \to E_0^{\mathcal{W}}[x/{\downarrow}x] \\
\mathcal{W}(E_0\ E_1) &= E_0^{\mathcal{W}}\,!\,(\uparrow E_1^{\mathcal{W}}) \\
\mathcal{W}(E_0\ \mathtt{wherevar}\ x_i = E_i\ \mathtt{end}_{i=1..m}) &= E_0^{\mathcal{W}}\ \mathtt{wherevar}\ x_i = E_i^{\mathcal{W}}\ \mathtt{end} \\
\mathcal{W}(E_0\ \mathtt{wheredim}\ x_i \leftarrow E_i\ \mathtt{end}_{i=1..m}) &= E_0^{\mathcal{W}}\ \mathtt{wheredim}\ x_i \leftarrow E_i^{\mathcal{W}}\ \mathtt{end}
\end{aligned}$$

$$\mathcal{W}\begin{pmatrix}
E_0 \\
\mathtt{where} \\
\quad \mathtt{dim}\ x_i \leftarrow E_i, i = 1..\ell \\
\quad \mathtt{var}\ x'_j = E'_j, j = 1..m \\
\quad \mathtt{fun}\ x''_k\ p_{k1}\ x_{k1} \cdots p_{km_k}\ x_{km_k} = E''_k, \\
\qquad k = 1..n \\
\mathtt{end}
\end{pmatrix} = \begin{pmatrix}
E_0^{\mathcal{W}}\ \mathtt{wherevar} \\
\qquad x'_j = \mathcal{W}(E'_j) \\
\qquad x''_k = \mathcal{F}(p_{k1})\ x_{k1} \to \cdots \to \\
\qquad\qquad \mathcal{F}(p_{km_k})\ x_{km_k} \to \mathcal{W}(E''_k) \\
\quad \mathtt{end} \\
\mathtt{wheredim} \\
\quad x_i \leftarrow E_i \\
\mathtt{end}
\end{pmatrix}$$

$$\begin{aligned}
\mathcal{F}(\text{`.'}) &= \lambda^{\mathrm{b}} \\
\mathcal{F}(\text{`!'}) &= \lambda^{\mathrm{v}} \\
\mathcal{F}(\text{` '}) &= \lambda^{\mathrm{n}}
\end{aligned}$$

Figure 3.2: Translation $\mathcal{W}$ from TransLucid to TransLucid Lite $\left(E^{\mathcal{W}} = \mathcal{W}(E)\right)$.

$$
\begin{aligned}
\mathcal{T}(x) &= x \\
\mathcal{T}({}^m c) &= {}^m c \\
\mathcal{T}(\texttt{\#}) &= \texttt{\#} \\
\mathcal{T}([E_{i0} \leftarrow E_{i1}]_{i=1..m}) &= [E_{i0}^{\mathcal{T}} \leftarrow E_{i1}^{\mathcal{T}}] \\
\mathcal{T}(\lambda x \to E_0) &= \lambda x \to E_0^{\mathcal{T}} \\
\mathcal{T}(E_0 \,.\, (E_i)_{i=1..m}) &= E_0^{\mathcal{T}} \,.\, (E_i^{\mathcal{T}}) \\
\mathcal{T}(\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}) &= \texttt{if } E_0^{\mathcal{T}} \texttt{ then } E_1^{\mathcal{T}} \texttt{ else } E_2^{\mathcal{T}} \texttt{ fi} \\
\mathcal{T}(E_0 \texttt{ @ } E_1) &= E_0^{\mathcal{T}} \texttt{ @ } E_1^{\mathcal{T}} \\
\mathcal{T}\big(\uparrow \{E_1, \ldots, E_m\}\ E_0\big) &= \big(\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x_{\kappa'} \\
&\qquad \to E_0^{\mathcal{T}} \texttt{ @ } [x_i \leftarrow x_\kappa.x_i]_{i=1..m} \texttt{ @ } x_{\kappa'}\big).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\texttt{\#} \\
\mathcal{T}\big(\downarrow E_0\big) &= E_0^{\mathcal{T}}.\texttt{\#} \\
\mathcal{T}\big(\lambda^{\mathrm{b}}\{E_1, \ldots, E_m\}\ x \to E_0\big) &= \big(\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x \\
&\qquad \to E_0^{\mathcal{T}} \texttt{ @ } [x_i \leftarrow x_\kappa.x_i]_{i=1..m}\big).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\texttt{\#} \\
\mathcal{T}\big(\lambda^{\mathrm{v}}\{E_1, \ldots, E_m\}\ x \to E_0\big) &= \big(\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x \to \lambda x_{\kappa'} \\
&\qquad \to E_0^{\mathcal{T}} \texttt{ @ } [x_i \leftarrow x_\kappa.x_i]_{i=1..m} \texttt{ @ } x_{\kappa'}\big).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\texttt{\#} \\
\mathcal{T}\big(E_0 \,!\, E_1\big) &= E_0^{\mathcal{T}}.E_1^{\mathcal{T}}.\texttt{\#} \\
\mathcal{T}(E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}) &= E_0^{\mathcal{T}} \texttt{ wherevar } x_i = E_i^{\mathcal{T}} \texttt{ end} \\
\mathcal{T}(E_0 \texttt{ wheredim } x_i \leftarrow E_i \texttt{ end}_{i=1..m}) &= E_0^{\mathcal{T}} \texttt{ wheredim } x_i \leftarrow E_i^{\mathcal{T}} \texttt{ end}
\end{aligned}
$$

Figure 3.3: Translation $\mathcal{T}$ from TransLucid Lite to Core TransLucid $\big(E^{\mathcal{T}} = \mathcal{T}(E)\big)$. Assume that $x_1, \ldots, x_m, x_\kappa, x_{\kappa'}$ are not free in $E_0, \ldots, E_m$.

3.8 Proofs of validity of syntactic translations

3.8.1 Basic equivalences

Proposition 4. *The following equivalences hold.*

$$\begin{aligned}
\big[\!\big[\lambda x \to E \mathbin{@} x\big]\!\big]\iota\zeta\kappa &\equiv [\![E]\!]\iota\zeta \qquad [x \text{ free in } E] \\
[\![E_0.E_1.E_2]\!]\iota\zeta\kappa &\equiv \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_1]\!]\iota\zeta({}_1\kappa)\big)\big([\![E_2]\!]\iota\zeta({}_2\kappa)\big) \\
[\![(\lambda x_1 \to \cdots \lambda x_m \to E_0).E_1.\cdots.E_m]\!]\iota\zeta\kappa &\equiv \text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \\
&\qquad \text{in } [\![E_0]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}\}\big)\big({}_0\kappa \lhd \{\rho\}\big)
\end{aligned}$$

Proof.

$$\begin{aligned}
&\big[\!\big[\lambda x \to E \mathbin{@} x\big]\!\big]\iota\zeta\kappa \\
&= \lambda d_a.[\![E \mathbin{@} x]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) \\
&= \lambda d_a.[\![E]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big(({}_{00}\kappa \lhd \{\rho\}) \dagger [\![x]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)({}_{10}\kappa \lhd \{\rho\})\big) \\
&= \lambda d_a.[\![E]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big(({}_{00}\kappa \lhd \{\rho\}) \dagger d_a\big) \\
&= \lambda d_a.[\![E]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}]\big)(d_a) \\
&= \lambda d_a.[\![E]\!]\iota\zeta(d_a) \\
&= [\![E]\!]\iota\zeta
\end{aligned}$$

$$\begin{aligned}
&[\![E_0.E_1.E_2]\!]\iota\zeta\kappa \\
&= \big([\![E_0.E_1]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_2]\!]\iota\zeta({}_1\kappa)\big) \\
&= \big([\![E_0]\!]\iota\zeta({}_{00}\kappa)\big)\big([\![E_1]\!]\iota\zeta({}_{10}\kappa)\big)\big([\![E_2]\!]\iota\zeta({}_1\kappa)\big) \\
&\equiv \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_1]\!]\iota\zeta({}_1\kappa)\big)\big([\![E_2]\!]\iota\zeta({}_2\kappa)\big)
\end{aligned}$$

$$\begin{aligned}
&\big[\!\big[(\lambda x_1 \to E_0).E_1\big]\!\big]\iota\zeta\kappa \\
&= \big([\![\lambda x_1 \to E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_1]\!]\iota\zeta({}_1\kappa)\big) \\
&= \big(\lambda d_a.[\![E_0]\!]\iota\big(\zeta \dagger \{x_1 \mapsto \widehat{d_a}]\big)({}_{00}\kappa \lhd \{\rho\})\big)\big([\![E_1]\!]\iota\zeta({}_1\kappa)\big) \\
&= \text{let } d_1 = [\![E_1]\!]\iota\zeta({}_1\kappa) \\
&\quad \text{in } [\![E_0]\!]\iota\big(\zeta \dagger \{x_1 \mapsto \widehat{d_1}]\big)\big({}_{00}\kappa \lhd \{\rho\}\big) \\
&\equiv \text{let } d_1 = [\![E_1]\!]\iota\zeta({}_1\kappa) \\
&\quad \text{in } [\![E_0]\!]\iota\big(\zeta \dagger \{x_1 \mapsto \widehat{d_1}\}\big)\big({}_0\kappa \lhd \{\rho\}\big)
\end{aligned}$$

$$
\begin{aligned}
&[\![(\lambda x_1 \to \cdots \lambda x_m \to E_0).E_1.\cdots.E_m]\!]\iota\zeta\kappa \qquad [m > 1]\\
\equiv\ &\text{let } d_1 = [\![E_1]\!]\iota\zeta({}_1\kappa)\\
&\quad \text{in } [\![(\lambda x_2 \to \cdots \lambda x_m \to E_0).E_2.\cdots.E_m]\!]\iota\big(\zeta \dagger \{x_1 \mapsto \widehat{d_1}\}\big)\big({}_0\kappa \lhd \{\rho\}\big)\\
\equiv\ &\text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } [\![E_0]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}\}\big)\big({}_0\kappa \lhd \{\rho\}\big)
\end{aligned}
$$

□

3.8.2 Proof of Proposition 3

Proof. We first prove that if E is a TransLucid Lite expression, that $[\![E^{\mathcal{T}}]\!]\iota\zeta\kappa \equiv [\![E]\!]\iota\zeta\kappa$, which we do by induction over the structure of E. There are three base cases, where E is of the form x, ${}^m c$ or `#`. In each of these three cases, $E^{\mathcal{T}} = E$, so $[\![E^{\mathcal{T}}]\!]\iota\zeta\kappa = [\![E]\!]\iota\zeta\kappa$.

There are twelve inductive cases. We let $n = 2M$, where M is the maximum m occurring in an instance of "$[\cdots]$", "`.`", "`!`", "`wherevar`" or "`wheredim`". The inductive hypothesis is that for any $j \in 0..n, [\![E_j]\!]\iota\zeta({}_j\kappa) \equiv [\![E_j]\!]\iota\zeta'({}_j\kappa')$.

For seven of these cases, the structure of $E^{\mathcal{T}}$ is the same as that of E, and it follows directly from the induction hypothesis that $[\![E^{\mathcal{T}}]\!]\iota\zeta\kappa = [\![E]\!]\iota\zeta\kappa$. These cases are '$[\cdots]$', '$\lambda$', '`.`', '`if`–`then`–`else`', '`@`', "`wherevar`" and "`wheredim`".

The remaining five cases are more difficult, since the transformation $\mathcal{T}$ is non-trivial.

$$
\begin{aligned}
&[\![\mathcal{T}\big(\uparrow \{E_1, \ldots, E_m\}\ E_0\big)]\!]\iota\zeta\kappa\\
=\ &[\![(\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x_{\kappa'}\\
&\quad \to E_0^{\mathcal{T}}\ \texttt{@}\ [x_i \leftarrow x_\kappa.x_i]\ \texttt{@}\ x_{\kappa'}).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\texttt{\#}]\!]\iota\zeta\kappa\\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } [\![\lambda x_{\kappa'} \to E_0^{\mathcal{T}}\ \texttt{@}\ [x_i \leftarrow x_\kappa.x_i]\ \texttt{@}\ x_{\kappa'}]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\big({}_0\kappa \lhd \{\rho\}\big)\\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0^{\mathcal{T}}\ \texttt{@}\ [x_i \leftarrow x_\kappa.x_i]]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\kappa_a\\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0^{\mathcal{T}}]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\big({}_0\kappa_a \dagger [\![[x_i \leftarrow x_\kappa.x_i]]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)_1\kappa_a\big)\\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0^{\mathcal{T}}]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0^{\mathcal{T}}]\!]\iota\zeta\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\\
\equiv\ &\text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0]\!]\iota\zeta\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\\
\equiv\ &\text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa)\\
&\quad \text{in } \lambda\kappa_a.[\![E_0]\!]\iota\zeta\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big)\\
=\ &[\![\uparrow \{E_1, \ldots, E_m\}\ E_0]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{T}(\downarrow E_0)]\!]\iota\zeta\kappa \\
=\ &[\![E_0^{\mathcal{T}}.\#]\!]\iota\zeta\kappa \\
\equiv\ &\big([\![E_0^{\mathcal{T}}]\!]\iota\zeta({}_0\kappa)\big)\big([\![\#]\!]\iota\zeta({}_1\kappa)\big) \\
\equiv\ &\big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)({}_1\kappa) \\
=\ &[\![\downarrow E_0]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{T}(\lambda^{\mathrm{b}}\{E_1,\ldots,E_m\}\ x \to E_0)]\!]\iota\zeta\kappa \\
=\ &[\![(\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x \\
&\quad \to E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\#]\!]\iota\zeta\kappa \\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa) \\
&\text{in } [\![\lambda x \to E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) \\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) \\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0^{\mathcal{T}}]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big) \\
&\quad \big(({}_{00}\kappa \lhd \{\rho\}) \dagger [\![[x_i \leftarrow x_\kappa.x_i]]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)\big({}_{10}\kappa \lhd \{\rho\}\big)\big) \\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0^{\mathcal{T}}]\!]\iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)\big({}_{00}\kappa \lhd \{d_i, \rho\}\big) \\
\equiv\ &\text{let } d_i = [\![E_i^{\mathcal{T}}]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0^{\mathcal{T}}]\!]\iota(\zeta \dagger \{x \mapsto \widehat{d_a}\})\big({}_{00}\kappa \lhd \{d_i, \rho\}\big) \\
\equiv\ &\text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0]\!]\iota(\zeta \dagger \{x \mapsto \widehat{d_a}\})\big({}_{00}\kappa \lhd \{d_i, \rho\}\big) \\
\equiv\ &\text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.[\![E_0]\!]\iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa \lhd \{d_i, \rho\}\big) \\
=\ &[\![\lambda^{\mathrm{b}}\{E_1,\ldots,E_m\}\ x \to E_0]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{T}\big(\lambda^{\mathrm{v}}\{E_1,\ldots,E_m\}\ x \to E_0\big)\rrbracket \iota\zeta\kappa \\
=\ &\llbracket (\lambda x_1 \to \cdots \to \lambda x_m \to \lambda x_\kappa \to \lambda x \to \lambda x_{\kappa'} \\
&\quad \to E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]).E_1^{\mathcal{T}}.\cdots.E_m^{\mathcal{T}}.\#\rrbracket \iota\zeta\kappa \\
\equiv\ &\text{let } d_i = \llbracket E_i^{\mathcal{T}}\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \llbracket \lambda x \to \lambda x_{\kappa'} \to E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]\ @\ x_{\kappa'}\rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}\}\big)\big({}_0\kappa \lhd \{\rho\}\big) \\
\equiv\ &\text{let } d_i = \llbracket E_i^{\mathcal{T}}\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0^{\mathcal{T}}\ @\ [x_i \leftarrow x_\kappa.x_i]\rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)\kappa_a \\
\equiv\ &\text{let } d_i = \llbracket E_i^{\mathcal{T}}\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0^{\mathcal{T}}\rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big) \\
&\quad \big({}_0\kappa_a \dagger \llbracket [x_i \leftarrow x_\kappa.x_i]\rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)({}_1\kappa_a)\big) \\
\equiv\ &\text{let } d_i = \llbracket E_i^{\mathcal{T}}\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0^{\mathcal{T}}\rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{d_i}, x_\kappa \mapsto \widehat{\kappa}, x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
\equiv\ &\text{let } d_i = \llbracket E_i^{\mathcal{T}}\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0^{\mathcal{T}}\rrbracket \iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
\equiv\ &\text{let } d_i = \llbracket E_i\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0\rrbracket \iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big({}_0\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
\equiv\ &\text{let } d_i = \llbracket E_i\rrbracket \iota\zeta({}_i\kappa) \\
&\text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0\rrbracket \iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
=\ &\llbracket \lambda^{\mathrm{v}}\{E_1,\ldots,E_m\}\ x \to E_0\rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{T}\big(E_0\ !\ E_1\big)\rrbracket \iota\zeta\kappa \\
=\ &\llbracket E_0^{\mathcal{T}}.E_1^{\mathcal{T}}.\#\rrbracket \iota\zeta\kappa \\
\equiv\ &\big(\llbracket E_0^{\mathcal{T}}\rrbracket \iota\zeta({}_0\kappa)\big)\big(\llbracket E_1^{\mathcal{T}}\rrbracket \iota\zeta({}_1\kappa)\big)\big(\llbracket\#\rrbracket \iota\zeta({}_2\kappa)\big) \\
\equiv\ &\big(\llbracket E_0\rrbracket \iota\zeta({}_0\kappa)\big)\big(\llbracket E_1\rrbracket \iota\zeta({}_1\kappa)\big)({}_2\kappa) \\
=\ &\llbracket E_0\ !\ E_1\rrbracket \iota\zeta\kappa
\end{aligned}
$$

Hence, by the inductive hypothesis, if E is a TransLucid Lite expression, $\llbracket E^{\mathcal{T}}\rrbracket \iota\zeta\kappa \equiv \llbracket E\rrbracket \iota\zeta\kappa$. Since $\mathcal{W}$ is a purely syntactic manipulation, it follows that for a general TransLucid expression E, that $\llbracket \mathcal{T}(\mathcal{W}(E))\rrbracket \iota\zeta\kappa \equiv \llbracket E\rrbracket \iota\zeta\kappa$. □

Chapter 4

The Geometrical View

This chapter presents a selection of programming examples in TransLucid, starting from a set of functions dating back to the original Lucid of the 1970s, followed by a presentation of the TransLucid solution to several standard programming problems. The problems presented here are viewed from a geometric perspective, which allows the visualisation of the arrays defined by a TransLucid program. Visualising these arrays has three benefits: 1) the general principles of programming in TransLucid can be better understood, 2) solving a specific problem in TransLucid can be made easier by visualising the arrays that are built, and 3) standard programming problems can be understood in a new geometric manner, which may provide fresh insight into these problems.

Several of the examples presented in this chapter come from previous publications on Lucid, and we present both the original Lucid version and the TransLucid version. By comparing the different ways of solving these problems, it can be seen that the use of higher-order functions and explicit dimensionality, both provided by TransLucid, make the solutions much cleaner, and easier to both read and specify.

The examples presented here are in some sense a *TransLucid 101*, giving a glimpse of what is possible with multidimensional programming. The programming methodology presented here is a result of solving these problems, but will evolve as new problems are also considered from a geometric point of view.

4.1 Intensional functions from Indexical Lucid

Before presenting complete programming examples, we first present several functions that come from Indexical Lucid. Each of these functions takes a dimension as parameter, and the original Lucid had corresponding functions that manipulated an implicit dimension, which was called `time` (not to be confused with the `time` dimension to be presented in §8.11).

We begin with the following five functions:

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

The function *index* takes a call-by-value parameter d, so it evaluates its body in the context in which the function is applied, and so returns the d-ordinate, plus one, at the point of call. The reason for defining *index* is that it is common to use the index of the entry in the second argument to *fby* ("followed by"), which decreases the ordinate by one, hence the plus-one to get back to its original value.

The function *at* evaluates its call-by-name argument X, changing the current context so that the d-ordinate becomes v. In Indexical Lucid, this function was written using the `@` symbol, which we use in TransLucid to change the ordinates of multiple dimensions at a time. We retain the single-dimension context-change function for brevity of code.

The next three functions, *first*, *next* and *fby*, are analogous to the functions *hd*, *tail* and *cons* for lists. The function *first* takes the zeroth entry of its parameter X in the d direction. If the rank (dimensionality) of X contains the dimension d, then the rank of the result does not.

The function *next* shifts its argument along by one. It can be viewed in two ways: in the extensional view, it produces an array that is its parameter shifted one to the left in the d direction; in the intensional view, in a given context in which the d-ordinate is n, expression $\mathit{next}.d\ X$ will access the $(n+1)$-st entry of X, in the d direction.

The function *fby* takes two intensions, and a direction. It defines an intension whose zeroth entry in the d direction is that of X, then the remaining entries are taken from Y starting at zero.

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	$\overset{\texttt{\#}.d}{\rightarrow}$
$\mathit{index}\ !\ d$	1	2	3	4	5	6	$\cdots$
$\mathit{at}.d.v\ A$	a_v	a_v	a_v	a_v	a_v	a_v	$\cdots$
$\mathit{first}.d\ A$	a_0	a_0	a_0	a_0	a_0	a_0	$\cdots$
$\mathit{next}.d\ A$	a_1	a_2	a_3	a_4	a_5	a_6	$\cdots$
$\mathit{fby}.d\ A\ B$	a_0	b_0	b_1	b_2	b_3	b_4	$\cdots$

4.2 Filters

Lucid had several standard functions which operated as *filters*, taking elements from a stream under certain conditions. These have been adapted to TransLucid and filter ele-

ments from intensions. The first filter is *wvr* ("whenever"), and is defined as follows:

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

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 the result of $\mathit{wvr}.d\ A\ B$ is as follows

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

The function *asa* ("as soon as") takes two intensions X and Y as arguments, and gives the first element from X for which the corresponding element from Y is `true`. It is defined, simply using *first* and *wvr*, as follows:

$$\texttt{fun}\ \mathit{asa}.d\ X\ Y = \mathit{first}.d\ (\mathit{wvr}.d\ X\ Y)$$

The last filter is the function *upon*, which takes as argument intensions X and Y, and produces an intension whose elements are copies of elements from X whenever Y is false. It is named after its behaviour of advancing upon X only when Y is true. Its definition is as follows:

$$\begin{aligned}&\texttt{fun}\ \mathit{upon}.d\ X\ Y = X\ \texttt{@}\ [d \leftarrow Z]\\ &\texttt{where}\\ &\quad\texttt{var}\ Z = \mathit{fby}.d\ 0\ (\texttt{if}\ Y\ \texttt{then}\ Z + 1\ \texttt{else}\ Z\ \texttt{fi})\\ &\texttt{end}\end{aligned}$$

For example, for variable A defined above, and if $B = \langle F, T, F, F, T, \ldots\rangle$ defined above, the expression $\mathit{upon}.d\ A\ B$ produces the following:

	0	1	2	3	4	5	$\overset{\#.d}{\rightarrow}$
$\mathit{upon}.d\ A\ B$	a_0	a_0	a_1	a_1	a_1	a_2	$\cdots$

4.3 Embedding finite data structures into infinite ones

In TransLucid, all intensions are infinite, but it is often necessary to deal with finite data. The standard way to carry out a computation with finite data is to embed the finite region into an infinite *sea* of 0's, 1's, ∞'s, or possibly some other value, depending on the problem; usually, the value chosen is the neutral element for the computation being carried out. To achieve this, we define a one-dimensional function, $\mathit{default}_1.d.m.n.v\ X$, and a two-dimensional function, $\mathit{default}_2.d_1.d_2.m_1.n_1.m_2.n_2.v\ X$. The function $\mathit{default}_1$ takes as input a dimension d and two integers m and n, and defines an intension whose values are X between m and n, and v everywhere else. The function $\mathit{default}_2$ defines a two-dimensional equivalent, which defines a similar region in two-dimensional space.

The function $default_1$ is defined as follows:

$$default_1.d.m.n.v\ X = \texttt{if}\ m \leq \texttt{\#}.d\ \texttt{\&\&}\ \texttt{\#}.d \leq n\ \texttt{then}\ X\ \texttt{else}\ v\ \texttt{fi}$$

For example, the variables $C = default_1.d.1.5.1$ (#.d), and $D = default_1.d.1.5.1\ A$, can be visualised as follows:

	0	1	2	3	4	5	6	7	$\overset{\texttt{\#}.d}{\rightarrow}$
C	1	1	2	3	4	5	1	1	$\cdots$
D	1	a_1	a_2	a_3	a_4	a_5	1	1	$\cdots$

The function $default_2$ is defined as follows:

$$\begin{aligned} default_2.d_1.m_1.n_1.d_2.m_2.n_2.v\ X = \texttt{if}\ & m_1 \leq \texttt{\#}.d_1\ \texttt{\&\&}\ \texttt{\#}.d_1 \leq n_1\ \texttt{\&\&} \\ & m_2 \leq \texttt{\#}.d_2\ \texttt{\&\&}\ \texttt{\#}.d_2 \leq n_2 \\ & \texttt{then}\ X \\ & \texttt{else}\ v \\ & \texttt{fi} \end{aligned}$$

(In Chapter 8, a construct reminiscent of pattern matching is introduced, making the above two definitions simpler.)

For example, the expression $default_2.a.1.3.b.1.3.0$ (#.a + #.b) defines the following intension:

	0	1	2	3	4	5	$\overset{\texttt{\#}.a}{\rightarrow}$
0	0	0	0	0	0	0	$\cdots$
1	0	2	3	4	0	0	$\cdots$
2	0	3	4	5	0	0	$\cdots$
3	0	4	5	6	0	0	$\cdots$
4	0	0	0	0	0	0	$\cdots$
5	0	0	0	0	0	0	$\cdots$
$\overset{\texttt{\#}.b}{\downarrow}$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

4.4 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
```

4.5 Transposing and rotating

In *Multidimensional Programming*, the following expression was presented to define the transpose of a matrix A that varies in dimensions x and y:

```
realign.t,x (realign.x,y (realign.y,t (A)))
where
  dimension t
  realign.a,b (C) = C @.a #.b
end
```

If X is an intension that varies in dimension a, then *realign*.a,b (X) produces an intension that varies in dimension b, with the same entries that X has in direction a. To carry out a transposition, since the `@` operator only allowed the changing of one dimension at a time, it was necessary to declare a third dimension, then use *realign* to rotate the array A three times to swap the appropriate dimensions.

In TransLucid, we call the *realign* function *rotate*, and it is defined similarly as follows:

$$\texttt{fun}\ \mathit{rotate}.d.d'\ X = X\ \texttt{@}\ [d \leftarrow \#.d']$$

However, since TransLucid can change the ordinates of multiple dimensions at the same time, we can do transposition in one go:

$$\texttt{fun}\ \mathit{transpose}.d_1.d_2\ X = X\ \texttt{@}\ [d_1 \leftarrow \#.d_2, d_2 \leftarrow \#.d_1]$$

4.6 Folding left $\neq$ folding right

In functional programming, a fold operation takes as input a list, a function and a neutral element, and produces as output an object that is produced by iteratively applying the

function to every element, starting with the neutral element.

In TransLucid, rather than lists, we have intensions, so we will define a fold over the elements of an intension, in a particular direction. Furthermore, we can make the fold infinite, leaving it up to the user to choose at which point in the resulting intension the desired value should be pulled out, thereby giving them the choice as to how many elements to fold.

The left-fold function is defined as follows:

$$
\begin{array}{l}
\texttt{fun}\ \mathit{foldl}.d.f.z\ X = F \\
\texttt{where} \\
\quad \texttt{var}\ F = \mathit{fby}.d\ z\ (f\ !\ F\ !\ X) \\
\texttt{end}
\end{array}
$$

Here, we define a variable F, whose entries are the result of folding the previous entries of the input X. The zeroth entry is the initial element z, then every subsequent entry is the result of applying the function f to the previous entry of the fold and the previous entry of the input X. The result is F, allowing zero elements of X to be folded, the result of which is z.

The *foldl* function can be visualised by the following table

F	0	1	2	3
	z	$f(z, x_0)$	$f(f(z, x_0), x_1)$	$f(f(f(z, x_0), x_1), x_2)$

Like the left fold, the right fold takes an array X, and defines an array which is the result of folding elements up to the corresponding element of X; however, the right fold starts at the last element and works its way back. We define a function $\mathit{foldr}.d.f.z\ X$, which makes use of a two-dimensional array, and the result is the zeroth column of that array. The zeroth row is made up of the neutral element z, the first row is z, except for its zeroth entry, which is the result of applying f to x_0 and z. The second row is all z, except for the first and zeroth entries, which are the result of applying f to x_1 and z, and applying f to x_0 and the first entry, respectively.

The right-fold operation can be visualised by the following table, which defines the two dimension array F. The result is the zeroth column.

F	0	1	2	3	4	$\overset{\#.d'}{\rightarrow}$
0	$\boxed{z}$	z	z	z	z	$\cdots$
1	$\boxed{f(x_0, z)}$	z	z	z	z	$\cdots$
2	$\boxed{f(x_0, f(x_1, z))}$	$f(x_1, z)$	z	z	z	$\cdots$
3	$\boxed{f(x_0, f(x_1, f(x_2, z)))}$	$f(x_1, f(x_2, z))$	$f(x_2, z)$	z	z	$\cdots$
$\#.d\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The function *foldr* is defined as follows:

```
fun foldr.d.f.z X = F
where
  dim d' ← 0
  var X' = rotate.d.d' X
  var F = if #.d' ≥ #.d then z
          else f ! X' ! (next.d' F)
          fi
end
```

We can also define a "left-fold-whenever" function, which carries out a left fold, but only includes the elements for which a specified condition is true. Therefore, we define *foldl_wvr.d.f.v* X Y, that folds array X in direction d with function f, whenever Y is true. When Y is false, the previous value is used. Its definition is as follows:

```
fun foldl_wvr.d.f X Y = F
where
  var F = fby.d X if Y then f ! F ! (next.d X) else F fi
end
```

4.7 Matrix multiplication

This section presents the multiplication of two matrices, $X \times Y$, in TransLucid. If the matrix X has m rows and k columns, and Y has k rows and n columns, then their multiplication is defined. To multiply the two matrices, we consider how each element of the resulting matrix is computed. The resulting matrix will have m rows and n columns, and the entry in row i and column j is computed by summing the multiple of each pair of entries from row i of X, and column j of Y. Figure 4.1 shows that cell $(0, 1)$ of the multiplication of a 2×3 and a 3×3 matrix is computed by summing the products of the zeroth row of the former and the first column of the latter.

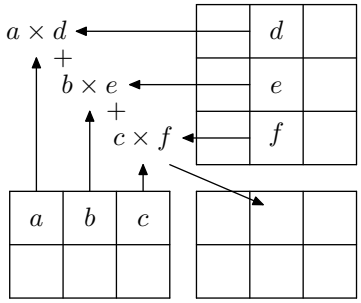


Figure 4.1: Cell $(0, 1)$ of a matrix multiplication

Matrix multiplication can of course be viewed geometrically, which aids in understanding how it is implemented in TransLucid. One can imagine that the columns of the matrices are a horizontal dimension in space, and the rows of the matrices are a vertical

dimension in space. The two matrices can then be rotated into a third dimension, where the columns of matrix X and the rows of matrix Y are moved to the third dimension. The matrix multiplication is computed in an $m \times n \times k$ parallelepiped, every element of which is the product of the corresponding elements from the rotated X and Y. The final result is then the sum of k elements along the third dimension.

Figure 4.2 shows the geometrical view of matrix multiplication. The original row and column dimensions go down the page and to the right respectively, and the third dimension comes out of the page. The rotated X matrix is on the left, and the rotated Y matrix is on the bottom. The result lies in the plane described by the original row and column dimensions. The cells used to compute one entry of the result are shown.

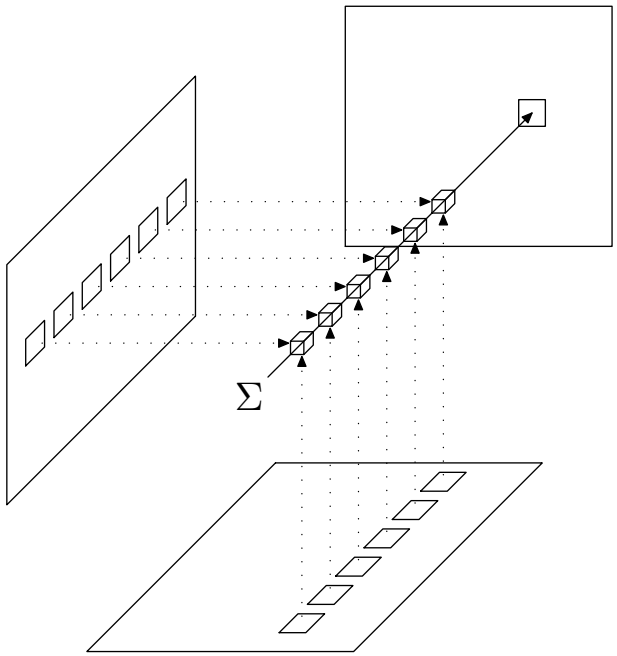


Figure 4.2: Geometric view of matrix multiplication

To define matrix multiplication in TransLucid, we define the function *matrix_multiply*, which takes as argument the row and column dimensions (d_r and d_c) of the matrices, the length k which is the width of the first and the height of the second, and the two matrices X and Y. Its definition is as follows:

```
fun matrix_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 = (foldl.d.plus.0.k Z) @ [d ← k]
end
```

We allocate a new dimension d, in which to rotate both arrays X and Y, then arrays X' and Y' are the result of rotating the d_c (resp. d_r) dimension of X (resp. Y) into the d direction. Then, the array Z is simply the three-dimensional cube that results from multiplying every corresponding element from X' and Y', and W is the result of summing back along the d direction, for k elements, which collapses the result back down to a two-dimensional array which is the result of multiplying the two input matrices.

4.8 Root mean square

The original Lucid book presented a program to compute the root mean square of a stream using Newton's method of approximation [43, p.1]. We present the original program below, and then the TransLucid solution to the same problem.

In the original Lucid solution, the program was given as a single expression which computed the root mean square of an input stream a, where each of the subsidiary functions were defined in the `where` clause as part of that expression. The program is defined as follows:

$$
\begin{array}{l}
sqroot(avg(square(a))) \\
\texttt{where} \\
\quad square(x) = x * x; \\
\quad avg(y) = mean \\
\quad \texttt{where} \\
\quad\quad n = 1 \texttt{ fby } n + 1; \\
\quad\quad mean = \texttt{first } y \texttt{ fby } mean + d; \\
\quad\quad d = (\texttt{next } y - mean)/(n + 1); \\
\quad \texttt{end} \\
\quad sqroot(z) = approx \texttt{ asa } err < 0.0001 \\
\quad \texttt{where} \\
\quad\quad Z \texttt{ is current } z; \\
\quad\quad approx = Z/2 \texttt{ fby } (approx + Z/approx)/2; \\
\quad\quad err = abs(square(approx) - Z); \\
\quad \texttt{end} \\
\texttt{end}
\end{array}
$$

We start with the definition of the function *avg*. It takes as argument a stream y, and returns a stream whose elements are the average of all of the elements of y up to that point. Since Lucid had no means to access the current element being defined, it was necessary to define the variable n, which is just a stream of natural numbers. In TransLucid, we can simply write $index\,!\,d$ to define the equivalent array varying in the d dimension.

The variable *mean* is computed incrementally. The mean of one number is just that number, hence `first` y for the first entry of *mean*. Then every subsequent value is computed by updating *mean* with the appropriate value, which is given by the stream d. To compute the value with which to update the mean, we subtract the old mean from the next value of the stream y, then divide it by its position in the stream. Since `fby`

computes its right-hand side at the previous index, it is necessary to use `next` y and $n+1$.

For the definition of *sqroot*, we use Newton's method of approximation. Each entry in the stream *approx* is the next approximation to the square root. However, for *approx*, we want a stream of approximations for every element of a. In TransLucid, we would simply allocate a new dimension, but Lucid did not have explicit access to dimensions, so the keyword `is current` was used to "freeze" the stream z at its current position, and that element would be referred to as Z. The remainder of the definition of *approx* is the standard definition using Newton's method. Finally, the variable *err* computes how far from the real value the approximation is, and in this definition, we take the value as soon as the error is less than 0.0001.

In the TransLucid solution below, we present the subsidiary functions as they would be given to the interpreter. Expression *rms.d A* defines an array whose entries are the root mean square of the entries of A up to the entry being defined, where A and the result vary in dimension d.

```
fun sqroot.x = asa.d approx (err < floatmp"0.0001")
where
  dim d ← 0
  var approx = fby.d (x/floatmp"2") ((approx + x/approx)/floatmp"2")
  var err = abs ! (approx × approx − x)
end

fun avg.d X = mean
where
  var n = index ! d
  var mean = fby.d X (mean + a)
  var a = (next.d X − mean)/(convert.floatmp.(n + 1))
end

fun rms.d X = sqroot.(avg.d (X × X))
```

The differences between the TransLucid solution and the Lucid solution seem small, however, they are important. A design decision of TransLucid is to not have implicit numeric conversion. Also, there are no floating-point literals, so floating-point numbers must be input explicitly using the syntax for type literals (the `floatmp` type is presented in detail in Chapter 9).

In TransLucid, as mentioned earlier, the variable n can be defined by directly accessing the index in which it varies. In addition, since we can declare new dimensions as required, the `is current` operator becomes unnecessary; we simply allocate a new dimension d in the function *sqroot*, and then *approx* simply varies in that dimension. We also do not define *sqroot* over an array, since we can pass individual elements, and the behaviour of the Lucid program can still be duplicated, by simply writing *sqroot.X*, where X is some intension. The result is then an intension which varies in the same dimensions as X, since base functions are applied pointwise to all elements.

4.9 Divide and conquer

A divide-and-conquer solution to a problem consists of breaking up the problem into a number of (approximately) equal-size subproblems, solving these subproblems, and combining the results thereof to produce the final result. This section presents a general framework for binary divide-and-conquer solutions in TransLucid.

In binary divide-and-conquer problems, since the problem is being divided in half each time, the solution to the problem will be found after $\lceil \log_2(n+1) \rceil$ division iterations. Therefore, we need a function to compute the ceiling of the logarithm of a number. We observe that the powers of 2, starting from zero, are $1, 2, 4, 8, 16$ and so on, and the logarithm of any power of two is its position in the list of powers of two. Therefore, the logarithm of any number between two powers of two is a number between the respective position of the powers. Since we want the ceiling, the logarithm of a number n is the position of the first power of two that is greater than or equal to n. To encode that calculation in TransLucid, we say that we want the index of the power of two *as soon as* the power of two is greater than or equal to n. The function *ilog*.n is therefore defined as follows:

$$
\begin{array}{l}
\texttt{fun}\ \mathit{ilog}.n = \mathit{asa}.d\ (\#.d)\ (\mathit{double} \geq n) \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow 0 \\
\quad \texttt{var}\ \mathit{double} = \mathit{fby}.d\ 1\ (\mathit{double} \times 2) \\
\texttt{end}
\end{array}
$$

Here we define an array *double* in direction d, which is the powers of 2, and we use the *asa* function. We are specifying that we want the index ($\#.d$), at the first position that the array *double* is greater than or equal to n.

The calculation of a factorial can be solved in a divide-and-conquer manner, and this solution is more suitable for large numbers. To compute the factorial of n, what is needed is to multiply every number from 1 to n together—the order is irrelevant. So, to compute factorial in a divide-and-conquer manner, we multiply pairs of numbers from the sequence 1 to n, then multiply pairs of the results, and so on until there is one number left, which will be after $\lceil \log_2 n \rceil$ iterations.

To carry out the above process in TransLucid, we can use *fby* to create an infinite array of rows, where each row of entries is the multiple of the appropriate pairs of elements from the previous row. The difficulty then is how to make the first row. If the number of elements in the first row is not a power of 2, then if we choose the first row incorrectly, we could be computing the wrong thing. The solution is that every other element in the first row should be the number 1, since 1 is the neutral element for multiplication. As a

result, the table for the computation of the factorial of 8 looks as follows:

$F_{n=8}$	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	$\cdots$
2	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	$\cdots$
$\#.d'\downarrow$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We use two local dimensions, d and d', where each row in direction d' is an array of multiplications which varies in d. The following list shows the relevant calculations.

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

For the purposes of divide-and-conquer, we also define the functions $lPair.d\ X$ and $rPair.d\ X$, which retrieve the left and right entries (from a pair) and combine them to make each subsequent row. For example, entry zero is created by combining entries zero and one, so for entry zero, $lPair$ gives 0 and $rPair$ gives 1. They are defined as follows:

$$\texttt{fun}\ lPair.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d \times 2]$$
$$\texttt{fun}\ rPair.d\ X = X\ \texttt{@}\ [d \leftarrow \#.d \times 2 + 1]$$

The function $fact.n$ below defines the factorial of the input n using divide-and-conquer, using $default_1$ to generate the initial row:

```
fun fact.n = F
where
  dim d ← 0
  dim d' ← ilog.(n + 1)
  var F = fby.d' (default_1.d.1.n.1 (#.d)) (lPair.d F × rPair.d F)
end
```

4.10 Powers

In the 1995 book *Multidimensional Programming*, a program was presented to compute the N-th powers of any natural number N. After presenting how the problem is solved geometrically, we will present their solution here, and then the TransLucid solution.

First we start with the observation that the running sum of the odd positive integers $(1, 3, 5, \ldots)$ produces the squares $(1, 4, 9, 16, \ldots)$. We make a generalisation of this procedure to produce any power. If we start with a number N, then drop every N-th number, and make a running sum of the result, then repeat that procedure for $N - 1$ until we get to 1, we will get the N-th powers. We use two variables, varying in two dimensions: n, signifying that we are dropping every n-th value, which varies in dimension a; and *seq*, varying in dimensions a and b, which is the sequence of numbers in direction b that are the result of dropping every n-th and summing the result.

For example, we will visualise here the tables produced for $N = 4$, where the boxes enclose the elements that are dropped to produce the next row.

n		*seq*	0	1	2	3	4	5	6	7	8	9	10	11	$\overset{\#.b}{\rightarrow}$
0	4	0	1	2	3	[4]	5	6	7	[8]	9	10	11	[12]	…
1	3	1	1	3	[6]	11	17	[24]	33	43	[54]	67	81	[96]	…
2	2	2	1	[4]	15	[32]	65	[108]	175	[256]	369	[500]	671	[864]	…
3	1	3	1	16	81	256	625	1296	2401	4096	6561	…			
$\#.a\downarrow$	⋮	$\#.a\downarrow$	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱			

The variable n holds the n-th element to drop each time around. The variable *seq* is the sequence of numbers at each iteration, with the row when n is 1 being the N-th powers. The table visualising *drop.b.n seq* is the result of dropping every n-th element of *seq*, and each row of *seq* is produced by summing the previous row of *drop.b.n seq*.

drop.b.n seq	1	2	3	4	5	6	7	8	9	10	11	12	$\overset{\#.b}{\rightarrow}$
0	1	2	3	5	6	7	9	10	11	13	14	15	…
1	1	3	11	17	33	43	67	81	113	131	171	193	…
2	1	15	65	175	369	671	1105	1695	2465	3439	4641	6095	…
$\#.a\downarrow$	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

The program, as presented in *Multidimensional Programming*, is presented in Figure 4.3. This program is presented as a single expression defining the N-th powers for some number N. Here, the variable *seq* varies in a dimension b, and dimension a is declared as a local dimension in the `where` clause. The function *drop* simply uses *wvr* to choose only the elements that are needed. Then *seq* is defined as the first row in direction b being the positive integers in direction a, and every subsequent row being the result of the sum of dropping the n-th element of the previous row.

In previous versions of Lucid, some of the standard functions were built in, and in fact, before Multidimensional Lucid, the functions could not be defined in the language, because dimensions could not be manipulated. Therefore, those functions could be recognised by the parser, and written using infix notation. So where the above program writes y `fby`.d `next`.d $y + s$, we write in TransLucid, *fby.d y* (*next.d y* + *s*).

It is important to realise that in the version of Lucid used here, Indexical Lucid, there is no partial application of functions, and that functions take two kinds of parameters.

```
seq @.a (N − 1)
where
  dimension a;
  seq = posint fby.a runningSum.b(drop.b(seq, n))
  posint = next.b #.b;
  runningSum.d(y) = s
  where
    s = y fby.d next.d y + s;
  end
  drop.c(U, m) = U wvr.c wanted
  where
    wanted = (#.c + 1) mod m ne 0;
  end
  n = N fby.a n − 1;
end
```

Figure 4.3: Powers program in Indexical Lucid

The parameters that appear after the period can only be dimensions, and if they are more than one, they are separated by commas. The parameters that appear in parentheses are streams.

The TransLucid version is as follows:

```
fun drop.d.n X = wvr.d X ((index ! d) % n ≢ 0)
fun powers.b ! N = seq
where
  dim a ← N − 1
  var seq = fby.a posint (foldl.b.plus.0 (drop.b.n seq))
  var posint = index ! b
  var n = N − #.a
end
```

Thus, we define the two functions *drop* and *powers*. Using *powers*, an array varying in dimension d defining the powers of a number N is given by the expression $powers.b\,!\,N$. We use a call-by-value function for the parameter N because the result of *powers* is an array, so a base function would not suffice.

Since TransLucid supports higher-order functions, we can define the running-sum part of the computation as a fold using the *plus* function. We can also initialise dimensions when they are declared in a `where` clause, so rather than writing $seq\ @.a\ (N-1)$, we can initialise the dimension when it is declared with $\texttt{dim}\ a \leftarrow N-1$. Alternatively, we could have written $seq\ @\ [a \leftarrow N-1]$.

4.11 Dynamic programming

Dynamic programming is similar to divide and conquer, except that the division into smaller subproblems is rarely equal. As a result, only a small piece of the problem is solved from each division. Then, the solution to that smaller problem is remembered, so that any of the bigger problems can be solved by recalling the results of smaller subproblems.

The knapsack problem is a good example of a dynamic programming problem. There are several versions of the knapsack problem, here we will look at the unbounded knapsack problem, with integer weights and values. The problem is to determine, from a list of objects with weights w_i and values v_i, the greatest value that can be achieved by choosing any number of each object, where the sum of the weights of each object does not exceed some specified weight W.

To solve the knapsack problem, we define an array K, where entry $K[w]$ is the maximum value that we can make out of objects that weigh no more than w. $K[0]$ is obviously 0, as having nothing in the knapsack has zero value. To compute every other entry, we notice that the optimum value of objects weighing less than some weight w is the value of some optimum object v_o, with weight w_o, whatever that object may be, plus the optimum value for a smaller knapsack that cannot fit that object, $K[w - w_o]$. So the goal then is to find that optimum value object. As we are trying to maximise the weight, the optimum value object will clearly be the one for which $v_o + K[w - w_o]$ is the greatest. So therefore, we have the following recurrence relation:

$$\begin{aligned} K[0] &= 0 \\ K[w] &= \max_{w_i \leq w} \big(v_i + K[w - w_i]\big) \end{aligned}$$

The solution to the problem, the maximum value that can be stored in the knapsack weighing no more than W, is the entry $K[W]$.

We define the function *knapsack_unbounded.d.n* W V, which defines an array whose entries are the maximum value that can be obtained by making a knapsack with maximum weight that is the index of the entry. There are n items, and the arrays W and V are the item weights and item values respectively, both varying in dimension d. The definition is fairly straightforward, the only difficulty being that we need to compute the maximum of an array, but only include entries whenever the condition $w_i \leq w$ is met.

For that, we use the left-fold-whenever function defined previously, and folding using the *max* function, to achieve what we want. We define the function *knapsack_unbounded* as follows:

$$
\begin{array}{l}
\texttt{fun}\ \mathit{knapsack\_unbounded.d.n}\ W\ V = K \\
\texttt{where} \\
\quad \texttt{dim}\ a \leftarrow n + 1 \\
\quad \texttt{var}\ K = \mathit{fby.d}\ 0\ \big(\mathit{fold\_wvr.a.max.0}\ \big(Va + (K\ \texttt{@}\ [d \leftarrow \texttt{\#}.d - Wa])\big) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \big(Wa \leq \texttt{\#}.d\big)\big) \\
\quad \texttt{var}\ Va = \mathit{rotate.d.a}\ V \\
\quad \texttt{var}\ Wa = \mathit{rotate.d.a}\ W \\
\texttt{end}
\end{array}
$$

To solve the knapsack problem, we define a variable K, varying in the d direction, whose entries are the maximum value that can be achieved for the weight which is the index of the entry. Since direction d is already being used, we define a new dimension a for the direction in which the *fold_wvr* filter will be carried out. Therefore, we rotate both the weights W and the values V into direction a, and define these as Wa and Va. Then, each entry of K is simply the maximum of the expression passed as argument to *fold_wvr*.

4.12 Sorting

The next few sections will examine the four best known sorting algorithms, and present their implementation in TransLucid, along with a geometric view of the structures created whilst carrying out the respective sorts.

4.12.1 Swapping

A common operation in sorting is to swap two elements. In TransLucid, since every expression defines a new intension, what is required is to define a function whose result is the input, but with the two requested elements swapped. We define a function $\mathit{swap.d.m.n}\ X$, which takes as input the array X, for which we would like to swap the elements m and n in direction d, and returns an array where those two elements are swapped. (Note that *swap* continues to work, even if $m = n$.)

$$
\begin{array}{l}
\texttt{fun}\ \mathit{swap.d.m.n}\ X = \texttt{if}\ \texttt{\#}.d \equiv m\ \texttt{then}\ X\ \texttt{@}\ [d \leftarrow n] \\
\qquad\qquad\qquad\qquad\quad \texttt{elsif}\ \texttt{\#}.d \equiv n\ \texttt{then}\ X\ \texttt{@}\ [d \leftarrow m] \\
\qquad\qquad\qquad\qquad\quad \texttt{else}\ X \\
\qquad\qquad\qquad\qquad\quad \texttt{fi}
\end{array}
$$

For example, the result of the function application $\mathit{swap}.0.0.1\ (10 - \texttt{\#}.0)$ is summarised in the following table:

	0	1	2	$\overset{\texttt{\#}.0}{\rightarrow}$
$10 - \texttt{\#}.0$	10	9	8	$\cdots$
$\mathit{swap}.0.0.1\ (10 - \texttt{\#}.0)$	9	10	8	$\cdots$

4.12.2 Bubble sort

Next we will explore the bubble-sort algorithm. The input to a bubble sort is an array of n objects, and the output is that array of objects in increasing order.

Let us first take a procedural view of bubble sort: it runs through the array n times, each time iterating through the array, swapping every pair of successive objects that is out of order. The bubble sort is named because it has the effect of *bubbling* each element up to its sorted position. After the first pass, the greatest element will be in its sorted position, after the second pass, the second greatest element, and so on. Therefore, after n passes, the entire array will be sorted. A small optimisation is to recognise that after i passes, the last i elements will be sorted, so pass i can be stopped at $n - i$ elements.

Let us now consider an indexical solution to bubble sort. When designing an indexical solution to a problem, the general idea is that every computational step is indexable. In other words, every transformation of data is accessible in some array by as many dimensions as necessary.

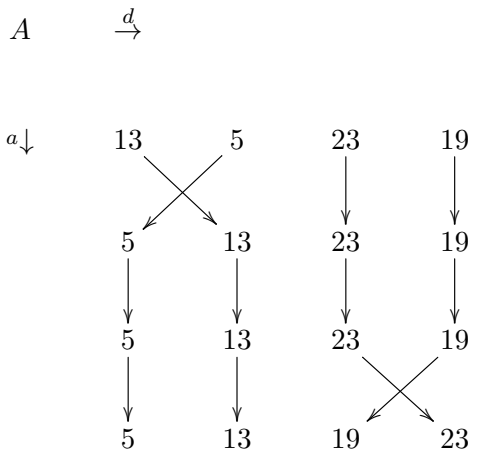
For bubble sort, we will build up the solution piece by piece, looking at each of the transformations carried out on the input array. The first step is to define a function that swaps two elements of an array if they are not in sorted order. Since, in TransLucid, every function defines a new intension based on its inputs rather than producing a side effect, the intension that we need to define has every element the same as the input, except for the two that we are concerned about. Then if the lower element is greater than the higher element, they are swapped.

The next step is to define the array that is the result of doing one pass of swaps. For that we define the function *bubble_one.d.n* X, that will only do one pass of the input array X, for n elements, in direction d.

$$
\begin{array}{l}
\mathit{bubble\_one.d.n}\ X = A \\
\texttt{where} \\
\quad \texttt{dim}\ a \leftarrow n - 1 \\
\quad \texttt{var}\ A = \mathit{fby.a}\ X \\
\qquad\qquad\qquad \texttt{if}\ \mathit{at.d.}(\texttt{\#.}a)\ A > \mathit{at.d.}(\mathit{index}\,!\,a)\ A \\
\qquad\qquad\qquad \texttt{then}\ (\mathit{swap.d.}(\texttt{\#.}a).(\mathit{index}\,!\,a)\ A) \\
\qquad\qquad\qquad \texttt{else}\ A \\
\qquad\qquad\qquad \texttt{fi} \\
\texttt{end}
\end{array}
$$

We define a local dimension a for the direction in which to carry out the swaps. Then the variable A is a two-dimensional array, where the a-th row is the result of swapping elements a and $a + 1$ if necessary. Suppose that the input array has 4 elements, and is the sequence of numbers 13, 5, 23, 19, then the intension A is defined as in the diagram

below, with arrows indicating the direction of the data flow.



The last step is to modify the array A so that it carries out n sequences of swaps. To achieve that, we declare another dimension b, so that each layer in direction b is one set of swaps in direction a. The neatest way is to define the whole top row as the input array X, and the first row of each layer is the last row of the previous layer. The new code is as follows:

```
bubble_sort.d.n X = A
where
  dim a ← n − 1
  dim b ← n − 1
  var A = fby.b X
                (fby.a (at.a.(n − 1) A)
                       (next.b if at.d.(#.a) A > at.d.(index ! a) A
                               then swap.d.(#.a).(index ! a) A
                               else A
                               fi))
end
```

In Figure 4.4, observe that the whole top layer in direction b is the input intension X, and every other layer below that is the second argument to *fby*. Within each layer, the zeroth entry in direction a is the same as the last entry in the previous row. Then the second argument to the second *fby* produces the remainder of each layer.

Using this geometric view of bubble sort, we can analyse its computational complexity. To do that, we look at how much work is done in each dimension of the cube.

The only place where actual work is done is in stepping from one row to the next in the a direction, which corresponds to the second argument to *fby.a*, where there is a conditional to determine whether the appropriate two elements should be swapped or not. One step carries out a constant amount of work, or $\Omega(1)$ work.

From there, it is simply a matter of counting up how many steps are carried out in each direction. As it is clear that there are n steps in the a direction, and n steps in the b

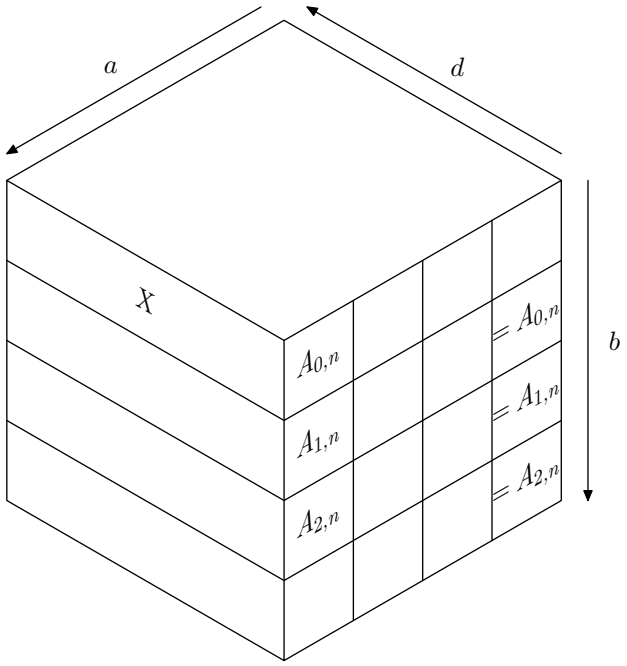


Figure 4.4: Bubble sort array A (version 1)

direction, the complexity of this algorithm is $\Omega(n^2)$, which agrees with the procedural implementation of bubble sort.

There is a small improvement that we can make to bubble sort. We observe that after i passes, the last i elements will be in sorted order. So therefore, it is unnecessary to continue the swaps after we have been through $n - i$ elements. In the TransLucid implementation, this corresponds to taking the result at row $\#.b = i$ at position $\#.a = n - i$. The faster bubble sort is defined below, and Figure 4.5 shows the three-dimensional structure created:

$$
\begin{array}{l}
\mathit{bubble\_sort\_faster}.d.n\ X = A \\
\texttt{where} \\
\quad \texttt{dim}\ a \leftarrow 0 \\
\quad \texttt{dim}\ b \leftarrow n - 1 \\
\quad \texttt{var}\ A = \mathit{fby}.b\ X \\
\qquad\qquad \big(\mathit{fby}.a\ (\mathit{at}.a.(n - \#.b)\ A) \\
\qquad\qquad\quad (\mathit{next}.b\ \texttt{if}\ \mathit{at}.d.(\#.a)\ A > \mathit{at}.d.(\mathit{index}\ !\ a)\ A \\
\qquad\qquad\qquad\quad \texttt{then}\ \big(\mathit{swap}.d.(\#.a).(\mathit{index}\ !\ a)\ A\big) \\
\qquad\qquad\qquad\quad \texttt{else}\ A\ \texttt{fi})\big) \\
\texttt{end}
\end{array}
$$

4.12.3 Insertion sort

In an insertion sort, the array to be sorted is split into two arrays: the still-unsorted part, and the already-sorted part. Initially, the already-sorted part is empty, and then each item from the still-unsorted part is inserted into the already-sorted part. We can use a binary search to look for the position to insert the next item, and then every item past it

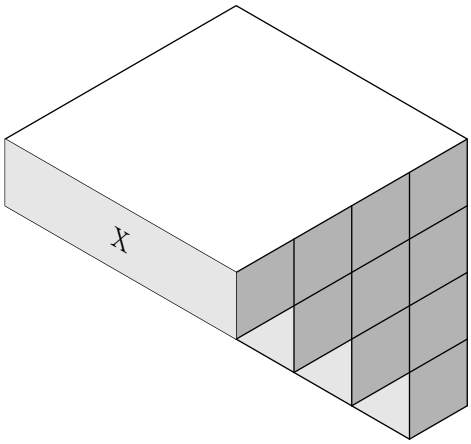


Figure 4.5: Bubble sort array A (version 2)

in the sorted array must be shifted along by one.

First, we define a binary-search function. To determine where to insert the next element v into the already-sorted array X, we want to find the first position in X that does not compare less than v. So we define the function *lower_bound*, which takes as input the direction to search in, the bounds of the area to search, the value to search for, and the array to search in.

The function *lower_bound* is defined as follows:

```
fun lower_bound.d.l.r.v X = asa.a left (left ≡ right)
where
  dim a ← 0
  var left = fby.a l (if (X @ [d ← middle]) < v then middle + 1 else left fi)
  var right = fby.a r (if (X @ [d ← middle]) < v then right else middle fi)
  var middle = (left + right)/2
end
```

Here we use three variables and a dimension a. The variables each vary in dimension a, and represent the search area at each step of the search. Initially, we search the full bounds of the area to search, then, if the value to search for is in the left half, we move the right end of the search area, and if it is in the right half, we move the left end of the search area. We stop the search when the search area is one item, and the position of that one item is the result.

The function *insertion_sort* is defined below, and sorts the array X in direction d from positions 0 to n.

```
fun insertion_sort.d.n X = if 0 ≤ #.d && #.d < n then sorted else X fi
where
  dim a ← n
  var sorted = fby.a infty (insert.d.v.insertPos sorted)
  var v = X @ [d ← #.a]
  var insertPos = lower_bound.d.0.(#.a + 1).v sorted
  fun insert.d.v.p X = if #.d < p then X elsif #.d ≡ p then v else prev.d X fi
end
```

We define a dimension a, which is used for each step of the sort—one step in direction a results in the next element from X being inserted into the sorted array. We initialise the sorted part of the array with the value *infty*—every integer compares less than *infty*—hence all insertions will be before the *infty* values. Then it is simply a matter of finding the position to insert with *lower_bound*, and inserting the next item in the appropriate position. The insert function inserts a value v at position p into array X. Everything before position p is unchanged, the value at position p becomes v, and every subsequent value is just the previous value from X.

4.12.4 Merge sort

Next we examine the TransLucid implementation of merge sort. Merge sort works by merging pairs of already sorted parts of the input array to make new sorted parts, then repeats the process until the array is completely sorted. Initially, none of the array is sorted, so it is considered that each item by itself is a sorted array of size one, and each pair of these one-element arrays is merged together, creating sorted arrays of size two. Then the pairs of arrays of size two are merged to create sorted arrays of size four, and so on. There are two parts to the solution, the first is to define the merge operation, the second is to decide how many times to perform the merge, and on which parts of the array, so that the whole array is sorted.

First, we consider the merge operation in sequential programming. We keep one pointer into each of the two arrays, initially both pointing to the first element of their respective array. The element that is chosen as the first element of the sorted array is the smaller of the two. Then, only the pointer for the array from which the element was chosen is advanced. This process, choosing the smaller and then advancing only its respective pointer, is repeated until the end of both arrays is reached.

Below, we present the definition of the merge function, an explanation will follow.

```
fun merge.d X Y = if X' ≤ Y' then X' else Y' fi
where
  var X' = upon.d X (X' < Y')
  var Y' = upon.d Y (Y' ≤ X')
end
```

Here we have a strange double recursion going on. A simple example will make sense of the situation. Consider the function application *merge*.0 A B, where arguments A and B both vary in dimension 0, and are defined as follows:

	0	1	2	3	$\overset{\#.0}{\rightarrow}$
A	2	5	7	9	...
B	3	6	7	10	...

Then the first eight entries of the variables X' and Y' are:

	0	1	2	3	4	5	6	7	$\overset{\#.0}{\rightarrow}$
X'	2	5	5	7	7	9	9	X_4	...
Y'	3	3	6	6	7	7	10	10	...

First, consider the zeroth entry of both X' and Y'. The zeroth entry of the array defined by the *upon* function is the zeroth entry of its second argument. So the zeroth entries of X' and Y' are simply the zeroth entries of A and B.

The *upon* function defines an array whose values are the result of *advancing upon* the second argument whenever the third argument is true; the third argument is evaluated at the previous index. So to see what the first entry of X' is, we compare the values from the zeroth entries of X' and Y', 2 and 3, and since 2 is less than 3, A is advanced upon, and the first entry is 5. For Y', a similar process is carried out, but with the direction of the comparison swapped. As 3 is not less than 2, B is not advanced upon, so the first entry of Y' is 3, the same as the zeroth entry. The same process is carried out for every subsequent pair of entries from X' and Y'.

The result of the merge function is then simply comparing each corresponding pair of entries from X' and Y', giving the result below:

	0	1	2	3	4	5	6	7	$\overset{\#.0}{\rightarrow}$
merge.0 A B	2	3	5	6	7	7	9	10	...

Once the merge function has been defined, it then remains to merge the appropriate parts of the input array. There are two problems to solve: 1) the merge function merges two arrays from position zero—we cannot specify an arbitrary starting point; and 2) the merge function is an infinite merge, it merges array elements until there are no more elements to merge. To solve this, we need to merge all subarrays from zero, and provide a way to terminate the merging.

We will describe the solution to both of these problems by way of an example. Suppose that the input array X, varying in dimension d is as in the following table, and we want to sort the first ten elements.

	0	1	2	3	4	5	6	7	8	9	$\overset{\#.d}{\rightarrow}$
X	43	48	43	8	25	46	3	2	5	4	...

To solve the above two problems, we use, as for the insertion sort, the value *infty*. By

surrounding the parts of the array that we want to sort with *infty* values, we effectively terminate the sorting at the *infty*. To start the merge of each subarray at zero, we rotate the input array into a new direction a, and surround the rotated array by *infty* in both directions a and d. To surround the array by *infty* in two dimensions, we use the function $default_2$, which takes six arguments defining a rectangle, a default value for the values outside of the rectangle, and an intension whose values will be used inside the rectangle. The rectangle is defined using the dimension of relevance, and the start and end positions, in both directions.

$default_2.d.0.0.a.0.9.infty\ (rotate.d.a\ X)$	0	1	$\overset{\#.d}{\rightarrow}$
$\overset{\#.a}{\downarrow}$	43	*infty*	...
	48	*infty*	...
	43	*infty*	...
	8	*infty*	...
	25	*infty*	...
	46	*infty*	...
	3	*infty*	...
	2	*infty*	...
	5	*infty*	...
	4	*infty*	...
	infty	*infty*	...
	⋮	⋮	⋱

Then, we only need to merge in direction d, every pair of arrays in direction a. This process of merging each pair of arrays is repeated until the array is completely sorted. We carry out each set of merges in a new direction b. If we define a variable Y to hold the result of the merges in the b direction, then the table presented above is Y @ $[b \leftarrow 0]$, and the following two tables are Y @ $[b \leftarrow 1]$ and Y @ $[b \leftarrow 2]$, respectively.

Y @ $[b \leftarrow 1]$	0	1	2	$\overset{\#.0}{\rightarrow}$
$\overset{\#.a}{\downarrow}$	43	48	*infty*	...
	8	43	*infty*	...
	25	46	*infty*	...
	2	3	*infty*	...
	4	5	*infty*	...
	infty	*infty*	*infty*	...
	⋮	⋮	⋮	⋱

Y @ $[b \leftarrow 2]$	0	1	2	3	$\overset{\#.0}{\rightarrow}$
$\#.a\downarrow$	8	43	43	48	...
	2	3	25	46	...
	4	5	*infty*	*infty*	...
	infty	*infty*	*infty*	*infty*	...
	⋮	⋮	⋮	⋮	⋱

What is going on here is that each row m in the a direction is the result of merging rows $2m$ and $2m+1$ from the previous set of entries in the b direction. To specify this in TransLucid, we want to use the function *fby*, in the b direction, where the first entry is the rotation and nesting in the sea of *infty* as above, and every subsequent entry merges as just described. The full merge sort definition is below:

```
fun merge_sort.d.n X = Y
where
  dim a ← 0
  dim b ← ilog.n
  var Y = fby.b (default_2.d.0.0.a.0.(n − 1).infty (rotate.d.a X))
                (merge.d (LofPair.a Y) (RofPair.a Y))
end
```

4.12.5 Quick sort

Quick sort is an interesting case in TransLucid, because the structure of the recursion is based on the data, rather than being fixed, as for merge sort. For merge sort, we can write down statically the intensions that will be built, which is regardless of the data being sorted; we know that we need a two-dimensional intension to sort any sequence of values using merge sort. In contrast, for quick sort, we cannot know how the computation will proceed until given some input data. We therefore set up quick sort to make two recursive calls to itself, each call allocating a new dimension for itself.

For quick sort, most of the work is in partitioning the array; it is trivial to join the sorted portions back together. To partition the array, we choose a pivot element p, and then we want all the elements that are smaller than p to be on its left, and all the elements that are greater than p on its right. To carry out the partition, we make one pass through the array, keeping a pointer to the end of the smaller portion, *smallend*, as computed so far. Then, when an element that is smaller than p is encountered, it is added to the end of the smaller portion, and *smallend* is incremented by one. Then, once we are at the end of the array, the partition is done.

Finally, there is some housekeeping necessary with the pivot value. Before partitioning, the pivot is placed at the end of the array, to keep it out of the way; after partitioning, it is placed in its final position in between the two partitioned parts, which will be at index *smallend*.

After partitioning, we then simply sort the left half and the right half separately, using the ranges $[m, \mathit{smallend})$ and $[\mathit{smallend}+1, n)$, since we do not need to sort the pivot value.

We define the function *quick_sort.d.m.n* X, which takes the dimension in which to sort, and the range $[m, n)$ in which to sort X. Every other value of X is left the same.

```
fun quick_sort.d.m.n X = if #.d < m || #.d ≥ n then X else result fi
where
  dim a ← n − m − 2
  var result = if n − m < 2 then X else glued fi
  var glued = if #.d < smallend
              then quick_sort.d.m.smallend partition
              else quick_sort.d.(smallend + 1).n partition
              fi
  var partition = swap.d.(n − 1).smallend P
  var P = fby.a (swap.d.pivoti.(n − 1) X)
                (if doswap then swap.d.current.smallend P else P fi)
  var current = #.a + m
  var smallend = upon.a (#.a + m) doswap
  var doswap = (P @ [d ← current]) < pivot
  var pivoti = m + (n − m)/2
  var pivot = X @ [d ← pivoti]
end
```

There are several parts of this program that need explaining. For the choice of the pivot, here we have simply chosen the middle value, but this can be chosen in any way desired.

The variable P is the result of partitioning X. For the first entry, we swap the pivot to the end of the array, then every subsequent entry is simply the result of swapping the appropriate element if necessary. The index of the current element to look at is *current* $= \#.a + m$, since we are starting from position m. The variable *doswap* is an array of Booleans that indicates whether the current element needs swapping or not. Then, the end of the smaller part of the array, *smallend*, is only advanced upon whenever we have to swap an element.

Finally, the fully partitioned array is given by *partitioned*, which requires us to swap the pivot back to its final position in the array. Then the actual result of the quick sort is given by the variable *glued*, which just carries out a quick sort on the lower half for elements smaller than the pivot, and in the upper half for elements greater than the pivot. We also do not want to try to sort anything if we have fewer than two elements to sort, since arrays of size zero and one are already sorted. Therefore *result* defines the fully sorted array. Then, finally, the expression defining the function ensures that only elements between m and n are actually sorted.

4.13 Arrays of functions

The final example, which motivated much of the work in Chapters 2 and 3, comes from the end of the book, *Lucid, the Dataflow Programming Language* [43], in which a hypo-

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

$$
\begin{array}{l}
\texttt{fun}\ exp.n = P \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ P = fby.d\ (\lambda^{\mathrm{b}}\ m \to 1)\ \left(\lambda^{\mathrm{b}}\ \{d\}\ m \to m \times P.m\right) \\
\texttt{end}
\end{array}
$$

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

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

4.14 Conclusion

This chapter has presented a selection of programming examples, using TransLucid, along with a guide to understanding these problems geometrically, which is applicable not only to TransLucid, but to understanding these problems in general.

The final example presenting arrays of functions solves the problem of the hypothetical language *Lambda Lucid*. In fact, the problem as it was originally presented could not have worked, as is, in Lucid, and it is only with the binding of dimensions that the problem was solved.

The examples presented in this chapter barely touch the surface of programming in general, and what is possible with multidimensional programming in TransLucid. It is only through further development of programming methodology, and attempts to solve more problems in TransLucid, that progress will be made in understanding multidimensional programming.

Nevertheless, these examples demonstrate that a geometric interpretation of problems is useful, both in understanding the problem, and in specifying the solution in TransLucid. If this geometric view is extended to implementations, then we will be better able to understand the physical movement of data through caches, pipelines and memories, also in a geometric manner.

Chapter 5

Operational TransLucid

This chapter presents *Operational TransLucid*, which is the language used in Chapter 6 for producing an operational semantics with memoization (caching) and in Chapter 7 for type inference. All TransLucid Lite expressions can be translated to Operational TransLucid, using a syntactic transformation given in this chapter. In Operational TransLucid, function parameters and local dimension identifiers become dimensions, to be manipulated in the context rather than in the environment. As a result, implementing TransLucid becomes simpler, since the only manipulation of the environment is in the `wherevar` clause.

5.1 Abstract syntax

The syntax of Operational TransLucid is given in Figure 5.1. Apart from identifiers (x), for every syntactic element in the semantics of TransLucid Lite, there is a corresponding syntactic element in Operational TransLucid: `#`, λ^{b}, λ^{v}, $\uparrow$ and `wheredim` become, respectively, $\texttt{\#}_\circ$, $\lambda^{\mathrm{b}}_\circ$, $\lambda^{\mathrm{v}}_\circ$, $\uparrow_\circ$ and $\texttt{wheredim}_\circ$. Identifiers x become either x (if x is defined in the initial environment or in a `wherevar` clause), or $\underline{\phi_x}$ (if x is defined as a function parameter or a local dimension identifier), where ϕ_x is a hidden dimension. The symbol Φ stands for a set of hidden dimensions of the form $\{\phi_{x_1}, \ldots, \phi_{x_n}\}$.

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

Figure 5.1: Syntax of Operational TransLucid expressions

The idea for a parameter x is that it will be replaced by the entry $\underline{\phi_x}$, which will evaluate to $\kappa(\phi_x)$, i.e., the ϕ_x-ordinate in the current context.

A TransLucid expression E, in which no identifier is masked by a `wherevar` or `wheredim` clause, will be transformed into an equivalent Operational TransLucid expression using transformations $\mathcal{W}$ (Figure 3.2, p.37) and $\mathcal{U}$ (Figure 5.2, p.71).

5.2 Using the context

The key concept of Operational TransLucid is that, other than the `wherevar` clause, uses of the environment are replaced with uses of the context. To achieve this, every identifier x in function abstractions and `wheredim` clauses is replaced with a unique hidden dimension ϕ_x. Then, where the semantics for TransLucid perturbs the environment with a new intension for x, the semantics of Operational TransLucid perturbs the context with a new value for the ordinate of ϕ_x, and every occurrence of x in the syntax is replaced with a context lookup for dimension ϕ_x.

This however, is not quite enough. We observe that in the original semantics, the body of an abstraction is evaluated in the environment in which it was *created*, not the one in which it is evaluated. Therefore, the setting of any function parameters and local dimension identifiers enclosing the relevant abstraction must be retained as part of the closure for that abstraction. Since we have just moved those parameters to the context, it is necessary to retain a set of dimensions containing the ϕ dimensions of the enclosing abstractions and `wheredim` clauses.

5.3 Transformation

Definition 16. *Let E be a TransLucid Lite expression in which no identifier is masked by a* `wheredim` *or* `wherevar` *clause, $\Phi = \{\phi_{x_i}\}$ be a set of hidden dimensions, where each ϕ_{x_i} corresponds to an identifier x_i defined in a λ^{b}-abstraction, a λ^{v}-abstraction or a* `wheredim` *clause for which E is a subexpression, and $X = \{x_j\}$ be a set of identifiers, where expression x_j is to be replaced by expression $\underline{\phi_{x_j}}$. Then expression $\mathcal{U}(E, \Phi, X)$ is the transformation of E into Operational TransLucid, where $\mathcal{U}$ is given by Figure 5.2. When E is a standalone expression, $\Phi = \emptyset$ and $X = \emptyset$, since there are no surrounding λ^{b}-abstractions, λ^{v}-abstractions or* `wheredim` *clauses.*

Definition 17. *Let E be a standalone TransLucid expression in which no identifier is masked by a* `wheredim` *or* `wherevar` *clause. Expression $\mathcal{U}(\mathcal{W}(E), \emptyset, \emptyset)$ transforms E into an Operational TransLucid expression.*

The transformation $\mathcal{U}$ takes three parameters: the expression to transform, a set of ϕ dimensions for the abstractions above the current expression, and a set of identifiers for the abstractions above the current expression. For the four abstractions, the transformation replaces them with their corresponding circle abstraction which retains the set Φ for the hidden dimensions to retain when the abstraction is created. Additionally, for the

$$
\begin{aligned}
\mathcal{U}(x, \Phi, X) &= \begin{cases} \underline{\phi_x}, & x \in X \\ x, & \text{otherwise} \end{cases} \\
\mathcal{U}({}^m c, \Phi, X) &= {}^m c \\
\mathcal{U}(\mathtt{\#}, \Phi, X) &= \mathtt{\#}_\circ \\
\mathcal{U}([E_{i0} \leftarrow E_{i1}]_{i=1..m}, \Phi, X) &= \left[E_{i0}^{\mathcal{U}} \leftarrow E_{i1}^{\mathcal{U}}\right] \\
\mathcal{U}(\lambda x \to E_0, \Phi, X) &= \lambda^{\mathrm{b}}_\circ\, \emptyset\ \Phi\ \phi_x \to \mathcal{U}\big(E_0, \Phi \cup \{\phi_x, \rho\}, X \cup \{x\}\big) \\
\mathcal{U}(E_0\,.\,(E_i)_{i=1..m}, \Phi, X) &= E_0^{\mathcal{U}}\,.\,E_i^{\mathcal{U}} \\
\mathcal{U}(\mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2\ \mathtt{fi}, \Phi, X) &= \mathtt{if}\ E_0^{\mathcal{U}}\ \mathtt{then}\ E_1^{\mathcal{U}}\ \mathtt{else}\ E_2^{\mathcal{U}}\ \mathtt{fi} \\
\mathcal{U}(E_0\ \mathtt{@}\ E_1, \Phi, X) &= E_0^{\mathcal{U}}\ \mathtt{@}\ E_1^{\mathcal{U}} \\
\mathcal{U}(\uparrow\{E_i\}_{i=1..m}\ E_0, \Phi, X) &= \uparrow_\circ \big\{E_i^{\mathcal{U}}\big\}\ \Phi\ E_0^{\mathcal{U}} \\
\mathcal{U}(\downarrow E_0, \Phi, X) &= \downarrow E_0^{\mathcal{U}} \\
\mathcal{U}(\lambda^{\mathrm{b}}\ \{E_i\}_{i=1..m}\ x \to E_0, \Phi, X) &= \lambda^{\mathrm{b}}_\circ\ \big\{E_i^{\mathcal{U}}\big\}\ \Phi\ \phi_x \to \mathcal{U}\big(E_0, \Phi \cup \{\phi_x, \rho\}, X \cup \{x\}\big) \\
\mathcal{U}(\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \to E_0, \Phi, X) &= \lambda^{\mathrm{v}}_\circ\ \big\{E_i^{\mathcal{U}}\big\}\ \Phi\ \phi_x \to \mathcal{U}\big(E_0, \Phi \cup \{\phi_x, \rho\}, X \cup \{x\}\big) \\
\mathcal{U}(E_0\,!\,E_1, \Phi, X) &= E_0^{\mathcal{U}}\,!\,E_1^{\mathcal{U}} \\
\mathcal{U}(E_0\ \mathtt{wherevar}\ x_i = E_i\ \mathtt{end}_{i=1..m}, \Phi, X) &= E_0^{\mathcal{U}}\ \mathtt{wherevar}\ x_i = E_i^{\mathcal{U}}\ \mathtt{end} \\
\mathcal{U}(E_0\ \mathtt{wheredim}\ x_i \leftarrow E_i\ \mathtt{end}_{i=1..m}, \Phi, X) &= \mathcal{U}\big(E_0, \Phi \cup \{\phi_{x_i}\}, X \cup \{x_i\}\big)\ \mathtt{wheredim}_\circ\ \phi_{x_i} \leftarrow E_i^{\mathcal{U}}\ \mathtt{end}
\end{aligned}
$$

Figure 5.2: Transformation $\mathcal{U}$ from TransLucid Lite to Operational TransLucid $\big(E^{\mathcal{U}} = \mathcal{U}(E, \Phi, X)\big)$.

abstractions that take a parameter, and the `wheredim` clause, their bodies are transformed with their hidden dimension added to Φ, and their parameter added to X.

The set X in the transformation is the set of identifiers which have been mapped to hidden dimensions. So when an identifier x is the parameter of an abstraction, or a local dimension identifier, and appears in the body of that abstraction or `wheredim` clause, it is replaced with a dimension lookup for the appropriate ϕ dimension. Otherwise, the transformation of an identifier is itself.

5.4 Semantic rules

Definition 18. *Let E be an Operational TransLucid expression over Σ and X, ι be an interpretation, ζ be an environment, κ be a context, and $\Phi_X \subset \Delta_O$ be a set of ϕ dimensions. Then the semantics of E is given by $[\![E]\!]\iota\zeta(\kappa \dagger \{\rho \mapsto \epsilon\})$, where the rules for $[\![\cdot]\!]$ are given in Figure 5.3.*

The operational semantic rules make use of a set Φ_X of ϕ dimensions. These dimensions are added to the semantic domain D by specifying that they are a part of the set Δ_O of other dimensions (cf. Preamble 1, p.23), which allows any number of new dimensions to be added to the system for the purposes of implementation. When the syntactic constructs for Operational TransLucid and TransLucid Lite differ, we explain the corresponding rules in the itemised paragraph below.

$$[\![\underline{\phi_x}]\!]\iota\zeta\kappa = \kappa(\phi_x) \tag{5.1}$$

$$[\![x]\!]\iota\zeta\kappa = \zeta(x)(\kappa) \tag{5.2}$$

$$[\![{}^m c]\!]\iota\zeta\kappa = \iota({}^m c) \tag{5.3}$$

$$[\![\#_\circ]\!]\iota\zeta\kappa = \kappa \lhd \Phi_X \tag{5.4}$$

$$[\![[E_{i0} \leftarrow E_{i1}]_{i=1..m}]\!]\iota\zeta\kappa = \{[\![E_{i0}]\!]\iota\zeta({}_i\kappa) \mapsto [\![E_{i1}]\!]\iota\zeta({}_{(i+m)}\kappa)\} \tag{5.5}$$

$$[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]\iota\zeta\kappa = \text{let } d_0 = [\![E_0]\!]\iota\zeta({}_0\kappa) \text{ in } \begin{cases} [\![E_1]\!]\iota\zeta({}_1\kappa), & d_0 \equiv \mathit{true} \\ [\![E_2]\!]\iota\zeta({}_2\kappa), & d_0 \equiv \mathit{false} \end{cases} \tag{5.6}$$

$$[\![E_0 \texttt{ @ } E_1]\!]\iota\zeta\kappa = [\![E_0]\!]\iota\zeta({}_0\kappa \dagger [\![E_1]\!]\iota\zeta({}_1\kappa)) \tag{5.7}$$

$$[\![\lambda^{\mathrm{b}}_\circ \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0]\!]\iota\zeta\kappa = \text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \text{ in } \lambda d_a.[\![E_0]\!]\iota\zeta\ (({}_0\kappa \lhd (\{\rho, d_i\} \cup \Phi)) \dagger \{\phi_x \mapsto d_a\}) \tag{5.8}$$

$$[\![E_0.(E_i)_{i=1..m}]\!]\iota\zeta\kappa = ([\![E_0]\!]\iota\zeta({}_0\kappa))([\![E_i]\!]\iota\zeta({}_i\kappa)) \tag{5.9}$$

$$[\![\uparrow_\circ \{E_i\}_{i=1..m}\ \Phi\ E_0]\!]\iota\zeta\kappa = \text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \text{ in } \lambda\kappa_a.[\![E_0]\!]\iota\zeta(\kappa_a \dagger (\kappa \lhd \{d_i\} \cup \Phi)) \tag{5.10}$$

$$[\![\downarrow E_0]\!]\iota\zeta\kappa = ([\![E_0]\!]\iota\zeta_0\kappa)_1\kappa \tag{5.11}$$

$$[\![\lambda^{\mathrm{v}}_\circ \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0]\!]\iota\zeta\kappa = \text{let } d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \text{ in } \lambda d_a.\lambda\kappa_a.[\![E_0]\!]\iota\zeta\ (\kappa_a \dagger (\kappa \lhd (\{d_i\} \cup \Phi)) \dagger \{\phi_x \mapsto d_a\}) \tag{5.12}$$

$$[\![E_0 \,!\, E_1]\!]\iota\zeta\kappa = ([\![E_0]\!]\iota\zeta_0\kappa)([\![E_1]\!]\iota\zeta_1\kappa)_2\kappa \tag{5.13}$$

$$[\![E_0 \texttt{ wheredim}_\circ\ \phi_{x_i} \leftarrow E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa = \text{let } \delta_i = \chi^i_{\kappa(\rho)},\ d_i = [\![E_i]\!]\iota\zeta({}_i\kappa) \text{ in } [\![E_0]\!]\iota\zeta(({}_0\kappa) \dagger \{\phi_{x_i} \mapsto \delta_i,\ \delta_i \mapsto d_i\}) \tag{5.14}$$

$$[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa = \text{let } \zeta_0 = \zeta \dagger \{x_i \mapsto \emptyset\},\ \zeta_{\alpha+1} = \zeta_\alpha \dagger \{x_i \mapsto [\![E_i]\!]\iota\zeta_\alpha\},\ \zeta_\sqcup = \text{lfp } \zeta_\alpha \text{ in } [\![E_0]\!]\iota(\zeta_\sqcup)\kappa \tag{5.15}$$

Figure 5.3: Semantics of Operational TransLucid

(5.1) The ordinate of dimension ϕ_x is looked up in the current context κ.

(5.8) An abstraction is created that evaluates its body in a context in which the frozen Φ dimensions are retained, and the context at the point of application is ignored.

(5.10) An abstraction is created that evaluates its body in the context passed in at the point of application, retaining the frozen Φ dimensions from the point of creation of the abstraction.

(5.12) An abstraction is created that evaluates its body in the context passed in at the point of application, putting its parameter in the context, along with the retained Φ dimensions.

(5.14) A new dimension δ_i is allocated for each local dimension, and the body of the `wheredim`$_\circ$ clause is evaluated, perturbing the context such that the hidden dimension ϕ_{x_i} maps to the allocated dimension δ_i, and δ_i maps to the initial value, d_i, for each dimension.

5.5 Validity of semantics

Proposition 5. *Let Φ be a set of ϕ dimensions, X be a set of identifiers, E be a TransLucid Lite expression, ι be an interpretation, ζ, ζ' be environments, κ, κ' be contexts such that*

$$\kappa = \kappa' \lhd \mathrm{dom}(\kappa) \tag{5.16}$$

$$\zeta' = \zeta \lhd \mathrm{dom}(\zeta') \tag{5.17}$$

$$\mathrm{dom}(\kappa') - \mathrm{dom}(\kappa) = \left\{\phi_x \mid x \in \mathrm{dom}(\zeta) - \mathrm{dom}(\zeta')\right\} \tag{5.18}$$

$$\mathrm{dom}(\zeta) - \mathrm{dom}(\zeta') = \left\{x \mid \phi_x \in \mathrm{dom}(\kappa') - \mathrm{dom}(\kappa)\right\} \tag{5.19}$$

$$\forall \phi_x \in \mathrm{dom}(\kappa'), \kappa'(\phi_x) = \zeta(x)(\kappa). \tag{5.20}$$

Then

$$[\![\mathcal{U}(E, \Phi, X)]\!]\iota\zeta'\kappa' = [\![E]\!]\iota\zeta\kappa.$$

Proof. By induction over the structure of E. There are four base cases, since there are two cases for x. In the first, $x \in X$:

$$\begin{aligned}
&[\![\mathcal{U}(x, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![\underline{\phi_x}]\!]\iota\zeta'\kappa' \\
&= \kappa'(\phi_x) \\
&= \zeta(x)(\kappa) \\
&= [\![x]\!]\iota\zeta\kappa
\end{aligned}$$

In the second, $x \notin X$:

$$\begin{aligned}
&[\![\mathcal{U}(x, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![x]\!]\iota\zeta'\kappa' \\
&= \zeta'(x)(\kappa') \\
&= \zeta'(x)(\kappa) \\
&= \zeta(x)(\kappa) \\
&= [\![x]\!]\iota\zeta\kappa
\end{aligned}$$

$$
\begin{aligned}
&\llbracket \mathcal{U}({}^m c, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket {}^m c \rrbracket \iota\zeta'\kappa' \\
&= \iota({}^m c) \\
&= \llbracket {}^m c \rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{U}(\mathtt{\#}, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \mathtt{\#}_\circ \rrbracket \iota\zeta'\kappa' \\
&= \kappa' \lhd \Phi_X \\
&= \kappa \\
&= \llbracket \mathtt{\#} \rrbracket \iota\zeta\kappa
\end{aligned}
$$

The transformation $\mathcal{U}$ (Definition 16) and semantics (Definition 18) maintain the invariant defined by properties (5.16)–(5.20). If this invariant holds, then the induction hypothesis is that $\llbracket \mathcal{U}(E, \Phi, X) \rrbracket \iota\zeta'\kappa' = \llbracket E \rrbracket \iota\zeta\kappa$. The first seven inductive cases, '$[\cdots]$', '`if`–`then`–`else`', '`@`', '.', '$\downarrow$', '!', and '`wherevar`', are straightforward.

$$
\begin{aligned}
&\llbracket \mathcal{U}([E_{i0} \leftarrow E_{i1}]_{i=1..m}, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket [E_{i0} \leftarrow E_{i1}] \rrbracket \iota\zeta'\kappa' \\
&= \left\{ \llbracket E^{\mathcal{U}}_{i0} \rrbracket \iota\zeta'({}_i\kappa') \mapsto \llbracket E^{\mathcal{U}}_{i1} \rrbracket \iota\zeta'({}_{(i+m)}\kappa') \right\} \\
&= \left\{ \llbracket E_{i0} \rrbracket \iota\zeta({}_i\kappa) \mapsto \llbracket E_{i1} \rrbracket \iota\zeta({}_{(i+m)}\kappa) \right\} \\
&= \llbracket [E_{i0} \leftarrow E_{i1}] \rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{U}(\mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2\ \mathtt{fi}, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \mathtt{if}\ E^{\mathcal{U}}_0\ \mathtt{then}\ E^{\mathcal{U}}_1\ \mathtt{else}\ E^{\mathcal{U}}_2\ \mathtt{fi} \rrbracket \iota\zeta'\kappa' \\
&= \text{let } d_0 = \llbracket E^{\mathcal{U}}_0 \rrbracket \iota\zeta'({}_0\kappa') \\
&\quad \text{in } \begin{cases} \llbracket E^{\mathcal{U}}_1 \rrbracket \iota\zeta'({}_1\kappa'), & d_0 \equiv \mathit{true} \\ \llbracket E^{\mathcal{U}}_2 \rrbracket \iota\zeta'({}_2\kappa'), & d_0 \equiv \mathit{false} \end{cases} \\
&= \text{let } d_0 = \llbracket E_0 \rrbracket \iota\zeta({}_0\kappa) \\
&\quad \text{in } \begin{cases} \llbracket E_1 \rrbracket \iota\zeta({}_1\kappa), & d_0 \equiv \mathit{true} \\ \llbracket E_2 \rrbracket \iota\zeta({}_2\kappa), & d_0 \equiv \mathit{false} \end{cases} \\
&= \llbracket \mathtt{if}\ E_0\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2\ \mathtt{fi} \rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{U}(E_0 \mathbin{@} E_1, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![E_0^{\mathcal{U}} \mathbin{@} E_1^{\mathcal{U}}]\!]\iota\zeta'\kappa' \\
&= [\![E_0^{\mathcal{U}}]\!]\iota\zeta'\big({}_0\kappa' \dagger [\![E_1^{\mathcal{U}}]\!]\iota\zeta'({}_1\kappa')\big) \\
&= [\![E_0]\!]\iota\zeta\big({}_0\kappa \dagger [\![E_1]\!]\iota\zeta({}_1\kappa)\big) \\
&= [\![E_0 \mathbin{@} E_1]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{U}(E_0 \,.\, (E_i)_{i=1..m}, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![E_0^{\mathcal{U}} \,.\, (E_i^{\mathcal{U}})]\!]\iota\zeta'\kappa' \\
&= \big([\![E_0^{\mathcal{U}}]\!]\iota\zeta'({}_0\kappa')\big)\big([\![E_i^{\mathcal{U}}]\!]\iota\zeta'({}_1\kappa')\big) \\
&= \big([\![E_0]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_i]\!]\iota\zeta({}_1\kappa)\big) \\
&= [\![E_0 \,.\, (E_i)]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{U}(\downarrow E_0, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![\downarrow E_0^{\mathcal{U}}]\!]\iota\zeta'\kappa' \\
&= \big([\![E_0^{\mathcal{U}}]\!]\iota\zeta'({}_0\kappa')\big)({}_1\kappa') \\
&= \big([\![E_0^{\mathcal{U}}]\!]\iota\zeta({}_0\kappa)\big)({}_1\kappa) \\
&= [\![\downarrow E_0]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathcal{U}(E_0 \mathbin{!} E_1, \Phi, X)]\!]\iota\zeta'\kappa' \\
&= [\![E_0^{\mathcal{U}} \mathbin{!} E_1^{\mathcal{U}}]\!]\iota\zeta'\kappa' \\
&= \big([\![E_0^{\mathcal{U}}]\!]\iota\zeta'({}_0\kappa')\big)\big([\![E_i^{\mathcal{U}}]\!]\iota\zeta'({}_1\kappa')\big)({}_2\kappa') \\
&= \big([\![E_0^{\mathcal{U}}]\!]\iota\zeta({}_0\kappa)\big)\big([\![E_i^{\mathcal{U}}]\!]\iota\zeta({}_1\kappa)\big)({}_2\kappa) \\
&= [\![E_0 \mathbin{!} E_1]\!]\iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{U}(E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket E_0^{\mathcal{U}} \texttt{ wherevar } x_i = E_i^{\mathcal{U}} \texttt{ end} \rrbracket \iota\zeta'\kappa' \\
&= \text{let } \zeta'_0 \quad = \zeta' \dagger \{x_i \mapsto \emptyset\} \\
&\qquad \zeta'_{\alpha+1} = \zeta'_\alpha \dagger \{x_i \mapsto \llbracket E_i^{\mathcal{U}} \rrbracket \iota\zeta'_\alpha\} \\
&\qquad \zeta'_\sqcup \quad = \text{lfp } \zeta'_\alpha \\
&\quad \text{in } \llbracket E_0^{\mathcal{U}} \rrbracket \iota(\zeta'_\sqcup)\kappa' \\
&= \text{let } \zeta_0 \quad = \zeta \dagger \{x_i \mapsto \emptyset\} \\
&\qquad \zeta_{\alpha+1} = \zeta_\alpha \dagger \{x_i \mapsto \llbracket E_i \rrbracket \iota\zeta_\alpha\} \\
&\qquad \zeta_\sqcup \quad = \text{lfp } \zeta_\alpha \\
&\quad \text{in } \llbracket E_0 \rrbracket \iota(\zeta_\sqcup)\kappa \\
&= \llbracket E_0 \texttt{ wherevar } x_i = E_i \texttt{ end} \rrbracket \iota\zeta\kappa
\end{aligned}
$$

The remaining four cases are the interesting ones, as the semantics for Operational TransLucid must ensure that the invariant holds.

$$
\begin{aligned}
&\llbracket \mathcal{U}(\uparrow \{E_i\}_{i=1..m}\ E_0, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \uparrow_\circ \{E_i^{\mathcal{U}}\}\ \Phi\ E_0^{\mathcal{U}} \rrbracket \iota\zeta'\kappa' \\
&= \text{let } d_i = \llbracket E_i^{\mathcal{U}} \rrbracket \iota\zeta'({}_i\kappa') \\
&\quad \text{in } \lambda\kappa_a.\llbracket E_0^{\mathcal{U}} \rrbracket \iota\zeta'\big(\kappa_a \dagger (\kappa' \lhd \{d_i\} \cup \Phi)\big) \\
&= \text{let } d_i = \llbracket E_i \rrbracket \iota\zeta({}_i\kappa) \\
&\quad \text{in } \lambda\kappa_a.\llbracket E_0 \rrbracket \iota\zeta\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
&= \llbracket \uparrow \{E_i\}\ E_0 \rrbracket \iota\zeta\kappa
\end{aligned}
$$

The invariant states that for $\phi_x \in \Phi$, $\phi_x \in \text{dom}(\kappa')$, $x \notin \text{dom}(\zeta')$, and $\zeta(x)(\kappa) = \kappa'(\phi_x)$. Since we can make no such assumption about κ_a, it is necessary to add those dimensions from κ' into κ_a before evaluating $E_0^{\mathcal{U}}$. Therefore, the invariant is maintained from step 2 to 3, and the induction hypothesis holds. Similar manipulations are required for the other three cases.

$$
\begin{aligned}
&\llbracket \mathcal{U}(\lambda^{\text{b}}\ \{E_i\}_{i=1..m}\ x \to E_0, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \lambda^{\text{b}}_\circ\ \{E_i^{\mathcal{U}}\}\ \Phi\ \phi_x \to \mathcal{U}(E_0, \Phi \cup \{\phi_x\}, X \cup \{x\}) \rrbracket \iota\zeta'\kappa' \\
&= \text{let } d_i = \llbracket E_i^{\mathcal{U}} \rrbracket \iota\zeta'({}_i\kappa') \\
&\quad \text{in } \lambda d_a.\llbracket E_0^{\mathcal{U}} \rrbracket \iota\zeta'\big(({}_0\kappa' \lhd (\{\rho, d_i\} \cup \Phi)) \dagger \{\phi_x \mapsto d_a\}\big) \\
&= \text{let } d_i = \llbracket E_i \rrbracket \iota\zeta({}_i\kappa) \\
&\quad \text{in } \lambda d_a.\llbracket E_0 \rrbracket \iota\big(\zeta \dagger \{x \mapsto \widehat{d_a}\}\big)({}_0\kappa \lhd \{\rho, d_i\}) \\
&= \llbracket \lambda^{\text{b}}\ \{E_i\}\ x \to E_0 \rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{U}(\lambda^{\mathrm{v}}\ \{E_i\}_{i=1..m}\ x \to E_0, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \lambda^{\mathrm{v}}_{\circ}\ \{E_i^{\mathcal{U}}\}\ \Phi\ \phi_x \to \mathcal{U}(E_0, \Phi \cup \{\phi_x\}, X \cup \{x\}) \rrbracket \iota\zeta'\kappa' \\
&= \text{let } d_i = \llbracket E_i^{\mathcal{U}} \rrbracket \iota\zeta'({}_i\kappa') \\
&\quad \text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0^{\mathcal{U}} \rrbracket \iota\zeta'\big(\kappa_a \dagger (\kappa' \lhd (\{d_i\} \cup \Phi)) \dagger \{\phi_x \mapsto d_a\}\big) \\
&= \text{let } d_i = \llbracket E_i \rrbracket \iota\zeta({}_i\kappa) \\
&\quad \text{in } \lambda d_a.\lambda\kappa_a.\llbracket E_0 \rrbracket \iota(\zeta \dagger \{x \mapsto \widehat{d_i}\}\big(\kappa_a \dagger (\kappa \lhd \{d_i\})\big) \\
&= \llbracket \lambda^{\mathrm{v}}\ \{E_i\}\ x \to E_0 \rrbracket \iota\zeta\kappa
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \mathcal{U}(E_0\ \texttt{wheredim}_{\circ}\ \phi_{x_i} \leftarrow E_i\ \texttt{end}_{i=1..m}, \Phi, X) \rrbracket \iota\zeta'\kappa' \\
&= \llbracket \mathcal{U}(E_0, \Phi \cup \{\phi_{x_i}\}, X \cup \{x_i\})\ \texttt{wheredim}_{\circ}\ \phi_{x_i} \leftarrow E_i^{\mathcal{U}}\ \texttt{end} \rrbracket \iota\zeta'\kappa' \\
&= \text{let } \delta_i = \chi^i_{\kappa(\rho)} \\
&\qquad d_i = \llbracket E_i^{\mathcal{U}} \rrbracket \iota\zeta'({}_i\kappa') \\
&\quad \text{in } \llbracket \mathcal{U}(E_0, \Phi \cup \{\phi_{x_i}\}, X \cup \{x_i\}) \rrbracket \iota\zeta'\big(({}_0\kappa) \dagger \{\phi_{x_i} \mapsto \delta_i, \delta_i \mapsto d_i\}\big) \\
&= \text{let } \delta_i = \chi^i_{\kappa(\rho)} \\
&\qquad d_i = \llbracket E_i \rrbracket \iota\zeta({}_i\kappa) \\
&\quad \text{in } \llbracket E_0 \rrbracket \iota\big(\zeta \dagger \{x_i \mapsto \widehat{\delta_i}\}\big)\big({}_0\kappa \dagger \{\delta_i \mapsto d_i\}\big) \\
&= \llbracket E_0\ \texttt{wheredim}\ x_i \leftarrow E_i\ \texttt{end} \rrbracket \iota\zeta\kappa
\end{aligned}
$$

Hence, by the inductive hypothesis, $\llbracket \mathcal{U}(E, \Phi, X) \rrbracket \iota\zeta\kappa = \llbracket E \rrbracket \iota\zeta'\kappa'$.

□

Chapter 6

Cached Evaluation

This chapter presents an operational semantics for TransLucid using a form of demand-driven computation called *eduction*. This semantics uses a cache storing the results of previously computed (*variable*, *context*) pairs. The key technical contribution is to provide a mechanism for this cache to work with arbitrary-dimensional spaces. We argue how this operational semantics is compatible with the denotational semantics of Chapter 5. This chapter is an extension of the technical part of a paper to be published in *Mathematical Structures in Computer Science* [11], and is inspired by the author's current implementation, and subsumes previous work by Rahilly and Plaice [37].

6.1 Eduction

In the previous chapters, the discussion focused on denotational semantics—the meaning of an expression—rather than on how to actually compute the value of an expression. The problem for evaluating TransLucid is that in general, once the values of its free variables are defined, an expression's value is an intension, and that intension has infinite extent. The only possible solution is, rather than computing the value of an expression, to compute the value of an expression in a particular context.

In fact, the semantics almost does this already. The semantic function $[\![\cdot]\!]$ gives the meaning of an expression in a particular context, and in fact all but two of the rules manipulate the context, and evaluate their subexpressions in the appropriate context. So, in fact, most of the denotational semantics rules already lead naturally to an implementation.

For example, for expression $\texttt{\#}.E$, when E evaluates to dimension δ, the δ-ordinate is simply looked up in the current context. Similarly for $E_0 \texttt{@} E_1$, expression E_1 is evaluated in the current context, producing a tuple, and E_0 is evaluated in the context produced by perturbing the current context with that tuple.

The one problematic rule is for E_0 `wherevar` $x_i = E_i$ `end`, which creates a sequence of environments produced by mapping each identifier x_i to the intension produced by the result of evaluating the corresponding expression E_i in the previous environment; then E_0 is evaluated in the environment which is the least fixed point of the sequence of environments. The problem here is that each identifier is mapping to an infinite intension, which cannot be computed.

The solution to evaluating a `wherevar` clause is called *eduction*; the use of the term first appeared in [43], and was derived from the word *educe*, which means to draw out. In evaluating using eduction, the value of an intension is computed in a particular context, which may result in other intensions, and possibly itself again, being evaluated in different contexts. Conceptually, a demand is made for a pair (x_0, κ_0) of an identifier and a context, which results in demands for further pairs (x_i, κ_i). When all of these demands have been satisfied, the final result can be computed.

As an example, we reproduce below the factorial function presented in Chapter 2.

$$
\begin{array}{l}
\texttt{fun}\ \mathit{fact}.n = F \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ F = \texttt{if}\ \#.d \equiv 0\ \texttt{then}\ 1\ \texttt{else}\ \#.d \times \big(F\ \texttt{@}\ [d \leftarrow \#.d - 1]\big)\ \texttt{fi} \\
\texttt{end}
\end{array}
$$

To compute the value of expression $\mathit{fact}.a$, for some positive integer $a \in \mathbb{N} - \{0\}$, the value of the variable F at the context $\{d \mapsto a\}$ must be computed. Since $\#.d \neq 0$, the `else` branch is evaluated, which results in a request for F at $\{d \mapsto a - 1\}$. This is repeated a times, until the `then` branch returns 1, and then every other value can be computed.

There are only two changes to the semantics required to implement eduction: 1) rather than passing around an environment ζ, it is necessary to pass a mapping ξ from identifiers to expressions; 2) the rule for the `wherevar` clause is modified to perturb ξ with a mapping from x_i to E_i, rather than to compute a least fixed point. In addition, the rule for x must be changed to evaluate the expression $\xi(x)$ in the current context.

Unfortunately, however, evaluation using naive eduction is typically an inefficient way to evaluate an expression, due to repeated requests for the same (x, κ) pair. Take as example the Fibonacci program below:

$$
\begin{array}{l}
\mathit{fib}.n = F \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ F = \texttt{if}\ \#.d \leq 1\ \texttt{then}\ \#.d\ \texttt{else}\ \big(F\ \texttt{@}\ [\#.d \leftarrow \#.d - 1]\big) + \big(F\ \texttt{@}\ [\#.d \leftarrow \#.d - 2]\big)\ \texttt{fi} \\
\texttt{end}
\end{array}
$$

Even demands for low numbers produce an enormous number of repeated requests. A demand for F in the context $\{d \mapsto 5\}$ requests F at $\{d \mapsto 4\}$ and $\{d \mapsto 3\}$. Those two demands produce another demand for F at $\{d \mapsto 3\}$, and additionally two demands at $\{d \mapsto 2\}$ and one at $\{d \mapsto 1\}$. The second demand at 3 then produces demands at 2 and 1 again. This only gets worse as the Fibonacci number requested grows.

The solution is to remember the results of particular entries the first time they are computed, so that they are not recomputed when later requested. Then they can simply be looked up for future requests. This technique is called *memoisation*, and is commonly used for caching intermediate results in calls of recursively-defined pure functions. The key technical contribution of this chapter is to extend this idea to memoisation in an arbitrary-dimensional space, what we call *multidimensional memoisation*.

6.2 Multidimensional memoisation

The difficulty in implementing a multidimensional cache scheme is that the dimensions of relevance in looking up a cache entry cannot be known *a priori*, and therefore must be discovered by the interpreter during evaluation. This chapter presents a solution that involves a completely new method of evaluation, in which there is a back-and-forth interaction between the cache and the evaluator, with the evaluator returning demands for dimensions as they are required, whilst the cache builds a hierarchy of those dimensions as they are requested.

The basic idea of caching results of computation is that for each entry in an intension that has been computed, we store the dimensions that were used to compute it, and the value computed. As an example, consider a simple multiplication table:

$$\texttt{var}\ M = \#.a \times \#.b$$

It is easy to see how a cache might work in this situation. When a request for the value of M at the context $\kappa = \{a \mapsto 5, b \mapsto 6\}$ is made, the cache is looked up at κ, and if an entry exists, it is returned, otherwise, the value 30 is computed and, the following fact is entered into the cache:

$$\text{value of } M \text{ at } \{a \mapsto 5, b \mapsto 6\} \text{ is } 30$$

This works fine for simple expressions, although, even this case raises some questions. The question that must be solved in order to implement a cache is, which dimensions are being used to define an intension? In this case, we can see that it is dimensions a and b, but especially in more complex expressions, it is not so trivial to arrive at that answer, and it is not so simple for the evaluator to determine that itself.

There are three possibilities: 1) static analysis is used, in order to determine which dimensions are used by any expression, 2) the cache simply uses the dimensions that are present in a computation, or 3) the computer uses some more complex method of determining which dimensions are relevant to a computation. The solution is (3), because we can eliminate the other two quite easily. Option (1) is not possible, because dimensions can be passed as parameters, and can be computed, and so the dimensions used to compute the value of an expression are not in general known statically. Option (2) does not work, because as will be explained shortly, there may be many other dimensions present that are not relevant for a particular cache request. Therefore, we present a solution for implementing the cache that involves the evaluator discovering, at run time, the dimensions relevant for a computation, and entering the appropriate information into a cache.

Finally, without some sort of garbage collection, the cache would quickly grow to an unmanageable size, so we also implement what was called the "retirement-plan" garbage-collection system by Faustini and Wadge [18], which is essentially a system for retaining the most recently used entries, and throwing out entries that have not been used for a while. In their implementation, almost nothing used later was thrown away, and the cache was kept to a reasonable size. The retirement-plan scheme is not unlike that used by an

operating system in managing a CPU's cache of memory, or the cache of a disk in RAM.

6.3 Accessing the cache

The difficulty with the cache becomes apparent with a simple example. When there are more dimensions in the context than necessary to define a particular expression, it is not immediately clear to the evaluator that not all of those dimensions should be used in defining the cache entry. The variable N, defined below, only uses the s-ordinate when it is greater than zero, otherwise, it uses both the s-ordinate and the t-ordinate:

$$N = \texttt{if \#.}s > 0 \texttt{ then } neg.s \texttt{ else \#.}s + \texttt{\#.}t \texttt{ fi}$$

In any evaluation, it is likely that both dimensions t and s will be present in the context. For example, if a request like the following were made:

$$\text{requesting the value of } N \text{ at } \{t \mapsto -5, s \mapsto -6\}$$

then the value is -11, and uses both dimensions s and t. But we may eventually see demands of the form:

$$\text{requesting the value of } N \text{ at } \{t \mapsto 4, s \mapsto 2\}$$

The calculation of this value will ignore the t-ordinate and produce the value -2. Ignoring the t-ordinate is not the problem; the problem is that the cache entry:

$$\text{value of } N \text{ at } \{t \mapsto 4, s \mapsto 2\} \text{ is } -2$$

will have $t \mapsto 4$ as part of its "tag". This information is irrelevant and should not be included in the tag. Otherwise, the cache will fill with duplicated entries:

$$\begin{aligned}&\text{value of } N \text{ at } \{t \mapsto 0, s \mapsto 2\} \text{ is } -2\\ &\text{value of } N \text{ at } \{t \mapsto 1, s \mapsto 2\} \text{ is } -2\\ &\text{value of } N \text{ at } \{t \mapsto 2, s \mapsto 2\} \text{ is } -2\\ &\ldots\end{aligned}$$

which should all be replaced by the generic entry:

$$\text{value of } N \text{ at } \{s \mapsto 2\} \text{ is } 2$$

The problem is not in producing suitably generic entries; as a computation proceeds, the interpreter could keep a running tally of those dimensions whose values are actually required, and tag the cache entry with the coordinates of just those dimensions.

The problem finally appears when we have to search the cache. Suppose the current request is for N at $\{t \mapsto 4, s \mapsto 2\}$. When we search the cache for the requested value,

what tag do we look for? We need to know which dimensions are actually required in the computation, and since we are trying to avoid performing the computation in the first place, there is no *a priori* reason that we should have this information.

If there are only two dimensions to begin with, the problem is not very serious. We could search first with the tag $\{t \mapsto 4, s \mapsto 2\}$, then try $\{t \mapsto 4\}$, then (successfully) try $\{s \mapsto 2\}$. But suppose there are dozens of dimensions to begin with, or that users can declare their own dimensions, or that there are in fact infinitely many. The cache scheme breaks down.

The pLucid interpreter [43] never solved this problem, which is one of the reasons that the multidimensionality was undocumented. Fortunately, many of the programs people wrote in practice did not cause severe duplication of cache entries. But this was not always the case, and would have been too much to hope for in the presence of, say, dimensions as function parameters.

6.4 Lazy tags

Wadge and Faustini proposed a possible solution in the late 1980s [18]. The idea is that during eduction and caching, even the tags are generated using a demand-driven scheme. It is this solution which we will retain in our implementation.

As indicated above, when the value of a variable is computed, a tally is kept, and only referenced dimensions are included in the tag. The key idea is that searching a value in the cache proceeds as a sort of dialogue, in which the searcher gradually assembles the tag associated with the value sought.

Suppose that we want the value of a variable X in the presence of a large number of dimensions, most of which will turn out to be irrelevant.

The first step is to ask (optimistically) for the value of X with an empty tag. The hope here is that X is a constant. There are three possibilities:

1. The cache provides a value—say, 42—then X is indeed the constant 42 and the search concludes successfully.

2. The cache finds no entry at all with the empty tag, then we conclude that nothing about X has yet been stored in the cache and we must proceed with the calculation.

3. The cache returns a dimension (i.e., a name, not an ordinate). Suppose that the name returned is "z". This means that we must provide the z-ordinate before we can retrieve any more information about X from the cache. In this case, the process is repeated, after providing the z-ordinate.

Suppose that the third option occurred, and that the current z-ordinate is 8. We ask for the cache for information about X tagged with $\{z \mapsto 8\}$. Once again, we have the same three possibilities. This time, we could have:

1. The cache returns a value, say, 64. This means that X has the value 64 whenever the z-ordinate has the value 8, and our search is successful.

2. There is no entry for X with tag $\{z \mapsto 8\}$. This means the value we need is yet to be computed, and we must proceed with the computation (adding appropriate entries to the cache as we do so).

3. The cache returns another dimension—say, u. This means that for $\{z \mapsto 8\}$ we also need the u-ordinate to get a value of X.

This time, suppose that we have the second option, which means that we must proceed with the computation. In doing so, there are two possibilities:

1. A value is returned, say, 42. The cache now records the fact that X has the value 42 at the tag $\{z \mapsto 8\}$.

2. Another dimension, or set of dimensions, say, $\{u\}$, is returned by the evaluator. The cache records that the u-ordinate is required for the value of X at $\{z \mapsto 8\}$, and we continue with the computation until case (1) occurs.

In this way we eventually either (1) build up the tag that retrieves the required value or (2) learn that we will have to compute it.

At the same time, the cache is building up that same hierarchy of dimensions needed to compute the value of X. Then, whenever a value from the cache is requested, the cache goes through each level of the hierarchy, either returning the dimensions needed to go further, or if those dimensions are available, moving to the next level until either a dimension is required or a value is present.

6.5 Caching TransLucid

The next section (§6.6) presents the complete cached semantics for TransLucid. The semantic rules describe a maximally parallel interpreter with a centralized cache. To achieve this, we define *threads*, each with an associated *clock*, and their interaction with the cache. The clock is necessary to synchronise the result of a computation when several threads attempt to compute the same thing at the same time. Rather than all threads computing the same value, all but one of the threads wait for that one thread, then their clocks are synchronised.

The semantic rules are for the syntax of Operational TransLucid, but with a modification to the `wheredim` clause, explained below, along with some assumptions about the structure of a program.

6.5.1 Assumptions

The cache-based implementation does not fully implement Operational TransLucid (Chapter 5). Rather, it is assumed that a static semantics is applied to TransLucid programs, and only those passing are implementable using a cache.

The denotational semantics for TransLucid allows the context to be passed around explicitly, simply by writing `#`. In the cache-based implementation, the context must always appear in an expression of the form `#`.E, i.e., the context may only appear in a

context-query situation; an arbitrary context may not be passed from one part of the program to another. In addition to simplifying the implementation, this choice can also be defended from a security point of view: the context can only reveal the ordinates of dimensions computed explicitly within the program.

The denotational semantics generates different dimensions for a given local dimension identifier each time that the local `wheredim` clause in which it is declared is entered, by using a list to encode the path through evaluation. This does not work for the cached semantics; for a variable defined inside a `wheredim` clause, the dimensions in which it varies will change every time the `wheredim` clause is evaluated, almost completely negating the point of the cache. For example, in any one evaluation of Fibonacci numbers, the cache would be useful, since the local dimension is allocated once. But if another Fibonacci number is requested, a new dimension will be allocated, making the previous memoisation useless. For a computation of a factorial, the cache would be completely useless.

To solve this, we allocate dimensions in a different way to the denotational semantics. Rather than allocate a different dimension for every path through the evaluation tree, we only allocate a different dimension when the evaluation of a `wheredim` clause re-enters itself; we guarantee that each dimension allocation at the same depth will always be the same. Most programs presented so far in this dissertation do not use recursion, and therefore do not enter the same `wheredim` clause multiple times. The only program that does is quick sort (§4.12.5), which when run in the cached implementation is orders of magnitude faster than in the naive implementation.

However, this allocation of dimensions is no longer correct for all programs. Fortunately, it turns out that the programs for which this allocation is a problem are either error cases, or are so bizarre that comprehending them, let alone writing them, is difficult, and they do not solve any meaningful problem. For the case that is already an error, we can make a clear statement. An abstraction defined inside a `wheredim`, and returned from that clause, must not have a local dimension identifier in its rank. This is obviously an error, because the dimension does not exist outside the `wheredim`, and therefore the array defined by the abstraction cannot be indexed. However, using the cache-based implementation, a valid result might be produced, rather than an error. This is analogous to returning the address of an automatic variable in `C`, which is undefined according to the `C` language standards.

As far as determining the boundary between meaningful and bizarre programs, the ideas about what constitutes a valid program for the cache are less clear. If an abstraction is created which freezes a local dimension, and is returned from the `wheredim` defining that dimension, then is later passed back into another instance of the same `wheredim` clause, cached evaluation may not give the correct results. However, we have yet to find any useful program that makes use of this: in fact, we have yet to find any *comprehensible* program that makes use of this.

6.5.2 Syntax

The cache semantics is for *tagged* Operational TransLucid expressions, written $E^{\mathcal{Q}}$. The only difference between the Operational TransLucid syntax and the tagged Operational TransLucid syntax is for the `wheredim` clause:

$$\begin{aligned} E^{\mathcal{Q}} ::= & \cdots \\ & \mid E_0^{\mathcal{Q}} \texttt{ wheredim}_W\ q\ \psi_q\ \phi_{x_i} \leftarrow E_i^{\mathcal{Q}} \texttt{ end} \end{aligned}$$

In the tagged `wheredim` clause, $q \in \mathbb{N}$ is the clause's *label*, and ψ_q is an *other* dimension, whose ordinate will always be a natural number. The transformation $\mathcal{Q}$ takes an Operational TransLucid expression E to produce $E^{\mathcal{Q}}$. For a given expression $E^{\mathcal{Q}}$, we write $Q(E^{\mathcal{Q}})$ for the set of all the $\texttt{wheredim}_W$ labels inside $E^{\mathcal{Q}}$.

Tagging the `wheredim` clauses allows us to allocate dimensions by keeping track of the depth of evaluation through each $\texttt{wheredim}_W$ clause. For each $q \in Q(E)$, the initial ψ_q ordinate is zero, and each entry into a $\texttt{wheredim}_W$ clause tagged q will increment its ψ_q-ordinate. This way, dimensions are allocated deterministically based on the depth of evaluation through each $\texttt{wheredim}_W$ clause. We can reuse the χ dimension allocation from Chapter 2. By making a two-element list $\langle q, \kappa(\psi_q)\rangle$, and by using the index i of the dimension being allocated, we allocate dimensions $\chi^i_{\langle q,\kappa(\psi_q)\rangle}$, as will be seen in Rule (6.17), p.93.

6.5.3 Definitions

For evaluation by eduction to work, it is necessary to pass a mapping from identifiers to expressions through the semantic rules, rather than a mapping to intensions, as for the denotational semantics.

Definition 19. *Let Σ be a signature, X a set of identifiers, then* $\mathbf{Env}^*(X, \mathbf{Expr}(\Sigma, X))$ *is the set of* expression environments *over Σ and X, i.e., mappings $\xi : X \mapsto \mathbf{Expr}(\Sigma, X)$.*

Given the increasing availability of parallelism in the computing infrastructure, at both the fine-grain and the coarse-grain levels, the rules provided below are designed to encourage maximal parallelism.

Definition 20. *A* thread *is a list of natural numbers, $w \in \mathbb{N}^*$. If $i \in \mathbb{N}$, then we write $w_i = w \mid\mid i$ to denote the appending of i to the list w. We write $w' \leq w$ to mean that thread w' is an ancestor to thread w, i.e., that w' is a prefix of w. The initial thread is written ϵ.*

In the cached rules, the evaluation of an expression E will require the evaluation of subexpressions E_i, $i = 1..n$. The evaluation of expression E will take place using a thread w, and the calculation of each of the subexpressions E_i will take place using a thread w_i.

Definition 21. *A* clock *is a natural number $t \in \mathbb{N}$.*

Each thread and the cache will have its own clock. Each time that a thread uses the cache, the clocks of both are set to the fastest clock. Each time that a thread uses the results from other threads, its clock is advanced to the fastest clock. All clocks start from zero.

6.5.4 The cache

A cache β is a synchronous, reactive machine with four internal variables:

- β.ck $\in \mathbb{N}$ is a clock (a counter), initially 0;
- β.age $\in \mathbb{N}$ is the *global retirement age*, initially 2, for the garbage collector;
- β.data contains the *nodes* of the cache;
- β.limit contains the *maximum number of nodes* in the cache.

The variable β.data contains a number of nodes γ_j, $j \in \mathbb{N}$. Each γ_j is either a leaf node or an internal node. If γ_j is a leaf node, it either stores an ordinary value d_j or a value $\texttt{calc}\langle w\rangle$, meaning that thread w is currently responsible for computing this entry. If γ_j is an internal node, it will hold a pair consisting of 1) a set Δ_j of dimensions whose ordinates must be provided, and 2) a set of pairs $(\mathit{ordinates}_{ji}, \mathit{nextNode}_{ji})$, meaning that when the dimensions in Δ_j take on the $\mathit{ordinates}_{ji}$, then the next node is $\mathit{nextNode}_{ji}$.

As a result, β.data acts a decision tree for each variable x, keeping track of the dimensions that are needed to access a value. Suppose that thread w is seeking the value for variable x and context κ in β.data. The root of the tree for x gives the stored value for evaluating variable x with the empty context $\emptyset$. Should $\beta.\text{data}(x, \emptyset)$ be an ordinary value d, then whatever the context κ, the value of x in κ will be d. Should $\beta.\text{data}(x, \emptyset)$ be a set of dimensions Δ_1, then the cache is asking that the ordinates for the dimensions in Δ_1 be provided. So the process is started over with $\beta.\text{data}(x, \kappa \lhd \Delta_1)$, and is repeated until ultimately an ordinary value d is returned or else the cache inserts a $\texttt{calc}\langle w\rangle$ value, meaning that thread w is responsible for calculating this entry.

In summary, should the value for variable x and context κ actually be in β.data, then there would be n mutually exclusive sets of dimensions $\Delta_1, \ldots, \Delta_n$, $n \in \mathbb{N}$, such that:

$$\begin{aligned}
\beta.\text{data}(x, \emptyset) &= \Delta_1 \\
\beta.\text{data}(x, \kappa \lhd \Delta_1) &= \Delta_2 \\
\beta.\text{data}\big(x, \kappa \lhd (\Delta_1 \cup \Delta_2)\big) &= \Delta_3 \\
&\ldots \\
\beta.\text{data}\big(x, \kappa \lhd (\Delta_1 \cup \cdots \cup \Delta_{n-1})\big) &= \Delta_n \\
\beta.\text{data}\big(x, \kappa \lhd (\Delta_1 \cup \cdots \cup \Delta_n)\big) &= d
\end{aligned}$$

We can then write that $\beta.\text{data}^*(x, \kappa) = d$.

The cache uses the retirement plan mentioned in Section 6. Each node γ_j has an age, γ_j.age, initialized to zero when the node is created. Every time that a node is retrieved,

its age is reset to zero. The garbage collector keeps a global retirement age, and every garbage collection run, the global retirement age is decreased by one. On the other hand, if at any point, a node whose age is greater than the global retirement age is retrieved, the global retirement age is increased to the age of the node before that node is set back to zero. Every garbage collection run, the age of every node is increased by one. Only leaf nodes can be collected. Should, during any one garbage collection run, an internal node have all of its children collected, then it can be collected immediately if its age is greater than the retirement age.

In the semantic rules, rather than passing an immutable cache object around, which is modified by certain rules, we pass a global cache object that has operations with side effects. The cache responds to two different instructions generated by threads, and one instruction generated internally:

- $(v', t') = \beta.\text{find}(x, \kappa, w, t)$
  - If $t > \beta.\text{ck}$, advance $\beta.\text{ck}$ to t.
  - If $\beta.\text{data}(x, \kappa) = \texttt{calc}\langle w' \rangle$ and $w' \leq w$, hang (do not return).
  - If $\beta.\text{data}(x, \kappa)$ is not defined, let $\beta.\text{data}(x, \kappa)$ be $\texttt{calc}\langle w \rangle$ and advance $\beta.\text{ck}$ by 1. Should β receive more than one $\beta.\text{find}(x, \kappa, w_i, t_i)$ instruction at the same instant $\beta.\text{ck}$, the w is nondeterministically chosen from among the w_i.
  - For all the nodes γ_j in the chain which stores $\beta.\text{data}^*(x, \kappa)$, if $\gamma_j.\text{age} > \beta.\text{age}$, then set $\beta.\text{age} = \gamma_j.\text{age} + 1$, then set $\gamma_j.\text{age} = 0$.
  - If the number of nodes is greater than $\beta.\text{limit}$, then run $\beta.\text{collect}()$.
  - Return $(\beta.\text{data}(x, \kappa), \beta.\text{ck})$.
- $(v', t') = \beta.\text{add}(x, \kappa, w, t, v)$
  - If $t > \beta.\text{ck}$, advance $\beta.\text{ck}$ to t.
  - If $\beta.\text{data}(x, \kappa) \neq \texttt{calc}\langle w \rangle$, hang (do not return).
  - Let $\beta.\text{data}(x, \kappa)$ be v and advance $\beta.\text{ck}$ by 1.
  - Return $(\beta.\text{data}(x, \kappa), \beta.\text{ck})$.
- $\beta.\text{collect}()$
  - $\beta.\text{age} = \beta.\text{age} - 1$.
  - For each node γ_j in the cache, in a post-order traversal of the tree:
    - If $\gamma_j.\text{age} \geq \beta.\text{age}$ then remove the node if it has no children, and if it is not holding the value $\texttt{calc}\langle w \rangle$.
    - If γ_j has not been collected, increment $\gamma_j.\text{age}$ by one.

6.5.5 Validity of the cache

In showing that the semantics are valid, we will not attempt to prove that the cache semantics produces the same result as the operational semantics, since we know that it does not, in general, due to the allocation of dimensions, but that for all reasonable programs it gives the same result. Therefore, there are two separate issues that need to be considered for these semantics to be valid, and therefore useful for evaluation: 1) The cache must correctly discover all dimensions of relevance for the value of an intension, with the correct hierarchy, for *sane* programs; 2) we must be able to put some definition on what a *sane* program is.

For the first point, it is necessary to show that the rules correctly pass all appropriate dimensions back to the cache, and in the correct order, so that the cache does not end up *broken*, with an invalid hierarchy of dimensions being stored. Therefore, we argue that the cache maintains an invariant, and that each cache function has preconditions for its use, which the semantics must respect. For the second, the definition of a sane program will be much less rigorous, and no attempt is made to detect whether a program is valid for the cache or not; this is left as future work.

The following invariant must be maintained in the cache. If $\beta.\mathrm{data}(x, \kappa)$ exists, then there exists a chain of entries $\beta.\mathrm{data}(x, \kappa_i)$, $i = 0..n$, where

$$\begin{aligned} \kappa_i &= \kappa \lhd (\Delta_0 \cup \cdots \cup \Delta_i) \\ \Delta_0 &= \emptyset \\ \forall j, k \text{ s.t. } j \neq k \quad \Delta_j \cap \Delta_k &= \emptyset \\ \kappa_n &= \kappa. \end{aligned}$$

Should $\beta.\mathrm{data}(x, \kappa) = d$, where d is not a set Δ, then there cannot exist a κ' such that $\mathrm{dom}\ \kappa \subset \mathrm{dom}\ \kappa'$, $\kappa = \kappa' \lhd \mathrm{dom}\ \kappa$, and $\beta.\mathrm{data}(x, \kappa')$ exists.

6.6 Semantics

The cached semantic rules presented below are of the form

$$[\![E]\!]^{\mathrm{w}} \xi \kappa \Delta \beta w t,$$

meaning an expression E is evaluated in a given environment ξ, at a particular context κ, using a cache β, a set Δ of dimensions, a thread w and a clock t. The interaction between each of these components is explained in the following paragraphs.

We begin by supposing that we are evaluating a demand for a variable $x = E$, in some context κ, where the variable x is being cached. There are no cache entries, so the job of the evaluator is to help the cache discover the dimensions of relevance in computing the value of E in the current context. Suppose that at some point inside E the ordinate of some dimension, say δ, is required. The evaluator needs to know at this point that the cache does not know about dimension δ, so that it can inform the cache that dimension δ

is required in computing the value of E. This is achieved by the cache setting Δ to $\emptyset$; when a lookup for dimension δ is reached, and δ is not in Δ, the evaluator returns to the cache, saying that it needs dimension δ.

The cache records this fact, along with the ordinate of δ, then attempts evaluation again, this time setting $\Delta = \{\delta\}$. Now suppose that some dimension γ is now required, this process is repeated, and the cache records this as a child node under δ, and evaluates again, setting $\Delta = \{\gamma, \delta\}$. This process is repeated until no more dimensions are required, and the evaluator returns a value instead of a set of dimensions.

However, should evaluation go through another cache node, this whole process is repeated from the start, for that cache node. As a result, each cache node builds its entries correctly. This back-and-forth process might seem to be overkill, but it correctly builds the hierarchy of dimensions required to compute a value, and experiments have shown that the time required for the initial discovery phase is quickly made up for in the time saved repeating computations.

The threads w and clocks t passed through the semantics play an important part in this computation. Suppose that while thread w is computing the value of an expression, thread w' wants that value in the same context. Rather than recompute that value, thread w' waits for the answer to come back from w. As a side effect of this, loops in the definitions can be detected, rather than a program running forever. If $w' < w$, then the thread w' is a descendant of w, meaning that the computation being done on thread w' was spawned by w. Therefore, the computation for E at κ requires E at κ, and there is a loop. In the semantics given below, the result is left undefined, but our implementation handles this case with error values.

Definition 22. *Let E be an Operational TransLucid expression, satisfying the conditions of §6.5.1, ι be an interpretation, ζ be an environment, and κ be a context such that $[\![E]\!]\iota\zeta\kappa$ is defined. Then, the* cached semantics *for E are given by*

$$\left[\!\left[E^{\mathcal{Q}}\right]\!\right]^{W} \iota\xi_0\kappa_0\Delta_0\beta_0 w_0 t_0$$

such that

$$\begin{aligned}
\xi_0 &= \emptyset \\
\kappa_0 &= \kappa \dagger \left\{\psi_q \mapsto 0 \mid q \in Q\left(E^{\mathcal{Q}}\right)\right\} \\
\Delta_0 &= \mathrm{dom}(\kappa) \cup \left\{\psi_q \mid q \in Q\left(E^{\mathcal{Q}}\right)\right\} \\
w_0 &= \epsilon \\
t_0 &= 0
\end{aligned}$$

and β_0 is the least β such that $\forall x, \kappa$, if $\zeta(x)(\kappa)$ is defined, $\beta_0.\mathrm{data}^*(x, \kappa) = \zeta(x)(\kappa)$, and where the rules for $[\![\cdot]\!]^{\mathrm{w}}$ are given in §6.6.1. Where the rules refer to v, this can either be $d \in \mathbf{D}$ or a set Δ of dimensions requested.

Here, it is necessary for β_0 to be preinitialised with every entry of the inputs to E. In the denotational semantics, the inputs are in ζ, which is a mapping from identifiers

to intensions. For the cached rules, we use ξ, a mapping from identifiers to expressions, and the inputs to E are not necessarily specified as expressions. Therefore, we place the contents of ζ for the inputs into the initial cache β_0.

Proposition 6. *Let E, ι, ζ, κ be as in Definition 22. If $[\![E]\!]\iota\zeta\kappa$ is defined, then*

$$[\![E]\!]\iota\zeta\kappa \equiv \left[\!\left[E^{\mathcal{Q}}\right]\!\right]^W \iota\xi_0\kappa_0\Delta_0\beta_0 w_0 t_0.$$

We will not prove the proposition, as it would parallel exactly the explanation of the rules that follow, and would take up too much space.

6.6.1 The cached semantics rules

$$[\![\underline{\phi_x}]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \tag{6.1}$$
$$= \begin{cases} \big(\{\phi_x\}, t\big), & \phi_x \notin \Delta \\ \big(\kappa(\phi_x), t\big), & \text{otherwise} \end{cases}$$

$$[\![{}^m c]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \tag{6.2}$$
$$= \big(\iota({}^m c), t\big)$$

$$\left[\!\left[[E_{i0} \leftarrow E_{i1}]_{i=1..m}\right]\!\right]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \tag{6.3}$$
$$= \text{let } (v_{ij}, t_{ij}) = [\![E_{ij}]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta w_{ij} t$$
$$\text{in } \begin{cases} \big(\bigcup_{ij} \Delta_{ij}, \max(t_{ij})\big), & v_{ij} \text{ is of form } \Delta_{ij} \\ \big(\{v_{i0} \mapsto v_{i1}\}, \max(t_{ij})\big), & \text{otherwise} \end{cases}$$

$$[\![\lambda^{\mathrm{b}}_{\circ}\ \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \tag{6.4}$$
$$= \text{let } (v_i, t_i) = [\![E_i]\!]^{\mathrm{w}}\iota\zeta\kappa\Delta\beta w_i t$$
$$\text{in } \begin{cases} \big(\bigcup_i \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i \\ \big(\bigcup_i \{v_i\}, \max(t_i)\big), & v_i \notin \kappa \\ \big(\lambda d.\lambda\beta.\lambda w\lambda t. & \\ \quad [\![E_0]\!]^{\mathrm{w}}\iota\zeta\big(\{\phi_x \mapsto d\} \dagger \big(\kappa \lhd (\{v_i\} \cup \Phi)\big)\big)(\{v_i\} \cup \{\phi_{x_j}\})\beta wt, & \\ \quad \max(t_i)\big), & \text{otherwise} \end{cases}$$

$$\left[\!\left[E_0\,.\,(E_i)_{i=1..m}\right]\!\right]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \tag{6.5}$$
$$= \text{let } (v_i, t_i) = [\![E_i]\!]\iota\xi\kappa\Delta\beta w_i t$$
$$\text{in } \begin{cases} \big(\bigcup_i \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i \\ v_0(v_i)\beta w\big(\max(t_i)\big), & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
&[\![\texttt{if } E_0 \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ fi}]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \qquad (6.6)\\
&= \text{let } (v_0, t_0) = [\![E_0]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt\\
&\quad \text{in} \begin{cases} (\Delta_0, t_0), & v_0 \text{ is of form } \Delta_0 \\ [\![E_1]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta w t_0, & v_0 \equiv \mathit{true} \\ [\![E_2]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta w t_0, & v_0 \equiv \mathit{false} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{\#}.E_0]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \qquad (6.7)\\
&= \text{let } (v_0, t_0) = [\![E_0]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt\\
&\quad \text{in} \begin{cases} (\Delta_0, t_0), & v_0 \text{ is of form } \Delta_0 \\ \big(\{v_0\}, t_0\big), & v_0 \notin \Delta \\ \big(\kappa(v_0), t_0\big), & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0 \texttt{ @ } E_1]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \qquad (6.8)\\
&= \text{let } (v_1, t_1) = [\![E_1]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt\\
&\quad \text{in} \begin{cases} (\Delta_1, t_1), & v_1 \text{ is of form } \Delta_1 \\ [\![E_0]\!]^{\mathrm{w}} \iota\xi(\kappa \dagger v_1)\big(\Delta \cup \mathrm{dom}(v_1)\big)\beta w t_1, & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\uparrow_\circ \{E_i\}_{i=1..m}\ \Phi\ E_0]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \qquad (6.9)\\
&= \text{let } (v_i, t_i) = [\![E_i]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta w_i t\\
&\quad \text{in} \begin{cases} \big(\bigcup \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i \\ \big(\bigcup_i \{v_i\}, \max(t_i)\big), & v_i \notin \kappa \\ \big(\lambda\kappa_a.\lambda\Delta_a.\lambda\beta_a.\lambda w_a.\lambda t_a. & \\ \quad [\![E_0]\!]^{\mathrm{w}} \iota\xi\big(\kappa_a \dagger \big(\kappa \lhd (\{v_i\} \cup \Phi)\big)\big)\big(\Delta_a \cup \{v_i\} \cup \Phi\big) w_a t_a, & \\ \quad \max(t_i)\big), & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\downarrow E_0]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \qquad (6.10)\\
&= \text{let } (v_0, t_0) = [\![E_0]\!] \iota\xi\kappa\Delta\beta wt\\
&\quad \text{in} \begin{cases} (\Delta_0, t_0) & v_0 \text{ is of form } \Delta_0 \\ v\kappa\Delta\beta w t_0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\lambda^{\mathrm{v}}_{\circ}\ \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \qquad (6.11)\\
&= \text{let } (v_i, t_i) = [\![E_i]\!]^{\mathrm{w}}\iota\zeta\kappa\Delta\beta w_i t\\
&\quad \text{in } \begin{cases}
\big(\bigcup_i \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i\\
\big(\bigcup_i \{v_i\}, \max(t_i)\big), & v_i \notin \kappa\\
\big(\lambda d_a.\lambda\kappa_a.\lambda\Delta_a.\lambda\beta_a.\lambda w_a.\lambda t_a. & \\
\quad [\![E_0]\!]^{\mathrm{w}}\iota\zeta & \\
\quad \big(\kappa_a \dagger \{\phi_x \mapsto d_a\} \dagger \big(\kappa \lhd (\{v_i\} \cup \Phi)\big)\big) & \\
\quad \big(\Delta_a \cup \{v_i\} \cup \{\phi_x\} \cup \Phi\big) & \\
\quad \beta_a w_a t_a, & \\
\quad \max(t_i)\big), & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![E_0\,!\,E_1]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \qquad (6.12)\\
&= \text{let } (v_i, t_i) = [\![E_i]\!]\iota\xi\kappa\Delta\beta w_i t\\
&\quad \text{in } \begin{cases}
\big(\bigcup_i \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i\\
v_0 v_1 \kappa\Delta\beta w\big(\max(t_i)\big), & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![x]\!]^{\mathrm{w}}\iota\xi\kappa\Delta\beta wt \qquad (6.13)\\
&= \text{let } (v_0, t_0) = [\![x]\!]^1\iota\xi\kappa\emptyset\Delta\beta wt\\
&\quad \text{in } \begin{cases}
(v_0, t_0) & v_0 \text{ is of form } \Delta_0\\
\beta.\mathrm{find}(x, \kappa \lhd \Delta, t, w), & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![x]\!]^1\iota\xi\kappa\Delta\Delta'\beta wt \qquad (6.14)\\
&= \text{let } (v_0, t_0) = [\![x]\!]^2\iota\xi\kappa\Delta\beta wt\\
&\quad \text{in } \begin{cases}
[\![x]\!]^1\iota\xi\kappa(\Delta \cup \Delta_0)\Delta'\beta wt, & v_0 \text{ is of form } \Delta_0 \text{ and } \Delta_0 \subseteq \mathrm{dom}(\kappa) \text{ and } \Delta_0 \subseteq \Delta'\\
(\Delta_0 - \Delta', t_0) & v_0 \text{ is of form } \Delta_0 \text{ and } \Delta_0 \not\subseteq \Delta'\\
(v_0, t_0), & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&[\![x]\!]^2\iota\xi\kappa\Delta\beta wt \qquad (6.15)\\
&= \text{let } (v_0, t_0) = \beta.\mathrm{find}(x, \kappa \lhd \Delta, t, w)\\
&\quad \text{in } \begin{cases}
\text{let } (v_1, t_1) = [\![\xi(x)]\!]\iota\xi\kappa\Delta\beta wt_0 & \\
\text{in } \beta.\mathrm{add}(x, \kappa \lhd \Delta, t_1, w, v_1), & v_0 \text{ is of form calc}\langle w'\rangle \text{ and } w' \equiv w\\
[\![x]\!]^2\iota\xi\kappa\Delta\beta w(t_0 + 1), & v_0 \text{ is of form calc}\langle w'\rangle \text{ and } w' \not\leq w\\
(v_0, t_0), & v_0 \text{ is not of form calc}\langle\cdots\rangle
\end{cases}
\end{aligned}
$$

$$[\![E_0 \texttt{ wherevar } x_i = E_i \texttt{ end}_{i=1..m}]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta wt \tag{6.16}$$
$$= [\![E_0]\!]^{\mathrm{w}} \iota\big(\xi \dagger \{x_i \mapsto E_i\}\big)\kappa\Delta\beta wt$$

$$[\![E_0 \texttt{ wheredim}_W\ Q\ \psi_Q\ \psi_{x_i} \leftarrow E_i \texttt{ end}]\!]\iota\xi\kappa\Delta\beta wt \tag{6.17}$$
$$= \text{let } \delta_i = \chi^i_{\langle q,\kappa(\psi_q)\rangle}$$
$$(v_i, t_i) = [\![E_i]\!]^{\mathrm{w}} \iota\xi\kappa\Delta\beta w_i t$$
$$\text{in } \begin{cases} \big(\bigcup_i \Delta_i, \max(t_i)\big), & v_i \text{ is of form } \Delta_i \\ [\![E_0]\!]^{\mathrm{w}} \iota\xi\big(\kappa \dagger \{\psi_q \mapsto \kappa(\psi_q) + 1, \phi_{x_i} \mapsto v_i\}\big)\Delta\beta w\big(\max(t_i)\big) & \text{otherwise} \end{cases}$$

6.6.2 Comments on the rules

All of the evaluation rules that involve some computation of subexpressions take into account the fact that any of those subexpressions might evaluate to a demand for one or more dimensions. Should this occur, computation does not continue, and all of these demands are aggregated into a larger demand, which becomes the result. This behaviour is consistent with the cache invariant, due to the behaviour of rule $[\![x]\!]^{\mathrm{w}}$ (explained below), which only evaluates its defining expression with respect to the dimensions already known by the cache. Then, if any further dimensions are required, these rules pass them back to the nearest $[\![x]\!]^{\mathrm{w}}$ node.

Rule (6.2) corresponds to the evaluation of constants. They evaluate to the same value in all contexts and do not interact with the cache in any manner.

Rules (6.3)–(6.6), (6.9)–(6.12) and (6.16) are trivial modifications of their equivalent rules from the denotational semantics. They simply pass on demands for dimensions resulting from the evaluation of subexpressions.

Rules (6.1) and (6.7), for context queries, is where demands for dimensions are generated. Any time the context is looked up, the requested dimension must be available in Δ. If it is not in Δ, then a demand is returned for that dimension.

Rule (6.8), for context perturbations, adds to Δ any dimensions whose ordinates have been changed. This is done because any dimension that is changed cannot possibly affect the value of an identifier being cached further up the tree. The dimension is then added to Δ so that if its value is later requested, a demand for it is not generated.

Rules (6.13)–(6.15) are where the interaction with the cache takes place. We will use a simple example to illustrate how these three rules interact with the cache. Suppose that we have a variable A with the following definition:

$$\texttt{var } A = \texttt{\#}.d$$

Clearly A depends on dimension d. Now, suppose that we request value of A at the context $\{d \mapsto 4\}$, then the cache should record that A at $\{d \mapsto 4\}$ is 4. If the evaluator just started evaluating the body of A in the current context, it would return the value 4 and the cache would not learn anything. Therefore, it is necessary for the cache to start *from scratch* to learn the dimensions relevant to computing A. This is why rule $[\![x]\!]^{\mathrm{w}}$ calls $[\![x]\!]^1$ with $\emptyset$ as the fourth argument, corresponding to the dimensions known about by the current cache node. Then, initially, rule $[\![x]\!]^1$ hands evaluation to $[\![x]\!]^2$, which tries to find an entry in the cache. The context here is $\emptyset$, and there is no cache entry. So the cache returns $\texttt{calc}\langle w\rangle$, indicating that this thread should calculate the value with no context.

It goes ahead and does that, and in this case, will come back with the result that it needs dimension d. Therefore, the fact that at context $\emptyset$, dimension d is required, will be entered into the cache. Control then returns back to $[\![x]\!]^1$, and since dimension d is present in κ, the same process will repeat, but this time with the fourth argument of $[\![x]\!]^1$ set to $\{d\}$, since the cache now knows about d. This time, an answer will be returned, and the fact that A at $\{d \mapsto 4\}$ is 4 can be entered into the cache.

Control then returns back to $[\![x]\!]^{\mathrm{w}}$. At this point, we could just return the value, but then no information about the dimensions used to compute x would be passed back up to any cache nodes above the current point of evaluation. Instead, we find the value in the cache, using the original Δ, which holds the dimensions known about by the previous cache node. If there are not enough dimensions to compute the value, then the cache will simply return the dimensions required, and the higher-up cache node will have discovered the relevant dimensions. This back-and-forth between cache nodes allows them to discover the relevant dimensions for a computation.

The cache invariant is maintained here by rule $[\![x]\!]^2$, because every time it finds a value in the cache, and receives back a `calc` value, it computes that value and enters it into the cache, ensuring that the find and add operations are paired. If another thread w' were to get back a `calc` value, it would be for the original thread w, and the cache guarantees that $w' \not\leq w$, so w' would simply wait until the answer comes back from w.

Rule (6.17): The $\texttt{wheredim}_W$ clause is similar to the one from Operational TransLucid. The difference is in how dimensions are allocated. The rule allocates dimensions using the χ dimensions from Chapter 2, by making a two-element list $\langle q, \kappa(\psi_q)\rangle$ and the index of the dimension being allocated, i. Since q is unique, $\kappa(\psi_q)$ is the depth of entry into the current $\texttt{wheredim}_W$ clause, and i is a unique index for the dimension being allocated, $\chi^i_{\langle q,\kappa(\psi_q)\rangle}$ will be a unique dimension for that $\texttt{wheredim}_W$ clause, and for the current depth of recursion. Then, the ψ_q-ordinate is incremented to evaluate the body E_0. When the current $\texttt{wheredim}_W$ clause has not previously been entered into, the ψ_q-ordinate will be zero. This has the result that if a $\texttt{wheredim}_W$ clause does not recursively use itself, then the local dimensions allocated will always be the same. Then, evaluating E_0 at $\{\psi_q \mapsto \kappa(\psi_q) + 1\}$ ensures that if a $\texttt{wheredim}_W$ clause does recursively use itself, the dimensions allocated will be the same at each depth.

6.7 Conclusion

This chapter has presented an effective semantics for the memoisation of TransLucid expressions, which solves a problem that first appeared in 1985 when the pLucid interpreter introduced multidimensionality, which was undocumented, due to the lack of a solution to that problem [43]. Furthermore, experiments have shown that cached evaluation is several orders of magnitude faster than naive eduction, and despite the change in the allocation of dimensions, works for all problems presented in this dissertation.

The evaluator presented in this chapter is significant, because it provides an upper bound on the execution time of programs, given a naive cached evaluator that does not know anything statically about the program it is running. In addition, it provides a starting point from which any future models of evaluation can be measured against, both for correctness and speed, without resorting to the completely naive interpreter, for which one might wait an incredibly long time for an answer to be returned.

Viewing all computation as storing values in a cache is incredibly useful, because improved methods of evaluation become a matter of putting bounds on the rank of a variable, and on the regions that might be demanded. For example, if it is possible to determine the rank of a variable precisely, which could be only one or two dimensions, then a fixed array in memory could be used rather than the tree built by the cache. This array could be a dynamic size if bounds could not be put on the possible regions to be demanded, but with techniques such as *Abstract Interpretation* [13], even these bounds could be determined, and a static array could be used.

Even with simple static analysis, which is not necessarily guaranteed to be complete, the cache could be used by an implementation that guesses which entries are the most useful to precompute. Then if those values were never used, no harm would be done, if they were used, then the implementation might have just saved some time.

Essentially, all static analysis comes down to determining the appropriate structure with which to cache a variable, and if insufficient information can be decided at compile time, the current cache model is always there as a fallback.

Despite the significance of the results in producing a cached evaluator, there is still one problem left unresolved. The allocation of dimensions presented in §6.5.1 is not correct for all programs, and there is currently no known way to check this whether it is. It would be advantageous if it were possible to check whether a program could be run by the cache and produce the correct result. If not, the cached implementation should refuse to run it.

Chapter 7

Type Inference

This chapter presents a type inference algorithm for TransLucid expressions. The key idea for any static analysis of a TransLucid program, is that the principal type of an object is itself, which is an idea from a 1977 paper by William W. (Bill) Wadge, "Data Types as Objects" [40]. The type inference system that is used in this chapter is a subtyping system with constraints, adapted from the system presented by François Pottier in his 1998 PhD thesis, *Type inference in the presence of subtyping: from theory to practice* [34]. Additionally, in [35], Pottier adds conditional constraints to the system, which allows a constraint to be activated based on the type of another object, used for conditional expressions, which we also adapt to TransLucid, and call *guarded constraints*.

In our type system, typing requirements are expressed as a system of constraints, where the constraints are subtyping requirements between the appropriate types. For example, for a function f, with type $\tau_0 \xrightarrow{v} \tau_1$, and expression x, which has type τ_2, the function application $f\,!\,x$ requires that the constraint $\tau_0 \xrightarrow{v} \tau_1 \leq \tau_2 \xrightarrow{v} \alpha$ be true, which implies the two constraints $\tau_2 \leq \tau_0$ and $\tau_1 \leq \alpha$, where α is the type of the result returned from the function application. What that set of constraints means is that the input to f, the type of x, must be a smaller type than the types allowed as inputs to f. Similarly, the type of the result is allowed to be a greater type than the specified output of f.

In general, to determine if an expression is well-typed, all the constraints implied by the subexpressions of an expression are gathered together into a *constraint graph* (§7.3.4), and if the graph is *closed* (Definition 46), then the expression is well-typed.

The theory underlying the type system presented in this chapter comes completely from Pottier's thesis, along with the addition of conditional constraints. The difference is in the exact types used and the way in which they are manipulated. Additionally, since the concept of running context is particular to the TransLucid programming language, we present a means to infer types for the ordinates of dimensions used in a TransLucid program (§7.5).

After determining if an expression is well-typed, a constraint graph can have a large number of constraints, most of which are not useful, as they only contributed to the type at intermediate stages. So §7.8 onwards presents ways to simplify a constraint graph.

First, we present our ground types (§7.1), which are types with no type variables. These are the monomorphic types such as **intmp** for GNU MP integers, **ustring** for

the Unicode strings, and every atomic object, since the principal type of an object is itself. Then, we introduce type variables to produce polymorphic types (§7.2). Once the types are defined, we can build a *type scheme* (§7.3), which holds all of the information necessary to describe the type of an expression. Once the type scheme has been described, we present how to build a type scheme, with the type inference rules, and then describe the simplifications that can be carried out on a type scheme.

This chapter closes with a presentation of how simplified type schemes are displayed, and with the complete working-through of several examples.

7.1 Ground types

The *ground types* are types with no occurrences of type variables. They are built up from a set of disjoint atomic types. The ground types are atomic objects, atomic types, function types and bottom and top elements. The whole system takes as input a set T of disjoint atomic types, meaning that the actual set of atomic types and objects used can change, and the system will still work. The only requirement is that T contain the type **bool**, containing exactly the values **bool**⟨`true`⟩ and **bool**⟨`false`⟩; from now on we write `true` and `false`.

Definition 23. *An* atomic type *is a name t, to which is associated a* set D_t of atomic objects. *An element a of D_t is written $t\langle a\rangle$, meaning atomic value a of type t.*

Definition 24. *Let T be a set of atomic types $\{t_1, \ldots, t_n\}$ such that $\forall i, j \in 1..n$, $i \neq j \Rightarrow D_{t_i} \cap D_{t_j} = \emptyset$ and $\nexists i, j \in 1..n$, $t_i \in D_{t_j}$. Then we define $D_T = \bigcup_{i \in 1..n} D_{t_i}$.*

In the current interpreter, the atomic types are:

- **bool**, containing the values `true` and `false`;
- **uchar**, for Unicode characters, where value **uchar**⟨'c'⟩ will be written as 'c'.
- **ustring**, for Unicode strings, where value **ustring**⟨"string"⟩ will be written as "string"; and
- **intmp**, for the GNU `mp` (arbitrary precision) integers, where values **intmp**⟨1⟩, **intmp**⟨2⟩, ... will be written as $1, 2, \ldots$.

In our system, we consider an atomic object to itself be a type. We also consider an *explicitly* named union of atomic objects and atomic types to be a type. So, the *basic types* are the atomic objects, the atomic types, and these unions.

Definition 25. *A* basic type $\mathbf{t}$ *is an atomic object, an atomic type, or an arbitrary union of those:*

$$\begin{aligned} \mathbf{t} ::= \;& t \\ \mid \;& t\langle a\rangle \\ \mid \;& \texttt{union}\langle t\langle a\rangle, \ldots, t\langle a\rangle, t, \ldots, t\rangle \end{aligned}$$

Definition 26. *The* denotation *of a basic type is the set of objects making it up, i.e., a subset of* $\mathbf{T} = 2^{T \cup D_T} - \{\emptyset\}$*:*

$$\begin{aligned} \text{denot}(t) &= D_t \\ \text{denot}\big(t\langle a\rangle\big) &= \big\{t\langle a\rangle\big\} \\ \text{denot}\big(\texttt{union}\langle t_1\langle a_1\rangle, \ldots, t_m\langle a_m\rangle, t'_1, \ldots, t'_n\rangle\big) &= \big\{t_1\langle a_1\rangle, \ldots, t_m\langle a_m\rangle\big\} \cup \bigcup_{j=1..n} D_{t'_j} \end{aligned}$$

The set D of the denotational semantics (Chapter 2) corresponds to the union of the denotations of the basic types.

We are now ready to define the ground types. These are the basic types, the minimal *bottom* ($\bot$) element, the maximal *top* ($\top$) element, and the function types, of which there are three kinds.

Definition 27. *The set of* ground types*, denoted by* G *(*$\ni g$*), is defined by the following grammar:*

$$\begin{array}{rll} g ::= & \mathbf{t} & (\textit{the basic types}) \\ | & \bot & (\textit{the bottom element}) \\ | & \top & (\textit{the top element}) \\ | & g \xrightarrow{v} g & (\textit{call-by-value function type}) \\ | & (g, ...) \xrightarrow{b} g & (\textit{base function types, of variable arity}) \\ | & \uparrow g & (\textit{the intension type}) \end{array}$$

As we are interested in subtyping, it is necessary to define a subtyping order $\sqsubseteq$ over the ground types. This order includes obvious ones, such as $42 \sqsubseteq \mathbf{intmp}$, but also describes the way in which functions can be decomposed to make subtypes. The definition is entirely structural for functions, and is defined in the standard way.

Definition 28. *We define a partial order,* $\sqsubseteq$*, over* G*, as follows:*

$\forall g \in G,\ \bot \sqsubseteq g$

$\forall g \in G,\ g \sqsubseteq \top$

$t\langle a\rangle \sqsubseteq t' \quad \text{iff} \quad t \equiv t'$

$t\langle a\rangle \sqsubseteq t'\langle a'\rangle \quad \text{iff} \quad t \equiv t' \wedge a \equiv a'$

$t\langle a\rangle \sqsubseteq \texttt{union}\langle t_1\langle a_1\rangle, \ldots t_m\langle a_m\rangle, t'_1 \ldots t'_n\rangle \quad \text{iff}$

$\quad \exists i \in 1..m, t\langle a\rangle \equiv t_i\langle a_i\rangle \vee \exists j \in 1..n, t \equiv t'_j$

$t \sqsubseteq \texttt{union}\langle t_1\langle a_1\rangle, \ldots t_m\langle a_m\rangle, t'_1 \ldots t'_n\rangle \quad \text{iff} \quad \exists i \in 1..n, t \equiv t_i$

$(g_1 \xrightarrow{v} g_0) \sqsubseteq (g'_1 \xrightarrow{v} g'_0) \quad \text{iff} \quad (g_0 \sqsubseteq g'_0) \wedge (g'_1 \sqsubseteq g_1)$

$((g_{j=1..m}) \xrightarrow{b}_m g_0) \sqsubseteq ((g'_{j=1..m}) \xrightarrow{b}_m g'_0) \quad \text{iff} \quad (g_0 \sqsubseteq g'_0) \ \wedge \bigwedge_{j=1..m} (g'_j \sqsubseteq g_j).$

$(\uparrow g) \sqsubseteq (\uparrow g') \quad \text{iff} \quad (g \sqsubseteq g')$

7.2 Types

The ground types do not suffice. The process of type inference relies on the ability to label the type of an expression, so that the type can be constrained by how it is used. For example, if we have the expression

$$f\,!\,E$$

then type inference is effectively saying, "Whatever the type of E is, it should be smaller than the required input type of f, whatever that is." There are two potential unknowns here: the type of f, and the type of E. To link the information about the input of f and the type of E, the type inference process draws from an infinite supply of *type variables*. A type variable can be used to represent *any* type, and is often constrained either from below, above, or both.

Due to the fact that we are working with subtyping, and that the ground types have a partial order, it makes sense to talk about the least upper bound and greatest lower bound of sets of elements using that order. These are used when an expression can have multiple types, due to a conditional expression, or when an input parameter is used in multiple functions. They can be seen as an approximation of a union of types and an intersection of types, respectively.

Consider the expression

$$\texttt{if } \underline{\phi_x} \texttt{ then } 1 \texttt{ else } 5 \texttt{ fi}$$

In traditional type inference, its type would be something like **intmp**, as both 1 and 5 are integers. However, we know that the only two possible values that can come from the expression are 1 and 5, so the most precise type is in fact the set $\{1, 5\}$, i.e., the union of the type 1 and the type 5.

In fact, it is appropriate to say that the output type should include every object that could be returned, but that more objects could also be included. Ideally, the type would include the *smallest* number of extra objects possible. Therefore, we use the least upper bound to combine two output types, which is an approximation of a union.

The case for function inputs is similar. Consider that a function parameter, ϕ_x, is used as the input to two functions inside the body of the function defining it. If the two functions work with input types of 1..10 and 5..20 respectively, then the only set of values that is allowed for both is 5..10. In this case, the allowed set of values that works for the functions is the *biggest* set of values that only includes values from both, which is the greatest lower bound, and an approximation of an intersection.

As a type variable represents any type, it is not possible to compute an actual least upper bound or greatest lower bound of two type variables. In the case of two ground types, those should be immediately computable, but when type variables are involved, it is necessary to hold onto the information that a least upper bound or greatest lower bound of two type variables, whatever their value may end up as later, is being computed. Therefore, to the ground types, we add type variables, as well as $\sqcup$ for least upper bound and $\sqcap$ for greatest lower bound, to make *types*.

As the least upper bound can only appear in output types, and the greatest lower

bound can only appear in input types, it is necessary to split the types into two sets of types, defined mutual-recursively. The function types are decomposed appropriately—when a function type is an output, the function's input is an input type, and its output is an output type. However, when a function is used as an input, the case is reversed, and the function's input is an output type, and its output is an input type. Therefore a type is either an output type, which we call a *pos-type*, or an input type, which we call a *neg-type*.

This terminology is reused in §7.10, when it will be seen that a type variable can be marked as negative or positive (or both), meaning that it is used as an input or an output respectively.

Definition 29. *The set* $\mathcal{V} = \{\alpha, \beta, \gamma, \ldots, v, v', v_0, v_1, v_2, \ldots\}$ *is an infinite enumerable set of* type variables*, which is the set of entities that may be used as variables.*

Below, we use characters from the beginning of the Greek alphabet (α, β, γ, etc.) in specific examples, and the v_i form whenever we need a large number of variables.

Definition 30. *The grammar for* pos-types τ^+*,* neg-types τ^-*, and* types τ *is:*

$$
\begin{aligned}
\tau^+ ::= \;& \mathcal{V} \\
\mid \;& \mathbf{t} \\
\mid \;& \tau_0^+ \xrightarrow{v} \tau_1^- \\
\mid \;& (\tau_j^+) \xrightarrow{b} \tau_0^- \\
\mid \;& \uparrow\tau^+ \\
\mid \;& \tau^+ \sqcup \tau^+
\end{aligned}
$$

$$
\begin{aligned}
\tau^- ::= \;& \mathcal{V} \\
\mid \;& \mathbf{t} \\
\mid \;& \tau_0^- \xrightarrow{v} \tau_1^+ \\
\mid \;& (\tau_j^-) \xrightarrow{b} \tau_0^+ \\
\mid \;& \uparrow\tau^- \\
\mid \;& \tau^- \sqcap \tau^-
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= \;& \tau^+ \\
\mid \;& \tau^-
\end{aligned}
$$

The set of pos-types (resp. neg-types, types), is written $\mathbb{T}^+$ *(resp.* $\mathbb{T}^+$*,* $\mathbb{T}$*).*

It is possible to construct types using $\sqcup$ and $\sqcap$ which are syntactically different, but which obviously denote the same type. To remove the ambiguity, we define a set of equivalences between types so that we can define a canonical form for writing a type.

Definition 31. *The equivalences between types are defined in Figure 7.1, where* $\overline{\tau}$ *is written instead of* τ^+*, and* $\underline{\tau}$ *instead of* τ^-*.*

Note that the equivalences defined in Figure 7.1 are purely structural, and are completely independent of the set $\mathcal{V}$ of variables and the set T of atomic types. Both $\sqcup$ and $\sqcap$

$$
\begin{aligned}
\overline{\tau} \sqcup \overline{\tau} &\equiv \overline{\tau} \\
\underline{\tau} \sqcap \underline{\tau} &\equiv \underline{\tau} \\
\overline{\tau}_1 \sqcup \overline{\tau}_2 &\equiv \overline{\tau}_2 \sqcup \overline{\tau}_1 \\
\underline{\tau}_1 \sqcap \underline{\tau}_2 &\equiv \underline{\tau}_2 \sqcap \underline{\tau}_1 \\
\overline{\tau}_1 \sqcup (\overline{\tau}_2 \sqcup \overline{\tau}_3) &\equiv (\overline{\tau}_1 \sqcup \overline{\tau}_2) \sqcup \overline{\tau}_3 \\
\underline{\tau}_1 \sqcap (\underline{\tau}_2 \sqcap \underline{\tau}_3) &\equiv (\underline{\tau}_1 \sqcap \underline{\tau}_2) \sqcap \underline{\tau}_3 \\
\bot \sqcup \overline{\tau} &\equiv \overline{\tau} \\
\bot \sqcap \underline{\tau} &\equiv \bot \\
\top \sqcup \overline{\tau} &\equiv \top \\
\top \sqcap \underline{\tau} &\equiv \underline{\tau} \\
(\underline{\tau}_1 \xrightarrow{v} \overline{\tau}_0) \sqcup (\underline{\tau}'_1 \xrightarrow{v} \overline{\tau}'_0) &\equiv (\underline{\tau}_1 \sqcap \underline{\tau}'_1) \xrightarrow{v} (\overline{\tau}_0 \sqcup \overline{\tau}'_0) \\
(\overline{\tau}_1 \xrightarrow{v} \underline{\tau}_0) \sqcap (\overline{\tau}'_1 \xrightarrow{v} \underline{\tau}'_0) &\equiv (\overline{\tau}_1 \sqcup \overline{\tau}'_1) \xrightarrow{v} (\underline{\tau}_0 \sqcap \underline{\tau}'_0) \\
\left((\underline{\tau}_{j=1..m}) \xrightarrow{b} \overline{\tau}_0\right) \sqcup \left((\underline{\tau}'_{j=1..m}) \xrightarrow{b} \overline{\tau}'_0\right) &\equiv (\underline{\tau}_j \sqcap_j \underline{\tau}'_j) \xrightarrow{b} (\overline{\tau}_0 \sqcup \overline{\tau}'_0) \\
\left((\overline{\tau}_{j=1..m}) \xrightarrow{b} \underline{\tau}_0\right) \sqcap \left((\overline{\tau}'_{j=1..m}) \xrightarrow{b} \underline{\tau}'_0\right) &\equiv (\overline{\tau}_j \sqcup_j \overline{\tau}'_j) \xrightarrow{b} (\underline{\tau}_0 \sqcap \underline{\tau}'_0) \\
(\uparrow\overline{\tau} \sqcup \uparrow\overline{\tau}') &\equiv \uparrow(\overline{\tau} \sqcup \overline{\tau}') \\
(\uparrow\underline{\tau}) \sqcap (\uparrow\underline{\tau}') &\equiv \uparrow(\underline{\tau} \sqcap \underline{\tau}')
\end{aligned}
$$

Figure 7.1: Type equivalences with respect to $\sqcup$ and $\sqcap$.

are idempotent, commutative and associative. For $\sqcup$ (resp. $\sqcap$), $\bot$ is the 0 (resp. 1) element and $\top$ is the 1 (resp. 0) element. Both $\sqcup$ and $\sqcap$ distribute over the function types, with inversion of polarity for the inputs. These equivalences are adapted from those defined by Pottier in Definition 2.2 of [34, p.27]. The identities are either standard or structural, and we have simply adapted the structural ones to the TransLucid types.

By interpreting the equivalencies involving $\top$, $\bot$ and the functional types, as rewrite rules towards the right, we can rewrite all types to be trees where the $\sqcap$ and $\sqcup$ appear on the leaves, in which variables and atomic types appear as arguments. For the types that we build in this paper, if a variable is an argument of a $\sqcap$ or $\sqcup$, then so are the other arguments. Furthermore, since $\sqcup$ and $\sqcap$ are both commutative and associative, we can write $\sqcup\{\alpha, \beta, \gamma\}$ instead of $\alpha \sqcup \beta \sqcup \gamma$ (similarly for $\sqcap$). For a set of variables $V \subset \mathcal{V}$, we can write $\sqcup V$ or $\sqcap V$.

The $\sqcup$ and $\sqcap$ operators are not defined above for arguments of the form $\mathbf{t}$. Suppose we have $\mathbf{t} \sqcup \tau$, for some $\mathbf{t}$. There are three possibilities:

1. τ is a variable. Because of the aforementioned construction restriction, this is not possible and so we do not define it.

2. τ is a function type. An example would be $5 \sqcup (\mathbf{intmp} \xrightarrow{v} \mathbf{ustring})$. We could imagine producing a union type combining the two, but we do not consider this to be the right decision. In all such cases, the result is $\top$, and the corresponding definition for $\sqcap$ is $\bot$.

3. τ is itself of the form $\mathbf{t}$. Then the rules are in Figure 7.2.

Note that the rules in Figure 7.2 can be added to, depending on the intermediate types that we wish to consider important. For example, we could add rules for ranges so that, for example, $5 \sqcup 10$ would be the range 5..10. This flexibility is the second input to the type inference system, alongside the choice of set T of atomic types. It is important to note here that Pottier's system did not include basic types, but, rather, left them as a parameterisation of the system, described in §14.1 and §14.2 of [34, pp.143–5]. Here we have simply made these choices explicit.

$$t_1\langle a_1\rangle \sqcup t_2\langle a_2\rangle \equiv \begin{cases} t_1\langle a_1\rangle & t_1 \equiv t_2 \wedge a_1 \equiv a_2 \\ t_1 & t_1 \equiv t_2 \\ \top & \text{otherwise} \end{cases}$$

$$t_1\langle a_1\rangle \sqcap t_2\langle a_2\rangle \equiv \begin{cases} a_1 & t_1 \equiv t_2 \wedge a_1 \equiv a_2 \\ \bot & \text{otherwise} \end{cases}$$

$$t \sqcup t\langle a\rangle \equiv t$$

$$t \sqcap t\langle a\rangle \equiv t\langle a\rangle$$

$$t \sqcup t' \equiv \begin{cases} t & t \equiv t' \\ \top & \text{otherwise} \end{cases}$$

$$t \sqcap t' \equiv \begin{cases} t & t \equiv t' \\ \bot & \text{otherwise} \end{cases}$$

$$t \sqcup \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle \equiv \begin{cases} \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle & \exists j \text{ s.t. } t_j \equiv t \\ \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t, t_j, \ldots\rangle & \text{otherwise} \end{cases}$$

$$t \sqcap \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle \equiv \begin{cases} t & \exists j \text{ s.t. } t \equiv t_j \\ \mathtt{union}\langle \{t_i\langle a_i\rangle \mid t_i \equiv t\}\rangle & \\ \emptyset & \text{otherwise} \end{cases}$$

$$t\langle a\rangle \sqcup \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle \equiv \begin{cases} \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle & \exists i \text{ s.t. } t_i\langle a_i\rangle \equiv t\langle a\rangle \\ \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle & \exists j \text{ s.t. } t_j \equiv t \\ \mathtt{union}\langle t\langle a\rangle, t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle & \text{otherwise} \end{cases}$$

$$t\langle a\rangle \sqcap \mathtt{union}\langle t_i\langle a_i\rangle, \ldots, t_j, \ldots\rangle \equiv \begin{cases} t\langle a\rangle & \exists i \text{ s.t. } t_i\langle a_i\rangle \equiv t\langle a\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 7.2: $\sqcup$ and $\sqcap$ equivalences for basic types

Definition 32. *The canonical form of a type is defined by pushing the $\sqcap$ and $\sqcup$ terms as deep as possible in the syntax tree defining that type, modulo associativity and commutativity. This corresponds with always replacing the left-hand sides of Figures 7.1 and 7.2 with the corresponding form on the right-hand side.*

We extend the order $\sqsubseteq$ to the types, by taking advantage of the properties of $\sqcup$ and $\sqcap$. By observing that $\alpha \sqcup (\alpha \sqcup \beta)$ is equal to $\alpha \sqcup \beta$, we see that the type α is "already included" in the type $\alpha \sqcup \beta$, and therefore conclude that the type $\alpha \sqcup \beta$ contains a superset

of the information which type α contains. Therefore, we can state that α is a subtype of $\alpha \sqcup \beta$.

Definition 33. *We define the order $\sqsubseteq_*$, an extension over types of the order $\sqsubseteq$.*

- $\forall \tau, \tau',\ \tau \sqsubseteq \tau' \implies \tau \sqsubseteq_* \tau'$
- $\tau_0^+ \sqsubseteq_* \tau_1^+$ *iff* $\tau_0^+ \sqcup \tau_1^+ \equiv \tau_1^+$
- $\tau_0^- \sqsubseteq_* \tau_1^-$ *iff* $\tau_0^- \sqcap \tau_1^- \equiv \tau_1^-$

We simply use $\sqsubseteq$ from now on.

When we later define guarded constraints, a constraint is added to the system only after the lower bound of another type variable is in the type class specified by the guarded constraint. The types are split into sets of type classes, where each basic type defines its own type class and each functional type has a type class. Then we define a type class selector, which will later be used as the guard of a guarded constraint.

Definition 34. *We define the type classes S as* $S = \big\{[t] \mid t \in \mathbf{t}\big\} \cup \big\{[\overset{v}{\rightarrow}], [\uparrow]\big\} \cup \bigcup_m [\overset{b}{\rightarrow}_m]$.

Definition 35. *A* type class selector *is a pair of a variable τ and a type class $s \in S$, written $\tau \in s$. Note that the symbol $\in$ is in bold face, to differentiate the relation for being in a type class, and is defined as follows:*

$$
\begin{aligned}
&\forall t \in \mathbf{t}, t \in [t] \\
&\forall t\langle a\rangle, t\langle a\rangle \in [t] \\
&t \in \mathtt{union}\big\langle t_i\langle a_i\rangle, \dots, t_j, \dots \big\rangle, \textit{if } \exists j \textit{ s.t. } t \equiv t_j \\
&t\langle a\rangle \in \mathtt{union}\big\langle t_i\langle a_i\rangle, \dots, t_j, \dots \big\rangle, \textit{if } \exists i \textit{ s.t. } t\langle a\rangle \equiv t_i\langle a_i\rangle \vee \exists j \textit{ s.t. } t \equiv t_j \\
&\forall \tau, \tau', (\tau \overset{v}{\rightarrow} \tau') \in [\overset{v}{\rightarrow}] \\
&\forall \tau, (\uparrow\tau) \in [\uparrow] \\
&\forall \tau_0, \tau_{m=1..j}, \big((\tau_m) \overset{b}{\rightarrow} \tau_0\big) \in [\overset{b}{\rightarrow}_m]
\end{aligned}
$$

The type inference algorithm supposes a certain structure on types, which is that their height is at most one.

Definition 36. *The function* $\mathrm{height}(\tau)$ *defines the* height *of a type, and is defined as follows:*

$$
\begin{aligned}
\mathrm{height}(v) &= 0 \\
\mathrm{height}(\mathbf{t}) &= 0 \\
\mathrm{height}(\tau \overset{v}{\rightarrow} \tau') &= \max\big(\mathrm{height}(\tau), \mathrm{height}(\tau')\big) + 1 \\
\mathrm{height}(\tau \overset{b}{\rightarrow} \tau') &= \max\big(\mathrm{height}(\tau), \mathrm{height}(\tau')\big) + 1 \\
\mathrm{height}(\uparrow\tau) &= \mathrm{height}(\tau) + 1 \\
\mathrm{height}(\tau \sqcup \tau') &= \max\big(\mathrm{height}(\tau), \mathrm{height}(\tau')\big) + 1 \\
\mathrm{height}(\tau \sqcap \tau') &= \max\big(\mathrm{height}(\tau), \mathrm{height}(\tau')\big) + 1
\end{aligned}
$$

Definition 37. *Let τ be a type, then τ is a* minimal-height type *iff* $\text{height}(\tau) \leq 1$.

7.3 The type scheme

This section describes a *type scheme*, which is made up of a *type context*, specifying the required types of the free parameters to an expression; the *type* of the expression; and a *constraint graph*, which links the two together, through a set of constraints defined below.

Consider the function *plusone* $= \lambda^{\text{v}}\phi_a \rightarrow \underline{\phi_a} + 1$, whose type we wish to determine. Doing so consists of determining two things: 1) the largest possible set of values that can be used as an input, i.e., the parameter ϕ_a, so that the function "works"; and 2) the set of values that could be output from this function. Both will generally be an approximation, but what is important is that for the inputs, we only include values that work, and for the outputs, we include at least the values that will be output. For example, for the absolute value function, it can take as input real numbers, and outputs real numbers. A better type would be that it takes as input real numbers and outputs positive real numbers. Another valid type is that it takes as input integers and outputs the natural numbers.

In order to determine the type of the function *plusone*, we traverse its parse tree, and generate type information from each subexpression. The first relevant subexpression that we reach is $\underline{\phi_a}$, i.e., an access of a function parameter. At this point, we do not know its type, so we allocate a new type variable for it, say α, and record in the type context that its type is α.

As an aside, it is necessary now to state that type variables are allowed to represent an input or an output, but not both. This is in order for simplifications presented later to work.

Returning to the example, the type α of $\underline{\phi_a}$ is its *input* type. We need another type variable for its output type, say β. We then link the two type variables together with a constraint, $(\alpha \leq \beta)$, and the type of expression $\underline{\phi_a}$ is therefore β, along with the type context $\{\phi_a \mapsto \alpha\}$ and the constraint $(\alpha \leq \beta)$.

Here, we will state without justification that the type of the function "+" is $\mathbf{intmp} \xrightarrow{v} \mathbf{intmp} \xrightarrow{v} \mathbf{intmp}$. In other words, it is a call-by-value function that takes as input two objects of *at most* type **intmp**, and outputs only an object of type **intmp**.

The next subexpression that we reach is the application of "+" to $\underline{\phi_a}$. What we require is that the type of the actual input to "+", i.e., the type of $\underline{\phi_a}$, is a smaller type than the required input, which is **intmp**. The way we do that is by generating the constraint $(\mathbf{intmp} \xrightarrow{v} \mathbf{intmp} \xrightarrow{v} \mathbf{intmp} \leq \beta \xrightarrow{v} \gamma)$. This constraint is then decomposed into two constraints, $(\beta \leq \mathbf{intmp})$ and $(\mathbf{intmp} \xrightarrow{v} \mathbf{intmp} \leq \gamma)$. Because we have the constraint $(\alpha \leq \beta)$, the existing constraint on β is transferred to α, so we also get the constraint $(\alpha \leq \mathbf{intmp})$.

This process is then repeated for the application to the value 1. The result is another constraint $(\mathbf{intmp} \xrightarrow{v} \mathbf{intmp} \leq 1 \xrightarrow{v} \delta)$, which is decomposed into the two constraints $(\mathbf{intmp} \leq \delta)$ and $(1 \leq \mathbf{intmp})$. The latter is trivially true. The type of the result of the final application is therefore δ.

This leaves us with the function abstraction. The type of its body is δ, along with all of the previously mentioned constraints and type context. The type of ϕ_a is pulled out of the type context, which is α, and therefore the type of the abstraction is $\alpha \xrightarrow{v} \delta$, along with all of the above constraints, and an empty type context, since the parameter is removed.

To recap, the type is $\alpha \xrightarrow{v} \delta$, along with an empty type context and the constraints below, which together make up the type scheme.

$$\begin{aligned} \alpha &\leq \textbf{intmp} \\ \alpha &\leq \beta \\ \beta &\leq \textbf{intmp} \\ \textbf{intmp} \xrightarrow{v} \textbf{intmp} &\leq \gamma \\ \textbf{intmp} &\leq \delta \\ 1 &\leq \textbf{intmp} \end{aligned}$$

At this point we can also give an intuition into garbage collection, presented in §7.10. It can be seen quite trivially that the only type variables that are *reachable* in the above constraint set are α and δ, therefore we can remove the others. The resulting type is simply $\alpha \xrightarrow{v} \delta$ with the constraints:

$$\begin{aligned} \alpha &\leq \textbf{intmp} \\ \textbf{intmp} &\leq \delta \end{aligned}$$

Lastly, this type can be displayed to the user in a much more friendly way. As α is an input, and it only has one upper bound, it can be replaced with that upper bound. Likewise for δ and its lower bound, since it is an output. Therefore, we can display to the user that the type of *plusone* is $\textbf{intmp} \xrightarrow{v} \textbf{intmp}$.

7.3.1 Type contexts

A type context represents the types that are required by the usage of an identifier or a function parameter, alternatively, the type of any inputs as required by their usage in the expression. In addition, the TransLucid context information is stored in a type context.

There are four parts to a type context:

1. a mapping from identifiers to the type required by their usage;
2. a mapping from ϕ dimensions (function parameters) to the type required by their usage;
3. a mapping from constants, which are known dimensions, to both their lower and upper bounds, as set by context changes, and as required by their respective usage; and
4. a mapping from ϕ dimensions, whose values are used as dimensions, to a triple of a type variable representing their actual type, the lower and upper bounds, which are the same as for the previous item.

Definition 38. *Let X be a set of identifiers and D_ϕ be a set of hidden dimensions, then a* type context *is a 4-tuple of functions $K = (K_x, K_\lambda, K_c, K_{\underline{\phi}})$, defined as follows:*

$$\begin{aligned} K_x &: X \rightarrowtail \mathbb{T}^- \\ K_\lambda &: D_\phi \rightarrowtail \mathbb{T}^- \\ K_d &: D \rightarrowtail (\mathbb{T}^+, \mathbb{T}^-) \\ K_{\underline{\phi}} &: D_\phi \rightarrowtail (\mathbb{T}^+, \mathbb{T}^+, \mathbb{T}^-) \end{aligned}$$

We define a join operator to join together type contexts. This is used in the type inference algorithm when an expression has multiple subexpressions, and the resulting type contexts must be joined to make one type context. When the domains of the two type contexts do not overlap, then this is simply function perturbation. When they do overlap, any pos-types must be combined with $\sqcup$, and any neg-types must be combined with $\sqcap$.

Definition 39. *Let K and K' be type contexts, then the generalised meet, $K \sqcap K'$, of the two type contexts is defined as below:*

$$\begin{aligned} \big(K \sqcap K'\big)(v) &= \big(K_x \sqcap K'_x, K_\lambda \sqcap K'_\lambda, K_c \sqcap K'_c, K_{\underline{\phi}} \sqcap K'_{\underline{\phi}}\big) \\ \big(K_{i\in\{x,\lambda,d,\underline{\phi}\}} \sqcap K'_i\big)(v) &= \begin{cases} K_i(v) & v \in \text{dom } K_i \wedge v \notin \text{dom } K'_i \\ K'_i(v) & v \notin \text{dom } K_i \wedge v \in \text{dom } K'_i \end{cases} \\ \big(K_x \sqcap K'_x\big)(x) &= K_x(x) \sqcap K'_x(x) \\ \big(K_\lambda \sqcap K'_\lambda\big)(\phi_x) &= K_x(\phi_x) \sqcap K'_x(\phi_x) \\ \big(K_d \sqcap K'_d\big)(d) &= (\tau_0 \sqcup \tau'_0, \tau_1 \sqcap \tau'_1), \qquad K_d(v) = (\tau_0, \tau_1) \wedge K'_d(v) = (\tau'_0, \tau'_1) \\ \big(K_{\underline{\phi}} \sqcap K'_{\underline{\phi}}\big)(\phi_x) &= (\tau_0 \sqcup \tau'_0, \tau_1 \sqcup \tau'_1, \tau_2 \sqcap \tau'_2), \\ & \qquad K_{\underline{\phi}}(\phi_x) = (\tau_0, \tau_1, \tau_2) \wedge K'_{\underline{\phi}}(\phi_x) = (\tau'_0, \tau'_1, \tau'_2) \end{aligned}$$

Here our type context is adapted from the one presented by Pottier in Definition 4.1 of [34, p.42]. Pottier's type context has two components, which correspond to our K_x and K_λ. We have adapted ours to have four components, as appropriate for TransLucid.

7.3.2 Constraints

A constraint is a pair $\tau \leq \tau'$ of two types, which specifies that the left-hand type must be a subset of or equal to the right-hand type. A constraint expresses the desire that $\tau \sqsubseteq \tau'$ be true. If a constraint appears in the system for which this is not true, then the relevant expression has a type error; this situation occurs when the function subc (Definition 42) is not defined for its input.

Definition 40. *A* constraint *is a pair of a pos-type $\tau^+ \in \mathbb{T}^+$ and a neg-type $\tau^- \in \mathbb{T}^-$, written $\tau^+ \leq \tau^-$.*

As will be described shortly, constraints are stored in a constraint graph, which is maintained as a closed graph at all times. A consequence of keeping the graph closed is that

$$
\begin{aligned}
\mathrm{subc}(\sqcup S \leq \tau') &= \bigcup_{\tau \in S} \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(\tau \leq \sqcap S) &= \bigcup_{\tau' \in S} \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(c) &= \{c\} \textit{ if } c \textit{ is elementary} \\
\mathrm{subc}\big(t\langle a\rangle \leq t\langle a\rangle\big) &= \emptyset \\
\mathrm{subc}\big(t\langle a\rangle \leq t\big) &= \emptyset \\
\mathrm{subc}(t \leq t) &= \emptyset \\
\mathrm{subc}\big(t \leq \mathtt{union}\langle t, \ldots\rangle\big) &= \emptyset \\
\mathrm{subc}\big(t\langle a\rangle \leq \mathtt{union}\langle t, \ldots\rangle\big) &= \emptyset \\
\mathrm{subc}\big(t\langle a\rangle \leq \mathtt{union}\langle t\langle a\rangle, \ldots\rangle\big) &= \emptyset \\
\mathrm{subc}(\bot \leq \tau) &= \emptyset \\
\mathrm{subc}(\tau \leq \top) &= \emptyset \\
\mathrm{subc}\big((\tau_0 \overset{v}{\to} \tau_1) \leq (\tau_0' \overset{v}{\to} \tau_1')\big) &= \mathrm{subc}(\tau_0' \leq \tau_0) \cup \mathrm{subc}(\tau_1 \leq \tau_1') \\
\mathrm{subc}\big(((\tau_{j=1..n}) \overset{b}{\to} \tau_0) \leq ((\tau_j') \overset{b}{\to} \tau_0')\big) &= \mathrm{subc}(\tau_0 \leq \tau_0') \cup \bigcup_j \mathrm{subc}(\tau_j' \leq \tau_j) \\
\mathrm{subc}(\uparrow\tau \leq \uparrow\tau') &= \mathrm{subc}(\tau \leq \tau')
\end{aligned}
$$

Figure 7.3: The function subc.

if any constraint implies any other constraint, then those other constraints are also stored in the graph. For that reason, it is necessary to define how constraints are decomposed into what we call *elementary constraints*. For example, the constraint $\tau_0 \overset{v}{\to} \tau_1 \leq \tau_0' \overset{v}{\to} \tau_1'$ implies the set of elementary constraints $\{\tau_0' \leq \tau_0, \tau_1 \leq \tau_1'\}$.

Definition 41. *A constraint $\tau \leq \tau'$ is* elementary *iff the following conditions are met:*

- *neither τ nor τ' are of the form $\sqcap V$ or $\sqcup V$;*
- *at least one of $\{\tau, \tau'\}$ is a type variable.*

Note that there is a special case for $\sqcap$ and $\sqcup$; they are special objects used to encode greatest-lower-bounds and least-upper-bounds of type variables. So the constraint $\sqcup\{\alpha, \beta\} \leq \gamma$ is not elementary—it can be broken into the set of elementary constraints $\{\alpha \leq \gamma, \beta \leq \gamma\}$. Both Definitions 40 and 41 correspond exactly to those presented in Definitions 3.1 and 3.4 of [34, pp.36–7].

Definition 42. *The rules given in Figure 7.3, whose order is important, define the function* subc*, which decomposes any solvable constraint into a set of elementary constraints. As* subc *is used in the constraint closure algorithm, if the input is not in its domain, then that indicates a type error.*

The function subc is presented by Pottier as Proposition 3.1 in [34, p.37]. We have simply adapted the definition to the types of TransLucid.

7.3.3 Guarded constraints

Here we extend Pottier's presentation of conditional constraints presented in [35] to TransLucid, and rename it to *guarded constraints*. Guarded constraints are an extension to type inference with subtyping, which allow a constraint to be dependent on some other type.

More specifically, guarded constraints allow a more refined type for conditional expressions. One way to handle conditional expressions would be to require that the condition have a type smaller than Boolean, and for the type of the conditional to be the common lower bound of the branches. With guarded constraints, if we know that the condition is always true, then the conditional only has the type of the true branch, the case is similar if the condition is always false. If the best that we can infer is Boolean for the type of the condition, then we would still use the common lower bound as before.

It is important to notice here that the principal type of an object being itself is necessary for guarded constraints to be useful. The implication is that an expression can have type `true` or `false`, rather than **bool**. If the type of the condition in a conditional expression could only ever be **bool**, then guarded constraints would be a complete waste of time.

Definition 43. *A* guarded constraint *is a pair of a type class selector* ($\tau \in s$)*, and a constraint* ($\tau_1 \leq \tau_2$)*, written* $\begin{pmatrix} \tau \in s \\ \tau_1 \leq \tau_2 \end{pmatrix}$.

The guarded constraint encodes that if τ is in the type class s, then the constraint $\tau_1 \leq \tau_2$ is also implied.

The reason for using type classes, rather than a complete type, is that a complete type may have type variables in it. As a result, it is impossible to decide whether one type is smaller than another. The solution is to use the type class instead, as it provides an obvious means to determine the condition of the constraint.

As an example of the use of the guarded constraint, consider the following expression:

$$\lambda^{\mathrm{v}} a \rightarrow \texttt{if}\ a\ \texttt{then}\ 42\ \texttt{else}\ \text{“hello world”}\ \texttt{fi}$$

Then the type of the expression would be $v_0 \xrightarrow{v} v_1$, with the following constraints:

$$\begin{aligned} &42 \leq v_2 \\ &\text{“hello world”} \leq v_3 \\ &v_0 \leq \textbf{bool} \end{aligned}$$

along with the two guarded constraints:

$$\begin{pmatrix} v_0 \in \texttt{true} \\ v_2 \leq v_1 \end{pmatrix}$$
$$\begin{pmatrix} v_0 \in \texttt{false} \\ v_3 \leq v_1 \end{pmatrix}$$

Here, v_0 is for the type of the input parameter a, v_1 is for the type of the whole conditional expression, v_2 is for the type 42, and v_3 is for the type “hello world”. The effect is that if

the input type is known to be `true`, then the type of the function will be 42; and if the input type is known to be `false`, then the type will be "hello world". If the input type is not known that precisely, i.e., it is **bool**, then both guarded constraints will be valid, and the type will be $42 \sqcup$ "hello world".

7.3.4 Constraint graph

Definition 44. *A* constraint graph $G = (V_G, \leq_G)$*, of domain* $V \subseteq \mathcal{V}$*, is a vertex- and edge-labelled directed graph. Each vertex label is a type interval, which consists of a lower bound, which is a pos-type, written* $G^{\downarrow}(v)$*, and an upper bound, which is a neg-type, written* $G^{\uparrow}(v)$*, whose free variables are in* V*. The edges,* $\leq_G$*, are labelled with type class selectors, labelled edges are written as* $v \overset{s \in \alpha}{\leq_G} v'$*, and unlabelled edges are written as* $v \leq_G v'$*.*

Here we have defined our constraint graph slightly differently to Pottier. His paper on conditional constraints does not even define a graph, rather, it talks about a set of constraints. The criteria we have defined for closure below is still consistent with Pottier's, whether a graph or a set of constraints is used. Our graph differs cosmetically in that we have labelled both vertices and edges, but nevertheless, the two are equivalent.

The following two definitions are about the closure of a graph. If a constraint graph is closed, then it has a solution, otherwise it does not. If the constraint graph generated by the type inference rules for a program is closed, then the program is well typed. The first definition is for the closure of a graph, the second an algorithm for computing the closure.

Definition 45. *Let* G *be a constraint graph of domain* V*. Then* G *is closed iff*

$$\forall \alpha, \beta, \gamma \in V, (\alpha \leq_G \beta) \wedge (\beta \leq_G \gamma) \implies (\alpha \leq_G \gamma)$$

$$\forall \alpha, \beta \in V \text{ s.t. } (\alpha \leq_G \beta), G^{\downarrow}(\beta) \sqsubseteq^* G^{\downarrow}(\alpha) \wedge G^{\uparrow}(\alpha) \sqsubseteq^* G^{\uparrow}(\beta)$$

$$\forall \alpha \in V, \mathrm{subc}(G^{\downarrow}(\alpha) \leq G^{\uparrow}(\alpha)) \text{ is defined}$$

$$(\alpha \leq_G \beta) \wedge \left(\gamma \overset{\beta \in s}{\leq_G} \delta\right) \implies \left(\gamma \overset{\alpha \in s}{\leq_G} \delta\right)$$

$$\left(\alpha \overset{\gamma \in s}{\leq_G} \beta\right) \wedge G^{\downarrow}(\gamma) \in s \implies (\alpha \leq_G \beta)$$

The first three points of this definition correspond to Pottier's definition of closure presented in Definition 3.10 of [34, p.40], the addition of the last two points corresponds to Definition 14 of [35, p. 4].

Definition 46. *Let* $G = (V_G, \leq_G)$ *be a constraint graph,* C *be a set of constraints of the form* $\alpha \leq \beta$*,* $\alpha \leq \tau$ *or* $\tau \leq \beta$*, where* α *and* β *are type variables, and* τ *is a minimal-height type. The closure algorithm described below maintains the transitive closure of* G*, after adding all of the constraints in* C*. If the set of constraints in* G *and* C *has no solution, then the function* subc *will fail because at least one input from the closure computation will not be in its domain.*

The algorithm produces a stream of pairs of constraint graphs and sets of constraints (G_i, C_i)*, where* $G_0 = G$*,* $C_0 = C$*, and cases for* $i \geq 1$ *are described below. The algorithm terminates with the result being* G_i *for the first* C_i *that is empty.*

Pick some constraint c_i in C_i, then $C_i' = C_i \setminus \{c_i\}$. The set $C_i^?$ is a set of constraints generated by conditional constraints at each step, and B_i' is a set of variables that each received a new lower bound at each step.

There are three cases, depending on the structure of the constraint c_i:

- *c_i is of the form $\alpha \leq \beta$. If $\alpha \leq_{G_i} \beta$, then $(G_{i+1}, C_{i+1}) = (G_i, C_i')$, as there is nothing further to do. Otherwise, G_{i+1} and C_{i+1} are defined as follows:*

$$
\begin{aligned}
&B_i' = \{\beta' \mid \beta \leq_G \beta'\} \\
&\leq_{G_{i+1}} = \leq_{G_i} \cup \left\{\alpha' \leq \beta' \mid \alpha' \leq_{G_i} \alpha \wedge \beta' \in B_i'\right\} \cup \left\{ \left(\gamma \overset{\alpha \in s}{\leq} \delta\right) \mid \left(\gamma \overset{\beta \in s}{\leq_{G_i}} \delta\right) \right\} \\
&G_{i+1}^{\downarrow}(\beta') = \begin{cases} G_i^{\downarrow}(\beta') \sqcup G_i^{\downarrow}(\alpha), & \beta' \in B_i' \\ G_{i+1}^{\downarrow}(\beta') = G_i^{\downarrow}(\beta'), & \textit{otherwise} \end{cases} \\
&G_{i+1}^{\uparrow}(\alpha') = \begin{cases} G_i^{\uparrow}(\alpha') \sqcap G_i^{\uparrow}(\beta), & \alpha' \in \{v \mid v \leq_{G_i} \alpha\} \\ G_i^{\uparrow}(\alpha'), & \textit{otherwise} \end{cases} \\
&C_i^? = \left\{ \gamma \leq \delta \;\middle|\; \left(\gamma \overset{\beta' \in s}{\leq_{G_i}} \delta\right) \wedge \beta' \in B_i' \wedge G_{i+1}^{\downarrow}(\beta') \in s \right\} \\
&C_{i+1} = C_i' \cup C_i^? \cup \mathrm{subc}(G_i^{\downarrow}(\alpha) \leq G_i^{\uparrow}(\beta))
\end{aligned}
$$

- *c is of the form $\alpha \leq \tau$. If $\tau \sqsubseteq G_i^{\uparrow}(\alpha)$, then $(G_{i+1}, C_{i+1}) = (G_i, C_i')$. Otherwise G_{i+1} and C_{i+1} are defined as follows:*

$$
\begin{aligned}
&\leq_{G_{i+1}} = \leq_{G_i} \\
&G_{i+1}^{\downarrow} = G_i^{\downarrow} \\
&G_{i+1}^{\uparrow}(\alpha') = \begin{cases} G_i^{\uparrow}(\alpha') \sqcap \tau, & \alpha' \leq_{G_i} \alpha \\ G_i^{\uparrow}(\alpha'), & \textit{otherwise} \end{cases} \\
&C_{i+1} = C_i' \cup \mathrm{subc}(G_i^{\downarrow}(\alpha) \leq \tau)
\end{aligned}
$$

- *c is of the form $\tau \leq \beta$. If $\tau \sqsubseteq G_i^{\downarrow}(\beta)$, then $(G_{i+1}, C_{i+1}) = (G_i, C_i')$. Otherwise, G_{i+1} and C_{i+1} are defined as follows:*

$$
\begin{aligned}
&B_i' = \{\beta \mid \beta \leq_{G_i} \beta'\} \\
&\leq_{G_{i+1}} = \leq_{G_i} \\
&G_{i+1}^{\downarrow}(\beta') = \begin{cases} G_i^{\downarrow}(\beta') \sqcup \tau, & \beta' \in B_i' \\ G_i^{\downarrow}(\beta'), & \textit{otherwise} \end{cases} \\
&G_{i+1}^{\uparrow} = G_i^{\uparrow} \\
&C_i^? = \left\{ \gamma \leq \delta \;\middle|\; \left(\gamma \overset{\beta' \in s}{\leq_{G_i}} \delta\right) \wedge \beta' \in B_i' \wedge G_{i+1}^{\downarrow}(\beta') \in s \right\} \\
&C_{i+1} = C_i' \cup C_i^? \cup \mathrm{subc}(\tau \leq G_i^{\uparrow}(v_1))
\end{aligned}
$$

In understanding the above algorithm, it is useful to remember that the less-than relation in a constraint graph is reflexive, meaning that for all constraint graphs $G = (V, \leq_G)$, $\quad \forall \alpha \in V, \quad \alpha \leq_G \alpha$.

This algorithm is presented in Definition 7.1 of [34, p.74], without conditional constraints. Here we have presented the algorithm in an iterative manner, and taken into account guarded constraints.

Finally, we present two operators over constraint graphs, the union and the closure.

Definition 47. *The union of two constraint graphs* $G_0 = (V_{G_0}, \leq_{G_0})$ *and* $G_1 = (V_{G_1}, \leq_{G_1})$ *can be taken when* $V_{G_0} \cap V_{G_1} = \emptyset$*, and is defined as follows:*

$$G_0 \cup G_1 = (V_{G_0} \cup V_{G_1}, \leq_{G_0} \cup \leq_{G_1})$$

Definition 48. *The closure of a constraint graph is written using the* $+$ *operator, where* $G + c$ *denotes adding the constraint set* $\{c\}$ *to the graph* G *using the incremental closure computation in Definition 46.*

Notation

Here we introduce the notation we use to write down a constraint graph in some of the examples presented later in this chapter. Here is an example constraint graph:

$$\begin{array}{rllll} & & v_1 & \leq & v_2 \\ \textbf{intmp} & v_1 & \leq & v_2 & \\ & v_4 & \leq & v_3 & \leq \quad v_5, v_6 \quad (v_1 \xrightarrow{v} v_2) \\ & & v_4 & \leq & v_3, v_5, v_6 \quad (v_1 \xrightarrow{v} v_2) \\ & v_4 & \leq & v_5 & \\ & v_4 & \leq & v_6 & \end{array}$$

Here we have six variables, each with some constraints. Each variable appears once in the center column, with its lower bounds to the left and its upper bounds to the right. We separate visually the bounds which are not simply variables. Here the variable v_1 only has the constraint $v_1 \leq v_2$, whilst v_2 also has a lower bound **intmp**. The variable v_3 has v_4 less than it, and v_5 and v_6 greater than, along with the upper bound $(v_1 \xrightarrow{v} v_2)$. The variable v_4 must have the same upper bound as v_3 for the graph to be closed, along with the same variables that are greater than it. Then v_5 and v_6 are also written down with v_4 as less than both.

7.3.5 Building a type scheme

The type of an expression is not a simple type, but a *type scheme*, which is a tuple of a type context, a type and a constraint graph.

Definition 49. *A* type scheme *is a triple* $\sigma = (K_\sigma, \tau_\sigma^+, G_\sigma)$*, normally written as*

$$K_\sigma \Rightarrow \tau_\sigma^+|_{G_\sigma}$$

and read as the pos-type τ_σ^+ such that the constraints G_σ hold given the inputs required by the type context K_σ.

Note that a type scheme is implicitly universally quantified over all type variables—there is no explicit for-all quantifier in our type grammar. A consequence of this is that the so called let-polymorphism is handled by renaming all of the type variables in the type scheme for a particular identifier.

Definition 50. *The function $(\sigma', V') = \text{rename}(\sigma, V)$ takes as input a type scheme and a set of variables (σ, V), and returns a type scheme and a set of variables (σ', V') such that $V' \subseteq V \wedge FV(\sigma) \cap FV(\sigma') = \emptyset \wedge FV(\sigma') \cap V' = \emptyset$.*

7.4 Constraint generation

The process of type inference consists of traversing an expression tree and generating constraints, whilst simultaneously keeping the constraint graph of all those constraints closed. If this succeeds, then the result is the most general type of the expression whose type is being inferred. The rules presented below can be used in a syntax directed way to infer the type of an expression.

Definition 51. *A type environment is a mapping ζ^τ, from TransLucid identifiers X to type schemes ζ^τ:*

$$\zeta^\tau : X \rightarrowtail \sigma$$

Definition 52. *Given an expression E, an interpretation ι, and a type environment ζ^τ, where $FV(E) \subseteq \text{dom}\ \zeta^\tau$, such that $\sigma_i = \zeta^\tau(x_i)$ is the most general type of the expression defining variable x_i, then the most general type of E is determined by evaluating the following:*

$$\iota, \zeta^\tau \overset{\tau}{\vdash} E : K \Rightarrow \tau|_G$$

meaning that E has the type τ such that the constraints in G are satisfied, under the type context K.

When composing constraints, the rules make use of two distinct operations. One is the union of two constraint graphs ($\cup$), and the other is adding to the closure of a constraint graph ($+$). When the union of two constraint graphs can be taken, it is because those constraint graphs have come from two separate syntax trees, which by definition share no type variables, so the union is already a closed graph. When we use the $+$ symbol with a constraint graph G and a constraint c, we are running the closure algorithm with the input $(G, \{c\})$. Even in constructing a graph with a single constraint, it is necessary to write $\emptyset + \alpha \leq \beta$, because the closure computation must construct a graph G, with an edge $\alpha \leq_G \beta$.

7.5 The TransLucid context

The computation of the general type of an expression E, as defined in Definition 52, assumes a type context K. According to Definition 38 (§5.1), K is a four-tuple of the form

$(K_x, K_\phi, K_d, K_{\underline{\phi}})$, where $K_d : D \rightarrowtail (\mathbb{T}^+, \mathbb{T}^-)$ maps a constant c—used as dimension—to the range of ordinates that c can take on as dimension.

However, the rank of an expression E, i.e., the set of dimensions relevant to the evaluation of E, cannot in general be computed statically. When the general type of an expression is of the form $\#.E_0$, the problem is that, in theory, E_0 can evaluate to any value to be used as dimension. A similar case applies to E'_0 in an expression of the form $E_0 \,@\, [E'_0 \leftarrow E'_1]$. As a result, the K_d component of K cannot in general be properly determined.

To resolve this problem correctly would require having an abstract interpreter built in to the type inference algorithm to determine the possible values of E_0 and E'_0 in the above expressions. We do not exclude this possibility in future work, but for now, we will only type expressions in which such expressions lead to a constant or an input parameter as dimension.

The rules therefore use a function, "unique_bound_pos", in order to determine that the type of an expression is a constant type. The rules use "unique_bound_pos" where a type variable is *positive*; we will define the *polarity*, both positive and negative, of a variable in §7.10.

Definition 53. *Given a constraint graph G, the* unique bound *of a positive type variable v is given by the function* $\text{unique\_bound\_pos}(v, G)$

$$\text{unique\_bound\_pos}(v, G) = \begin{cases} v', & G^{\downarrow}(v) \equiv \bot \text{ and } v' \text{ is the unique } v' \text{ s.t. } v' \leq_G v \\ \tau, & G^{\downarrow}(v) \equiv \tau \text{ and there is no } v' \text{ s.t. } v' \leq_G v \end{cases}$$

When unique_bound_pos *is not defined, the positive type variable v does not have a unique bound.*

To determine which dimension is being used in a context change or query, we consider three specific cases: 1) a dimension in a context change or context query is an atomic value, which corresponds to the type of E_0 or E'_0 above having a unique bound, 2) a dimension in a context change or context query is passed as a parameter to a function, and 3) a dimension is a local dimension declared in a `wheredim` clause.

As well as inferring the actual dimensions used in a program, it is necessary to determine a type for the ordinate of each dimension. Type information for the ordinate of a dimension d can come from two places: 1) the way in which an ordinate is used after a context lookup, $\#.d$, which provides an upper bound for the ordinate of d, and 2) a context change, $E_0 \,@\, [d \leftarrow E'_1]$, which provides a lower bound for the ordinate of dimension d. For a TransLucid context to be well-typed, it is necessary that the lower bounds provided by context changes be consistent with the upper bounds required by the usage of an ordinate.

Inferring the type of an ordinate of a dimension in the TransLucid context is made difficult by the fact that dimensions are dynamically bound, which means that the ordinate of a dimension could come from any context change in a program.

To see the solution to determining the type of each ordinate, we can look at the scope in which an ordinate can be used, so that we can see what the appropriate lower and upper bounds for that ordinate are.

First we will consider the type of a function parameter, since an ordinate is like a function parameter, but used in a slightly different way. Inside a function, an input parameter will potentially be used in a number of ways, each providing an upper bound on the type allowed for the parameter. The final type for the input parameter is the greatest lower bound of all those upper bounds. The scope for a function parameter is inside the function, which is why we take the greatest lower bound of all the upper bounds of the parameter as used inside the function, which is anywhere that it can be used in its scope.

We do the same thing for ordinates of dimensions. The difference is that the scope of a dimension is the entire program, so we need to take the greatest lower bound τ_u, of all the upper bounds of a dimension as required by the entire program. The lower bound of the ordinate of a dimension is set in a similar way to the lower bound of a function parameter, except that rather than being set by a function application, the ordinate of a dimension is set by every context change in the whole program that modifies that dimension. So for a dimension d, we take the least upper bound, τ_l, of the lower bound of all the expressions that set the ordinate of d. Finally, we add the set of constraints "subc($\tau_l \leq \tau_u$)" to our constraint graph.

7.5.1 Atomic values

Firstly, we will consider an atomic value c being used as a dimension. There are two places in which c can be used, a context lookup and a context change, each providing, respectively, an upper bound and a lower bound for the ordinate of that dimension.

Let us consider a program in which the following three expressions appear:

$$\begin{aligned} &\texttt{\#}.0 \times 5 \\ &E_0 \texttt{@} [0 \leftarrow 1] \\ &E_1 \texttt{@} [0 \leftarrow \texttt{\#}.0 + 1] \end{aligned}$$

The two `#` subexpressions are used in such a way that the ordinate of dimension 0 has an upper bound of **intmp**—both the addition and multiplication functions require their inputs to be a smaller type than **intmp**. As a result, we have an upper bound for the ordinate of dimension 0: **intmp** $\sqcap$ **intmp**, which is **intmp**. For the `@` expressions, the ordinate of dimension 0 is initialised with an expression whose type is 1 and **intmp** respectively. Therefore, the lower bound for the ordinate of dimension 0 is $1 \sqcup$ **intmp**, which is **intmp**.

For a program in which the above three expressions are the only occurences of dimension 0 being accessed, we can therefore infer that both the lower and upper bounds for the ordinate of dimension 0 are **intmp**.

All of the above is trivial, as it is the same as the way in which function parameters are handled. The difficulty is in determining, in general, what the dimension is—since it can be computed dynamically, it is undecidable by the type checker—we require here that the dimension be computable in the rules. For the above example, consider the expression

‘#.0’. The type of expression ‘0’ is a type variable α, with the constraint graph $(0 \leq \alpha)$. As the type variable α has the lower bound 0, with no type variables smaller than it, we say that the *unique bound* of α is 0. It is because the unique bound is defined that we can say that we are looking up dimension 0 in the context. The case for a context change expression is identical.

7.5.2 Dimensions as parameters

Second, we will consider the case when a dimension is passed as a function parameter. As described above, a dimension must be an atomic value. However, when the body of a function is being analysed, we do not know what its actual parameters are, so we cannot assign the use of a dimension to a specific atomic value. As a result, we must also keep track of dimensions that are passed as function parameters, and then assign them to the actual dimension once it is known.

Consider a function $\lambda\phi_d \rightarrow E_0$, whose body E_0 contains the expression ‘#.$\underline{\phi_d}$’, i.e. an expression looking up in the context the dimension passed as parameter ϕ_d. When checking the type of this expression, we do not know which dimension is being accessed, because it will be passed as a parameter when the function is applied. This expression violates the previous condition that the type of a dimension be known, but we cannot exclude it, since one of the core concepts of TransLucid is that dimensions can be passed around.

The solution then is to look for expressions that are consistent with looking up a dimension passed as a parameter. A parameter is the expression $\underline{\phi_x}$, and a context lookup is #.E, so we need a rule for #.$\underline{\phi_x}$. The rule is similar to the one for context lookup with an atomic value as dimension. The main difference is that we must also keep around the type variable that will eventually have as its lower bound the type of the actual dimension.

Then, whenever the function whose parameter is used as a dimension is applied, the lower bound of that parameter will be set by the application, and as long as that lower bound is an atomic value, then this case degenerates into the case when a dimension is an atomic value.

7.5.3 The `wheredim` clause

Third, we consider a dimension declared in a `wheredim` clause. A dimension declared in a `wheredim` clause is treated almost the same as in the first case, for an atomic value; in fact, the value used for a local dimension *is* an atomic value, it is just a special hidden value that can only be accessed through a local dimension identifier. The solution then is quite similar to the first case. The differences are that a `wheredim` clause can initialise its dimension, which acts like a context change, and since the scope of a local dimension is inside its defining `wheredim` clause, that `wheredim` clause combines the lower and upper bounds for its dimensions in the same way as is done for atomic values at the whole program scope.

7.6 Abstract syntax

The abstract syntax that we use for type inference is the syntax used by the operational semantics, presented in Chapter 5, with some restrictions. These restrictions are as follows:

1. As for Chapter 6, all uses of the context must be an immediate lookup of a dimension, or the lookup of a dimension passed as parameter, i.e., that `#` only appear in expressions of the form `#`.E, or `#`.$\underline{\phi_x}$.

2. All context changes must have as their right-hand side an explicit tuple builder expression, i.e., that `@` only appear in expressions of the form E `@` $[E \leftarrow E, \ldots]$.

The actual structure of `wherevar` clauses manipulated by the implementation is more complicated than what is presented here. The implementation carries out substantial syntactic-level manipulation to reorganise systems of equations into strongly connected components. This is done so that groups of mutually recursive variables have their most general types correctly inferred. Nevertheless, the rules presented below apply regardless of the structure of `wherevar` clauses, and are what is relevant to this chapter. But it is noted that for the best types to be inferred, this restructuring should be carried out.

7.7 The rules

The rules presented below are in the form of structural operational semantics. They provide a syntax-directed mechanism by which to infer the type of an expression. The rules are of the form:

$$\frac{p}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} E : K \Rightarrow \tau|_G}$$

The symbol p above the line stands for all of the predicates and a set of variable bindings. When the set of predicates in p are true, and the expression whose type is being inferred matches the form of E, then that rule is applicable. When the rule is applicable, we can say that E has the type $K \Rightarrow \tau|_G$. By convention, for each rule, we explicitly define K, τ, and G. To avoid the rules becoming too messy, we do not write down an implicit V component, which is used to allocate type variables. Whenever any of α, β, γ, δ or ϵ appears free in a rule, a new type variable is being taken from the set V. When subexpressions E_0, E_1, ... are being typed, the set V needs to be split into V_0, V_1, ... in such a way that the V_i are always infinite.

$$\frac{\begin{array}{l} G = \emptyset + \left(\iota({}^{m}c) \leq \alpha\right) \\ \tau = \alpha \\ K = (\emptyset, \emptyset, \emptyset, \emptyset) \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad {}^{m}c \quad : K \Rightarrow \tau|_G} \tag{7.1}$$

$$\frac{\begin{array}{l} G = \emptyset + (\alpha \leq \beta) \\ \tau = \beta \\ K = \big(\emptyset, \{\phi_x \mapsto \alpha\}, \emptyset, \emptyset\big) \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \underline{\phi_x} \quad : K \Rightarrow \tau|_G} \tag{7.2}$$

$$\frac{\begin{array}{l} G = \emptyset + (\alpha \leq \beta) + (\delta \leq \epsilon) \\ \tau = \epsilon \\ K = \big(\emptyset, \{\phi_x \mapsto \alpha\}, \emptyset, \{\phi_x \mapsto (\beta, \gamma, \delta)\}\big) \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \#.\underline{\phi_x} \quad : K \Rightarrow \tau|_G} \tag{7.3}$$

$$\frac{\begin{array}{l} \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \text{unique\_bound\_pos}(\tau_0, G_0) : c \\ G = G_0 + (\beta \leq \alpha) \\ \tau = \alpha \\ K = K_0 \sqcap \big(\emptyset, \emptyset, \{c \mapsto (\alpha, \beta)\}, \emptyset\big) \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \#.E_0 \quad : K \Rightarrow \tau|_G} \tag{7.4}$$

$$\frac{\begin{array}{l} \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_i : K_i \Rightarrow \tau_i|_{G_i} \\ K_{0_\lambda}(\phi_{x_j}) : \tau_j \\ G = \big(G_0 \cup \bigcup_i G_i\big) + \big((\tau_j) \to^{\mathrm{b}} \tau_0 \leq \alpha\big) \\ \tau = \alpha \\ K = \big(K_{0_x}, K_{0_\lambda} \lhd \bigcup_j \{\phi_{x_j}\}, K_{0_c}, K_{0_{\underline{\phi}}}\big) \sqcap \prod_i K_i \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \lambda^{\mathrm{b}}_{\circ}\, \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0 \quad : K \Rightarrow \tau|_G} \tag{7.5}$$

$$\frac{\begin{array}{l} \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_i : K_i \Rightarrow \tau_i|_{G_i} \\ G = \big(G_0 \cup \bigcup_i G_i\big) + \big(\tau_0 \leq (\tau_i) \to^{\mathrm{b}} \alpha\big) + \big(\alpha \leq \beta\big) \\ \tau = \beta \\ K = (K_0 \sqcap \prod_i K_i) \end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad E_0\,.\,(E_i)_{i=1..m} \quad : K \Rightarrow \tau|_G} \tag{7.6}$$

$$\frac{\begin{array}{l}\iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_1 : K_1 \Rightarrow \tau_1|_{G_1} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_2 : K_2 \Rightarrow \tau_2|_{G_2} \\ G = \big(G_0 \cup G_1 \cup G_2\big) + \big(\tau_0 \leq \alpha\big) + \big(\alpha \leq \texttt{bool}\big) + \left(\begin{smallmatrix}\tau_0 \in \texttt{true} \\ \tau_1 \leq \beta\end{smallmatrix}\right) + \left(\begin{smallmatrix}\tau_0 \in \texttt{false} \\ \tau_2 \leq \beta\end{smallmatrix}\right) \\ \tau = \beta \\ K = K_0 \sqcap K_1 \sqcap K_2\end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \texttt{if}\ E_0\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \texttt{fi} \quad : K \Rightarrow \tau|_G} \tag{7.7}$$

$$\frac{\begin{array}{l}\iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E'_0 : K'_0 \Rightarrow \tau'_0|_{G'_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E'_1 : K'_1 \Rightarrow \tau'_1|_{G'_1} \\ \text{unique\_bound\_pos}(\tau'_0, G'_0) : c \\ G = G_0 \cup G'_0 \cup G'_1 \\ \tau = \tau_0 \\ K = K_0 \sqcap K'_0 \sqcap K'_1 \sqcap \big(\emptyset, \emptyset, \{c \mapsto (\tau'_1, \alpha)\}, \emptyset\big)\end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad E_0\ @\ [E'_0 \leftarrow E'_1] \quad : K \Rightarrow \tau|_G} \tag{7.8}$$

$$\frac{\begin{array}{l}\iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E'_1 : K'_1 \Rightarrow \tau'_1|_{G'_1} \\ G = (G_0 \cup G'_1) + (\tau'_1 \leq \gamma) + (\alpha \leq \beta) \\ \tau = \tau_0 \\ K = K_0 \sqcap K'_1 \sqcap \big(\emptyset, \{\phi_x \mapsto \alpha\}, \emptyset, \{\phi_x \mapsto (\beta, \gamma, \delta)\}\big)\end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad E_0\ @\ [\underline{\phi_x} \leftarrow E'_1] \quad : K \Rightarrow \tau|_G} \tag{7.9}$$

$$\frac{\begin{array}{l}\iota, \zeta^{\tau} \overset{\tau}{\vdash} E_0 : K' \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^{\tau} \overset{\tau}{\vdash} E_i : K_i \Rightarrow \tau_i|_{G_i} \\ K_\lambda(\phi_x) : \tau \\ G = G_0 + (\tau \to \tau_0 \leq \alpha) \\ \tau = \alpha \\ K = \big(K'_x, K'_\lambda \lhd \{\phi_x\}, K'_c, K'_{\underline{\phi}}\big) \sqcap \bigsqcap_i K_i\end{array}}{\iota, \zeta^{\tau} \overset{\tau}{\vdash} \quad \lambda^{\mathrm{v}}_{\circ}\ \{E_i\}_{i=1..m}\ \Phi\ \phi_x \to E_0 \quad : K \Rightarrow \tau|_G} \tag{7.10}$$

$$\frac{\begin{array}{l}\iota, \zeta^\tau \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^\tau \overset{\tau}{\vdash} E_1 : K_1 \Rightarrow \tau_1|_{G_1} \\ G = (G_0 \cup G_1) + (\alpha \leq \beta) + (\tau_0 \leq \tau_1 \to^{\mathrm{v}} \alpha) \\ \tau = \beta \\ K = K_0 \sqcap K_1\end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad E_0 \,!\, E_1 \quad : K \Rightarrow \tau|_G} \tag{7.11}$$

$$\frac{\begin{array}{l}\iota, \zeta^\tau \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ \iota, \zeta^\tau \overset{\tau}{\vdash} E_i : K_i \Rightarrow \tau_i|_{G_i} \\ G = \big(G_0 \cup \bigcup_i G_i\big) + \big(\uparrow\tau_0 \leq \alpha\big) \\ \tau = \alpha \\ K = K_0 \sqcap \bigsqcap_i K_i\end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad \uparrow\{E_i\}_{i=1..m}\, E_0 \quad : K \Rightarrow \tau|_G} \tag{7.12}$$

$$\frac{\begin{array}{l}\iota, \zeta^\tau \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ G = G_0 + (\tau_0 \leq \uparrow\alpha) + (\alpha \leq \beta) \\ \tau = \beta \\ K = K_0\end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad \downarrow E_0 \quad : K \Rightarrow \tau|_G} \tag{7.13}$$

$$\frac{\begin{array}{l}\iota, \zeta^\tau \overset{\tau}{\vdash} E_i : \sigma_i = K_i \Rightarrow \tau_i|_{G_i} \\ K' = \bigsqcap_i K_i \\ G' = (\bigcup_i G_i) + \dot{+}_i(\tau_i \leq \alpha_i) + (\alpha_i \leq \beta_i) + (\beta_i \leq \tau_i^{\mathrm{c}}) \\ \zeta = \zeta^\tau \dagger \big\{x_i \mapsto ((K'_x \lhd \bigcup_i \{x_i\}, K'_\lambda, K'_c, K'_{\underline{\phi}}) \Rightarrow \tau_i|_{G'})\big\} \\ \iota, \zeta \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ G = G_0 \\ \tau = \tau_0 \\ K = K_0\end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad E_0 \ \texttt{wherevar}\ x_i = E_i\ \texttt{end}_{i=1..m} \quad : K \Rightarrow \tau|_G} \tag{7.14}$$

$$\frac{\begin{array}{l}x \notin \operatorname{dom} \zeta^\tau \\ G = \emptyset + (\alpha \leq \beta) \\ \tau = \beta \\ K = \big(\{x \mapsto \alpha\}, \emptyset, \emptyset, \emptyset\}\big)\end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad x \quad : K \Rightarrow \tau|_G} \tag{7.15}$$

$$\frac{x \in \operatorname{dom} \zeta^\tau \qquad \sigma = \operatorname{rename}\big(\zeta^\tau(x)\big)}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad x \quad : \sigma} \tag{7.16}$$

$$\frac{\begin{array}{l} \iota, \zeta^\tau \overset{\tau}{\vdash} E_i : K_i \Rightarrow \tau_i|_{G_i} \\ \zeta = \zeta^\tau \dagger \{\phi_{x_i} \mapsto \tau_i\} \\ \iota, \zeta \overset{\tau}{\vdash} E_0 : K_0 \Rightarrow \tau_0|_{G_0} \\ G = G_0 \cup \bigcup_i G_i \\ \tau = \tau_0 \\ K = K_0 \sqcap \bigsqcap_i K_i \end{array}}{\iota, \zeta^\tau \overset{\tau}{\vdash} \quad E_0 \ \texttt{wheredim}_\circ \ \phi_{x_i} \leftarrow E_i \ \texttt{end}_{i=1..m} \quad : K \Rightarrow \tau|_G} \tag{7.17}$$

We explain each of the cases below.

$^m c$ For an m-ary constant $^m c$, we look up its interpretation $d = \iota(^m c)$. If d is a constant of arity zero, then it is the same as for the type of d, otherwise, it is a base function.

$\underline{\phi_x}$ To type the contextual lookup of constant dimension ϕ_x, which will always be the lookup of a function parameter or a local dimension declared in a `wheredim` clause, we allocate a fresh variable α for the upper bound of the parameter ϕ_x, and the type is the lower bound β. The two are linked with the constraint $(\alpha \leq \beta)$.

`#`.$\underline{\phi_x}$ For the type of a lookup of a dimension passed as a parameter to a function with argument ϕ_x, we place the same information in the type context and constraint graph as in $\underline{\phi_x}$. Additionally, we have the variable β for the eventual actual parameter c of the function defining ϕ_x, the variable γ for the lower bound of the ordinate of c, and δ for the upper bound of the ordinate of c.

`#`.E For a context lookup to be well-typed, the expression defining the dimension to lookup must have a unique bound. When the unique bound is a constant c, we place type variable α (resp. β) in the type context, which represents the lower (resp. upper) bound of the ordinate of c. The type of the lookup is the lower bound α.

$\lambda^{\mathrm{b}} \ \{E_i\}_{i=1..m} \ x \to E_0$
$\lambda^{\mathrm{v}} \ \{E_i\}_{i=1..m} \ x \to E_0$
$\uparrow\{E_i\}_{i=1..m} \ E_0$ For the type of a base function abstraction, we determine that the type of the body E_0 is τ_0, and look up the types, τ_j, of the parameters, x_j, required by the type context K_0. The type of the abstraction is a new type variable α, where its

lower bound is a base function type construct from the inferred τ_j types, $\big((\tau_j) \xrightarrow{b} \tau_0\big)$. The type of a call-by-value (resp. intension) abstraction is the same as for base function abstraction, except that there is one (resp. zero) parameter.

$E_0 \,.\, (E_i)_{i=1..m}$
$E_0 \,!\, E_1$
$\downarrow E_0$ For the type of a base function application, the subexpressions E_0 and E_i have type τ_0 and τ_i, respectively. We generate constraints that require that τ_0 be a smaller type than $\big((\tau_i) \xrightarrow{b} \alpha\big)$, which is some function mapping the types of the arguments to a new type variable α.

`if` E_0 `then` E_1 `else` E_2 `fi` The type of a conditional expression is specified using conditional constraints. The condition E_0 has type τ_0, the upper bound of which must be **bool**. Then the type of the conditional includes the `true` (resp. `false`) branch if τ_0 includes the type `true` (resp. `false`.)

E_0 `@` $[E'_0 \leftarrow E'_1]$ An unbounded context change expression must have a unique bound for the dimension that is being changed, which is the context change pair of (7.4). If the unique bound is a constant c, then we return in the type context that the type of expression E'_1 is a lower bound for the ordinate of c.

When E'_0 is of the form $\underline{\phi_x}$, the dimension being changed is a dimension passed as a function parameter ϕ_x. The function whose parameter is ϕ_x will later be applied to a constant c, and the type of expression E'_1 provides a lower bound for the ordinate of c.

E_0 `wherevar` $x_i = E_i$ `end`$_{i=1..m}$ For the type of a `wherevar` clause, it is assumed that the variables defined are a group of mutually recursive variables. Expressions E_i define each variable x_i, and have types τ_i. In the type context, K_i for each expression will be an upper bound for each identifier. We combine all the K_i to produce K', then link the upper bounds to the lower bounds of each identifier with the constraints $\{\tau_i \leq K'_x(x_i)\}$. The type environment is then perturbed with the resulting type schemes, and the body E_0 has its type inferred.

x There are two rules for x, rule (7.15) is for when x is not in the domain of the type environment ζ^τ, and rule (7.16) is for when it is in the domain. The former will be matched when inferring the type of an identifier that is defined in the same mutually recursive group that is currently having the types of its variables inferred. In this case, there is no type information for the identifier, so we treat the identifier in the same way as a function parameter. Its type is constrained from above by its use inside the mutually recursive group, and constrained from below by the type of the expression defining it.

In the case of the latter rule we do have type information for the identifier, so we simply look up the type scheme in the type environment, and rename each of the type variables that appear in the type scheme.

E_0 `wheredim` $x_i \leftarrow E_i$ `end`$_{i=1..m}$ For the type of a `wheredim` clause, the expression initialising each dimension x_i provides a lower bound for the ordinate of x_i, and the use of x_i inside the body E_0 provides an upper bound.

7.8 Simplifying constraints

Even for fairly simple expressions, a large number of constraints is generated. For example, type inference of the TransLucid standard header generates several thousand constraints before simplification! Nevertheless, most of these constraints are unnecessary for the final result, as they only play an intermediate role in computing the final type of an expression. Furthermore, it is next to impossible for the user to decipher the meaning of more than a few constraints. Fortunately, it is often possible to reduce the number of constraints to just a few, and from there it is often possible to display the type to the user as a single type term, not subject to the minimal-height types invariant, but more useful to the user.

The process of simplifying constraints always takes as input a type scheme, and produces as output a type scheme that denotes the same type, but is written down in a different manner. The simplification of a type scheme is a four-stage process:

1. Canonisation (§7.9), in which all occurrences of the form $\sqcup V$ or $\sqcap V$ are replaced by new type variables, and appropriate constraints are added so that the denoted type scheme is the same.

2. Garbage collection (§7.10), in which all superfluous constraints are removed from a type scheme. When computing the type of an expression, many of the constraints generated are generated in computing the type of subexpressions, and only play an intermediate role in computing the final type, playing no role in denoting the final type, hence can be removed.

3. Minimisation (§7.11), in which a type scheme is minimised in a process similar to deterministic finite automata minimisation, which identifies sets of equivalent type variables, and for each of these sets, replaces all members by a single variable throughout the type scheme.

4. Displaying (§7.12), in which a type scheme is displayed in a manner readable to the user.

There is a question as to when to apply these simplifications. In his thesis, Pottier suggests that a type scheme be simplified before it is placed in the environment. For him, that means simplifying the type of an identifier in a `let` expression; in our case, we simplify a type scheme before placing it in the environment during the typing of a whole program. The simplifications can be applied at any point and in almost any order. The only restriction is that the input to garbage collection is a type scheme with no occurrences of $\sqcap$ or $\sqcup$. To achieve the best results, a type scheme should be canonised, then garbage collected, finally minimised.

We will use the Y-combinator, as defined in TransLucid, as an example for simplifying a type scheme. In concrete syntax, it is defined as

$$\lambda^{\mathrm{n}} f \to \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)\ \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)$$

which is translated to the following abstract syntax

$$\lambda^{\mathrm{v}} \phi_f \to \left(\lambda^{\mathrm{v}} \phi_a \to \downarrow\underline{\phi_f}\,!\,(\downarrow\underline{\phi_a}\,!\,\downarrow\underline{\phi_a})\right)\,!\,\left(\lambda^{\mathrm{v}} \phi_b \to \downarrow\underline{\phi_f}\,!\,(\downarrow\underline{\phi_b}\,!\,\downarrow\underline{\phi_b})\right)$$

The full example for the Y-combinator is presented in Appendix 7.15. Nevertheless, here we present a summary. Initially, the expression generates 49 constrained type variables, which grow to 66 with canonisation. However, this is not a problem, since garbage collection removes most of those constraints, after which only 7 constrained type variables remain. Two of those are equivalent according to the minimisation algorithm, so after the full simplification process, only 6 constrained type variables remain. Even a constraint graph with only 6 type variables is not particularly easy for the user to understand, but with the mechanism to display types in a sane way, the Y-combinator in TransLucid has the type $\uparrow(\uparrow\alpha \xrightarrow{v} \alpha) \xrightarrow{v} \alpha$. (All types are implicitly universally quantified over all type variables, so we do not need $\forall\alpha$ at the beginning of the type.)

7.9 Canonisation

In this section, we present the canonisation algorithm. It takes a type scheme, and returns a type scheme in which all occurrences of the form $\sqcup V$ or $\sqcap V$ are replaced by new type variables, and new constraints are added for said variables; the resulting type scheme denotes the same type.

To remove the $\sqcup$ (resp. $\sqcap$), for each set V of type variables, each occurrence of $\sqcup V$ (resp. $\sqcap V$) is replaced with the variable λ_V (resp. γ_V), and the set of constraints $\{\alpha \leq \lambda_V \mid \alpha \in V\}$ (resp. $\{\gamma_V \leq \alpha \mid \alpha \in V\}$) is added. This is not sufficient, as the closure of the constraint graph must be maintained, and only adding the necessary constraints with the closure algorithm could reintroduce $\sqcup$ or $\sqcap$. Therefore, it is necessary to define a set of rewriting rules that add the necessary constraints and maintain closure of the constraint graph while guaranteeing that neither $\sqcup$ nor $\sqcap$ are re-added to the graph.

Definition 54. *The* canonisation *of a type scheme $K \Rightarrow \tau|_G$ is*

$$\mathrm{canonise}(K \Rightarrow \tau|_G) = (K' \Rightarrow \tau'|_{G'})$$

where K', τ' and G' are defined below, and the functions r^+ and r^- are defined in Figure 7.4.

If K is as follows:

$$K = \begin{pmatrix} \{x_i \mapsto \tau_i\}, \\ \{\phi_{x_j} \mapsto \tau_j\}, \\ \{d_k \mapsto (\tau_{0k}, \tau_{1k}\}, \\ \{\phi_{x_l} \mapsto (\tau_{0l}, \tau_{1l}, \tau_{2l})\} \end{pmatrix}$$

then

$$K' = \begin{pmatrix} \{x_i \mapsto r^-(\tau_i)\}, \\ \{\phi_{x_j} \mapsto r^-(\tau_j)\}, \\ \big\{d_k \mapsto \big(r^+(\tau_{0k}), r^-(\tau_{1k})\big)\big\}, \\ \big\{\phi_{x_l} \mapsto (r^+(\tau_{0l}), r^+(\tau_{1l}), r^-(\tau_{2l}))\big\} \end{pmatrix}$$

We define $\tau' = r^+(\tau)$, and G' is defined in Figures 7.5 and 7.6.

$$\begin{aligned} r^+(\mathbf{t}) &= \mathbf{t} & r^-(\mathbf{t}) &= \mathbf{t} \\ r^+(\alpha) &= \alpha & r^-(\alpha) &= \alpha \\ r^+(\sqcup V) &= \lambda_V & r^-(\sqcap V) &= \gamma_V \\ r^+(\bot) &= \bot & r^-(\bot) &= \bot \\ r^+(\top) &= \top & r^-(\top) &= \top \\ r^+\big((\tau_j) \xrightarrow{b} \tau_0\big) &= r^-(\tau_j) \xrightarrow{b} r^+(\tau_0) & r^-\big((\tau_j) \xrightarrow{b} \tau_0\big) &= r^+(\tau_j) \xrightarrow{b} r^-(\tau_0) \\ r^+(\tau_1 \xrightarrow{v} \tau_0) &= r^-(\tau_1) \xrightarrow{v} r^+(\tau_0) & r^-(\tau_1 \xrightarrow{v} \tau_0) &= r^+(\tau_1) \xrightarrow{v} r^-(\tau_0) \\ r^+(\uparrow\tau) &= \uparrow r^+(\tau) & r^-(\uparrow\tau) &= \uparrow r^-(\tau) \end{aligned}$$

Figure 7.4: Rules for canonisation rewriting function

$$\begin{aligned} G'^{\downarrow}(\alpha) &= r^+\big(G^{\downarrow}(\alpha)\big) & G'^{\uparrow}(\alpha) &= r^-\big(G^{\uparrow}(\alpha)\big) \\ G'^{\downarrow}(\gamma_V) &= \bot & G'^{\uparrow}(\gamma_V) &= r^- \left(\bigsqcap_{\alpha \in V} G^{\uparrow}(\alpha) \right) \\ G'^{\downarrow}(\lambda_V) &= r^+ \left(\bigsqcup_{\alpha \in V} G^{\downarrow}(\alpha) \right) & G'^{\uparrow}(\lambda_V) &= \top \end{aligned}$$

Figure 7.5: New bounds for canonisation

This definition of canonisation corresponds to Definition 11.5 of [34, p. 118], except for the extra variables that appear in the type context K. The only difference with the type context is that where Pottier applies r^- to the members of the two components of his type context, we apply r^- and r^+ to the appropriate members of the four components of our type context. Pottier's proof of correctness is given as Theorem 11.1 and Proposition 11.11

$$\begin{array}{rll} \alpha \leq_{G'} \beta & \text{when } \alpha \leq_G \beta & \\ \gamma_V \leq_{G'} \alpha & \text{when } \exists \beta \in V \mid \beta \leq_G \alpha & \\ \alpha \leq_{G'} \lambda_V & \text{when } \exists \beta \in V \mid \alpha \leq_G \beta & \\ \gamma_V \leq_{G'} \lambda_T & \text{when } \exists \alpha \in V \quad \exists \beta \in T & \alpha \leq_G \beta \end{array}$$

Figure 7.6: New less than relation for canonisation

of [34, pp.113,119]. The addition of guarded constraints, corresponds to Definition 20, with the proof of its correctness given as Theorem 3 in [35, pp.7,9]. The change in K does not affect the proof, all we are doing is ensuring that every type can be rewritten. The system presented in [35] only adds to the system presented in [34], and in fact refers to the proof from the latter for part of its result. Since our system is consistent with the one presented in the former, we can state that the definition of garbage collection here is correct.

To return to the Y-combinator example, there are several occurrences of $\sqcup$ and $\sqcap$ in its type, all of which are removed, and extra constraints generated by canonisation. This results in the type having 66 constrained type variables.

7.10 Garbage collection

In this section, we present the garbage-collection algorithm, which removes constraints from a type scheme that are not relevant to the type denoted by that type scheme. During type inference, many constraints are generated which play a part in generating the final type, but whose information has been incorporated into other constraints, and no longer convey any information about the type. All these constraints do is clutter the type scheme, make it slower to process by the computer, and make it harder for the user to read. Therefore, it is desirable to remove them.

To determine which type variables are superfluous, we compute which variables play a part in the type denoted by the type scheme. To do that, we present the idea of *polarity*, which is that we say that a variable can be positive, negative, bipolar, or neutral.

Polarity is a specific form of reachability, which is to determine which type variables in a type scheme actually contribute to denoting the type. A coarse notion of reachability is simply that a type variable v in G is *reachable* if v appears in K or τ, or if it appears in the upper or lower bounds or the less than relation in G of any other reachable variable. If a variable cannot be reached, then it can be discarded from the constraint graph, because it plays no part in the denotation of the type.

We will consider an example to demonstrate the coarse idea of reachability, and then use the same example to demonstrate the refinement that polarity makes to reachability, which is used to define, garbage collection. Consider the type scheme $\emptyset \Rightarrow \left(\alpha \xrightarrow{v} \gamma\right)|_G$,

where G is defined by the constraints graph:

$$\begin{array}{rcl} & \alpha \leq \beta, \gamma \quad \delta \xrightarrow{v} \epsilon \\ \alpha \leq \beta \leq \gamma \quad \delta \xrightarrow{v} \epsilon \\ \alpha, \beta \leq \gamma \leq \delta \xrightarrow{v} \epsilon \\ \alpha \xrightarrow{v} \gamma \leq \lambda \\ \top \leq \epsilon \end{array}$$

From the point of view of our coarse notion of reachability, we look at the type $\alpha \xrightarrow{v} \gamma$ and see that α and γ are reachable; then looking at the constraints on α and γ, we see that β, δ and ϵ are also reachable. The only variable that is not reachable is λ, so it can clearly be removed, as it plays no part in the type.

The idea of polarity is finer than what we just mentioned, by taking into account the direction of data flow. In a type scheme, τ is a pos-type, which represents an output, so it is marked as *positive*, and K is made up of both pos-types and neg-types, which represent outputs and inputs respectively. So each neg-type in K is marked as *negative*, and each pos-type is marked as positive. This idea of marking type variables we call *polarity*, and we say that a type variable in a type scheme can be positive, negative, bipolar or neutral. The rules for propagation of polarity are finer than for reachability. If a variable is positive, its lower bound is marked as positive; if the variable is negative, then its upper bound is marked as negative.

To revisit the previous example and mark the polarity of each variable, we notice that since $\alpha \xrightarrow{v} \gamma$ is an output, α represents an input and is marked as negative, and γ represents an output and so is marked as positive. So now we mark the upper bound of α as negative, and the lower bound of γ as positive, which results in δ being positive, and ϵ being negative. There is nothing further to do, so the positive type variables are $\{\gamma, \delta\}$ and the negative type variables are $\{\alpha, \epsilon\}$; the remaining variables are neutral.

Our definition of polarity corresponds to Definition 10.3 in [34, p.104]. The only difference being that our type context K contains more information, and polarity takes into account all of the type variables mentioned in K.

Definition 55. *The polarity of a type variable in a type scheme $\sigma = (K, \tau, G)$ is a subset of $\{+, -\}$, and we say that a variable is either positive, negative, bipolar or neutral. We define the function* polarity*, which takes as input a type scheme σ, and a type variable v, and returns a subset of $\{+, -\}$:*

$$\epsilon \in \text{polarity}(\sigma, v) \iff v \in V^{\epsilon}$$

where V^+ and V^- are defined below.

Computing polarity makes use of two auxiliary functions,

$$\text{mark}^+ : \mathbb{T} \to \big(\{P, N\} \to V\big)$$
$$\text{mark}^- : \mathbb{T} \to \big(\{P, N\} \to V\big)$$

where each returns a set of positive and negative type variables in τ based on their structure. Both functions are defined in Figure 7.7.

The computation of polarity is an iterative process, where in iteration i we compute the sets V_i^+ and V_i^-. The results V^+ and V^- are the least fixed points of V_i^+ and V_i^-.

$$
\begin{aligned}
\text{let} \quad & P_x = \text{mark}^-(K_x) \\
& P_\lambda = \text{mark}^-(K_\lambda) \\
& (d_0, d_1) = K_d \\
& P_d = \text{mark}^+(d_0) \cup \text{mark}^-(d_1) \\
& (\underline{\phi}_0, \underline{\phi}_1, \underline{\phi}_2) = K_{\underline{\phi}} \\
& P_{\underline{\phi}} = \text{mark}^+(\underline{\phi}_0) \cup \text{mark}^+(\underline{\phi}_1) \cup \text{mark}^-(\underline{\phi}_2)
\end{aligned}
$$

$$
\begin{aligned}
\text{in} \quad & V_0^+ = \text{mark}^+(\tau)(P) \quad \cup \quad \textstyle\bigcup_{v \in dom\, G} \big(P_x(P) \cup P_\lambda(P) \cup P_d(P) \cup P_{\underline{\phi}}(P)\big) \\
& V_0^- = \text{mark}^-(K) \quad \cup \quad \textstyle\bigcup_{v \in dom\, G} \big(P_x(N) \cup P_\lambda(N) \cup P_d(N) \cup P_{\underline{\phi}}(N)\big) \\
& V_{i+1}^+ = \bigcup_{v \in V_i^+} \text{mark}^+\big(G^\downarrow(v)\big)(P) \quad \cup \quad \bigcup_{v \in V_i^-} \text{mark}^-\big(G^\uparrow(v)\big)(P) \\
& V_{i+1}^- = \bigcup_{v \in V_i^+} \text{mark}^+\big(G^\downarrow(v)\big)(N) \quad \cup \quad \bigcup_{v \in V_i^-} \text{mark}^-\big(G^\uparrow(v)\big)(N) \\
& V^+ = \text{lfp}\ V_i^+ \\
& V^- = \text{lfp}\ V_i^-
\end{aligned}
$$

After marking all variables, the garbage collection process is quite simple. We only keep each component from the constraint graph under one of the following four conditions:

1. $\alpha \leq_G \beta$ if $\alpha \in V^-$ and $\beta \in V^+$.
2. $G^\downarrow(\alpha)$ if $\alpha \in V^+$.
3. $G^\uparrow(\alpha)$ if $\alpha \in V^-$.
4. $\beta \overset{\alpha \in s}{\leq_G} \gamma$ if $\alpha \in V^-$.

Otherwise, the constraint can be removed from the graph.

This definition of garbage collection corresponds to Pottier's Definition 10.5, with the proof of its correctness given in Theorem 10.1 of [34, p.106]. The addition of guarded constraints corresponds to Definition 18, with the proof of correctness given in Theorem 2 of [35, p.6]. The addition of guarded constraints is correct for the same reason given in §7.9.

To finish our example, as the variables $\{\gamma, \delta\}$ are positive and $\{\alpha, \epsilon\}$ are negative, the

$$
\begin{aligned}
\mathrm{mark}^+(\alpha) &= \{P \mapsto \{\alpha\}, N \mapsto \emptyset\} \\
\mathrm{mark}^-(\alpha) &= \{P \mapsto \emptyset, N \mapsto \{\alpha\}\} \\
\mathrm{mark}^+((\tau_j) \xrightarrow{b} \tau_0) &= \text{let} \quad l_j = \mathrm{mark}^-(\tau_j) \\
&\qquad\quad r = \mathrm{mark}^+(\tau_0) \\
&\qquad \text{in} \quad \{P \mapsto \left(\bigcup_j l_j(P)\right) \cup r(P), N \mapsto \left(\bigcup_j l_j(N)\right) \cup r(N)\} \\
\mathrm{mark}^-\big((\tau_j) \xrightarrow{b} \tau_0\big) &= \text{let} \quad l_j = \mathrm{mark}^+(\tau_j) \\
&\qquad\quad r = \mathrm{mark}^-(\tau_0) \\
&\qquad \text{in} \quad \{P \mapsto \left(\bigcup_j l_j(P)\right) \cup r(P), N \mapsto \left(\bigcup_j l_j(N)\right) \cup r(N)\} \\
\mathrm{mark}^+\big(\tau_0 \xrightarrow{v} \tau_1\big) &= \text{let} \quad \mathit{lhs} = \mathrm{mark}^-(\tau_0) \\
&\qquad\quad \mathit{rhs} = \mathrm{mark}^+(\tau_1) \\
&\qquad \text{in} \quad \{P \mapsto \mathit{lhs}(P) \cup \mathit{rhs}(P), N \mapsto \mathit{lhs}(N) \cup \mathit{rhs}(N)\} \\
\mathrm{mark}^-(\tau_0 \xrightarrow{v} \tau_1) &= \text{let} \quad \mathit{lhs} = \mathrm{mark}^+(\tau_0) \\
&\qquad\quad \mathit{rhs} = \mathrm{mark}^-(\tau_1) \\
&\qquad \text{in} \quad \{P \mapsto \mathit{lhs}(P) \cup \mathit{rhs}(P), N \mapsto \mathit{lhs}(N) \cup \mathit{rhs}(N)\} \\
\mathrm{mark}^+(\uparrow\tau) &= \mathrm{mark}^+(\tau) \\
\mathrm{mark}^-(\uparrow\tau) &= \mathrm{mark}^-(\tau)
\end{aligned}
$$

Figure 7.7: Rules for polarity decomposition

constraint graph can be reduced to the two constraints:

$$
\begin{aligned}
\alpha &\leq \delta \xrightarrow{v} \epsilon \\
\alpha &\leq \gamma
\end{aligned}
$$

To continue with the Y-combinator example, it is now useful to present the type scheme after garbage collection, as it has been reduced to just 7 constrained type variables. Its type is $\emptyset \Rightarrow v_2|_G$, where G equals (instead of Greek letters, we have here used v_i for type variables):

$$
\begin{aligned}
v_6 &\leq v_1 \\
(v_3 \xrightarrow{v} v_1) &\leq v_2 \\
v_3 &\leq \uparrow v_4 \\
v_4 &\leq (v_5 \xrightarrow{v} v_6) \\
\uparrow v_7 &\leq v_5 \\
v_6 &\leq v_1, v_7 \\
v_6 &\leq v_7
\end{aligned}
$$

7.11 Minimisation

In this section we present minimisation, which takes as input a type scheme $K \Rightarrow \tau|_G$, and produces as output an equivalent type scheme in which sets of variables that are equivalent have been merged together. This is an adaptation of DFA minimisation, with the definition of what makes two variables equivalent being specific to this problem.

At this point in the simplification process, we will have a type scheme that has no occurrences of $\sqcup$ or $\sqcap$, and no superfluous constraints. However, we might still not have an optimal representation for the type scheme. Especially due to the nature of canonisation, it is possible that there will be groups of type variables that are all equivalent, and each group can be replaced with one type variable.

Broadly speaking, two variables are equivalent if nothing distinguishes them: if they are less than and greater than the same variables, if they have the same polarity and if their lower bounds and upper bounds are equivalent. The lower and upper bounds do not have to be equal, rather, if the respective variables that make up one of the bounds are equivalent, then the bounds are equivalent. For example, for the types $\alpha \xrightarrow{v} \beta$ and $\gamma \xrightarrow{v} \delta$, if α and γ are equivalent, and β and δ are equivalent, then those two types are equivalent.

It is necessary to define two functions:

Definition 56. *Let G be a constraint graph, and $\alpha \in dom\, G$, then*

$$\begin{aligned}\mathrm{pred}_G(\alpha) &= \{\beta \mid \beta \leq_G \alpha\}\\ \mathrm{succ}_G(\alpha) &= \{\beta \mid \alpha \leq_G \beta\}\end{aligned}$$

Definition 57. *Two variables α and $\beta \in \mathrm{dom}\, G$ are said to be equivalent if the following conditions hold, then we write $\alpha \equiv^G \beta$:*

$$\begin{aligned}&\mathrm{pred}_G(\alpha) = \mathrm{pred}_G(\beta)\\ &\mathrm{succ}_G(\alpha) = \mathrm{succ}_G(\beta)\\ &\mathrm{head}\big(G^{\downarrow}(\alpha)\big) = \mathrm{head}\big(G^{\downarrow}(\beta)\big)\\ &\mathrm{head}\big(G^{\uparrow}(\alpha)\big) = \mathrm{head}\big(G^{\uparrow}(\beta)\big)\\ &\mathrm{polarity}_G(\alpha) = \mathrm{polarity}_G(\beta)\\ &\textit{if } G^{\uparrow}(\alpha) = (v_j) \xrightarrow{b} v_0 \textit{ and } G^{\uparrow}(\beta) = (v'_j) \xrightarrow{b} v'_0 \textit{ and } v_0 \equiv^G v'_0 \wedge \bigwedge_j v_j \equiv^G v'_j\\ &\textit{if } G^{\downarrow}(\alpha) = (v_j) \xrightarrow{b} v_0 \textit{ and } G^{\downarrow}(\beta) = (v'_j) \xrightarrow{b} v'_0 \textit{ and } v_0 \equiv^G v'_0 \wedge \bigwedge_j v_j \equiv^G v'_j\\ &\textit{if } G^{\uparrow}(\alpha) = v_1 \xrightarrow{v} v_0 \textit{ and } G^{\uparrow}(\beta) = v'_1 \xrightarrow{v} v'_0 \textit{ and } v_0 \equiv^G v'_0 \wedge v_1 \equiv^G v'_1\\ &\textit{if } G^{\downarrow}(\alpha) = v_1 \xrightarrow{v} v_0 \textit{ and } G^{\downarrow}(\beta) = v'_1 \xrightarrow{v} v'_0 \textit{ and } v_0 \equiv^G v'_0 \wedge v_1 \equiv^G v'_1\\ &\textit{if } G^{\uparrow}(\alpha) = \uparrow v \textit{ and } G^{\uparrow}(\beta) = \uparrow v' \textit{ and } v \equiv^G v'\\ &\textit{if } G^{\downarrow}(\alpha) = \uparrow v \textit{ and } G^{\downarrow}(\beta) = \uparrow v' \textit{ and } v \equiv^G v'\\ &\overset{\alpha\in s}{\gamma \leq_G \gamma'} \wedge \overset{\beta\in s}{\delta \leq_G \delta'} \implies \gamma \equiv^G \delta \wedge \gamma' \equiv^G \delta'\end{aligned}$$

The equivalence classes described in the above definition can be computed using standard algorithms for DFA minimisation, such as Hopcroft's algorithm [22].

The definition of equivalence given here is the same as the one given by Pottier in Chapter 13 of [34, p.135] and [35], except that it has been adapted to the TransLucid types. The proofs of correctness are given in Lemma 13.3 and Theorem 13.2 of the former, and Theorem 4 of the latter. Our system is correct for the same reason given in §7.9.

Let us return to our Y-combinator example, after canonisation and garbage collection, its type is as given at the end of the previous section. The type variables v_1 and v_7 are equivalent, as the only constraint on the two is that each is greater than v_6. Since they are equivalent, we can substitute all instances of v_7 with v_1, which removes v_7 from the constraint graph, and replaces one occurrence of $\uparrow v_7$ with $\uparrow v_1$. The resulting constraint graph is:

$$\begin{aligned} v_6 &\leq v_1 \\ (v_3 \xrightarrow{v} v_1) &\leq v_2 \\ v_3 &\leq \uparrow v_4 \\ v_4 &\leq (v_5 \xrightarrow{v} v_6) \\ \uparrow v_1 &\leq v_5 \\ v_6 &\leq v_1 \end{aligned}$$

The Y-combinator does not have any constructed bounds, so it is trivial to see where the equivalent variables are. As a better illustration, consider the following constraint graph:

$$\begin{aligned} v_1 &\leq v_5, v_6 \\ v_2 &\leq v_5, v_6 \\ v_5 \xrightarrow{v} v_1 &\leq v_3 \\ v_6 \xrightarrow{v} v_2 &\leq v_4 \\ v_1, v_2 &\leq v_5 \\ v_1, v_2 &\leq v_6 \end{aligned}$$

The variables v_1 and v_2 are equivalent, as they both have v_5 and v_6 as their successors, similarly, v_5 and v_6 are equivalent. The purpose of this example is to demonstrate that v_3 and v_4 are equivalent. For the lower bounds of v_3 and v_4, the left-hand sides are v_5 and v_6 respectively. Since v_5 and v_6 are equivalent, the left-hand side does not differentiate v_3 and v_4. Similarly, the right-hand sides are equivalent. Therefore, v_3 and v_4 are equivalent, and the whole constraint graph can be collapsed to just three type variables:

$$\begin{aligned} v_1 &\leq v_3 \\ v_3 \xrightarrow{v} v_1 &\leq v_2 \\ v_1 &\leq v_3 \end{aligned}$$

7.12 External display

In this section we present a means for types to be presented in a readable manner to the user: i.e., typically as one-type terms. The fact that no type can have depth more than one means that a large number of type variables are used in the constraint graph of the type; this can be very difficult for the user to read. At this point, the only purpose of this simplification is to display the type in a more readable manner to the user; it is not so useful for the computer for any further analysis.

For example, whilst the displayed type of the Y-combinator is $\uparrow(\uparrow\alpha \xrightarrow{v} \alpha) \xrightarrow{v} \alpha$, its inferred type is more complex. We reproduce its type here for reference, although it is the same as the type given in the previous section. Its type is $\emptyset \Rightarrow v_2|_G$, where G is the

constraint graph (instead of Greek letters, we have here used v_i for type variables):

$$\begin{array}{rcl} v_6 & \leq & v_1 \\ (v_3 \xrightarrow{v} v_1) & \leq & v_2 \\ & & v_3 \leq \uparrow v_4 \\ & & v_4 \leq (v_5 \xrightarrow{v} v_6) \\ \uparrow v_1 & \leq & v_5 \\ & & v_6 \leq v_1 \end{array}$$

which is difficult to interpret. More complex functions only get worse.

Fortunately, it is quite simple to produce a readable type: if a type variable has a unique bound, then it can be replaced by that bound as long as the variable does not appear free in the said bound. There is an extra condition required by guarded constraints: if a type variable appears in a guarded constraint, then it cannot be simplified.

We previously presented the definition of the unique bound for positive variables in Definition 53, here we present the symmetric definition for negative variables.

Definition 58. *Given a constraint graph G, the* unique bound *of a negative type variable v is given by the function* $\text{unique\_bound\_neg}(v, G)$.

$$\text{unique\_bound\_neg}(v, G) = \begin{cases} v', & G^{\uparrow}(v) \equiv \top \text{ and } v' \text{ is the unique } v' \text{ s.t. } v \leq_G v' \\ \tau, & G^{\uparrow}(v) \equiv \tau \text{ and there is no } v' \text{ s.t. } v \leq_G v' \end{cases}$$

Definition 59. *Let $A \Rightarrow \tau|_G$ be a type scheme. Let α be a type variable in the domain of G. Then α can be replaced with its unique bound under the following two conditions:*

1. $\text{unique\_bound\_pos}(\alpha, G)$ *is defined if α is a positive variable, or* $\text{unique\_bound\_neg}(\alpha, G)$ *is defined if α is a negative variable.*
2. $\forall s, \alpha, \beta, \gamma$ *s.t.* $\left(b \overset{a \in s}{\leq} c\right) \in G$, $\alpha \not\equiv a \wedge \alpha \not\equiv b \wedge \alpha \not\equiv c$

Then, it is simply a matter of repeatedly replacing variables with their unique bound until this process is no longer possible.

Finally, we will go through the above process with the Y-combinator, so that it can be seen how to get from the above type scheme to its display type. The type of the Y-combinator is v_2, which is positive, so we start by replacing that with its lower bound, $(v_3 \xrightarrow{v} v_1)$. The variables v_1 and v_3 are positive and negative respectively, so we replace them with their lower and upper bounds and get $(\uparrow v_4 \xrightarrow{v} v_6)$. Here, we must make an arbitrary decision; replacing v_6 with its upper bound, v_1, would result in going in circles, continually replacing v_6 with v_1 and v_1 with v_6. So when a variable has one predecessor or successor, we stick with the smaller variable, in this case v_6. Next we replace v_4 with its upper bound, so the type is now $(\uparrow(v_5 \xrightarrow{v} v_6) \xrightarrow{v} v_6)$. We now replace v_5 with $\uparrow v_1$, which gives us $(\uparrow(\uparrow v_1 \xrightarrow{v} v_6) \xrightarrow{v} v_6)$. Then finally, replacing v_1 with v_6 gives us $(\uparrow(\uparrow v_6 \xrightarrow{v} v_6) \xrightarrow{v} v_6)$. We can then arbitrarily replace the v_n type variables with alphabetic type variables, and state that the type of the Y-combinator is $(\uparrow(\uparrow \alpha \xrightarrow{v} \alpha) \xrightarrow{v} \alpha)$.

The usefulness of minimisation can now be seen. Without it, displaying the Y-combinator type scheme in a readable manner would be impossible because the variable v_6 did not have a unique bound before minimisation.

7.13 Examples

This section presents examples of the types that are inferred for some of the standard TransLucid functions.

7.13.1 `fby`

The *fby* function is defined as

$$\texttt{fun}\ \mathit{fby}.d\ X\ Y = \texttt{if}\ \#.d \leq 0\ \texttt{then}\ X\ \texttt{else}\ Y\ \texttt{@}\ [d \leftarrow \#.d - 1]\ \texttt{fi}$$

which is translated to abstract syntax as

$$\begin{aligned}\texttt{var}\ \mathit{fby} = \lambda^{\mathrm{b}}\phi_{d_1} \to \lambda^{\mathrm{v}}\phi_X \to \lambda^{\mathrm{v}}\phi_Y \to \\ \texttt{if}\ \#.\underline{\phi_{d_1}} \leq 0\ \texttt{then}\ {\downarrow}\underline{\phi_X}\ \texttt{else}\ {\downarrow}\underline{\phi_Y}\ \texttt{@}\ [\underline{\phi_{d_1}} \leftarrow \#.\underline{\phi_{d_1}} - 1]\ \texttt{fi}\end{aligned}$$

Here we use ϕ_{d_1} for the ϕ dimension allocated for the first parameter, because our next example, *upon*, also has a parameter d, which we will allocate ϕ_{d_2}.

Its display type is

$$\mathit{fby} :: \alpha \xrightarrow{b} {\uparrow}\beta \xrightarrow{v} {\uparrow}\beta \xrightarrow{v} \beta$$

However, the type shown above is not its full type. Although the display type is useful for seeing an overall picture of the type, it is missing the context information, which is the following:

$$\mathbf{intmp} \leq \#.(\phi_{d_1} = \alpha) \leq \mathbf{intmp}$$

The context information here contains solely dimensions that are passed as parameters, as the *fby* function only uses the one dimension passed as a parameter. The line above describes the dimension passed as a ϕ parameter, whose type is α, which is the type of the first parameter to *fby*, and whose ordinate has lower bound **intmp**, and upper bound **intmp**. Or in other words, its ordinate is set to an integer in the function, and its usage inside the function requires it to be at most an integer.

Now to see the use of the above type, we see that if we apply *fby* to the value 0, the type of the resulting function is

$$\mathit{fby}.0 :: ({\uparrow}\alpha \xrightarrow{v} {\uparrow}\alpha \xrightarrow{v} \alpha)$$

with context

$$\mathbf{intmp} \leq \#.0 \leq \mathbf{intmp}$$

Now the context has moved from being dependent on a function parameter to knowing

the exact dimension of relevance. From the above we can see that the use of dimension 0 requires it to be no more than an **intmp**, and that it is set to something in the set **intmp**, which is consistent, and therefore the use of dimension 0 is well-typed.

7.13.2 `upon`

The standard function *upon* is defined as

$$
\begin{array}{l}
\texttt{fun}\ \mathit{upon}.d\ X\ Y = X\ \texttt{@}\ [d \leftarrow T] \\
\texttt{where} \\
\quad \texttt{var}\ T = \mathit{fby}.d\ 0\ (\texttt{if}\ Y\ \texttt{then}\ T + 1\ \texttt{else}\ T\ \texttt{fi}) \\
\texttt{end}
\end{array}
$$

which is translated to the following abstract syntax

$$
\begin{array}{l}
\mathit{upon} = \lambda^{\mathrm{b}}\phi_{d_2} \rightarrow \lambda^{\mathrm{v}}\phi_X \rightarrow \lambda^{\mathrm{v}}\phi_Y \rightarrow \downarrow\underline{\phi_X}\ \texttt{@}\ [\underline{\phi_{d_2}} \leftarrow T] \\
\texttt{wherevar} \\
\quad T = \mathit{fby}.\underline{\phi_{d_2}}\ !\ (\uparrow 0)\ !\ (\uparrow\ \texttt{if}\ \downarrow\underline{\phi_Y}\ \texttt{then}\ T + 1\ \texttt{else}\ T\ \texttt{fi}) \\
\texttt{end}
\end{array}
$$

Its display type is:

$$\mathit{upon} :: \alpha \xrightarrow{b} \uparrow\beta \xrightarrow{v} \gamma \xrightarrow{v} \beta$$

with the constraints:

$$
\begin{array}{rcl}
 & & \gamma \leq \mathbf{bool} \\
0 & \zeta \leq & \delta \\
\mathbf{intmp} & \leq & \epsilon \\
 & & \zeta \leq \delta
\end{array}
$$

with the guarded constraints:

$$
\begin{array}{c}
\texttt{false} \leq \gamma\ ?\ \eta \leq \zeta \\
\texttt{true} \leq \gamma\ ?\ \epsilon \leq \zeta
\end{array}
$$

and with the context:

$$
\begin{array}{rcccl}
\delta & \leq & \#.(\phi_{d_2} = \alpha) & \leq & \top \\
\mathbf{intmp} & \leq & \#.(\phi_{d_1} = \alpha) & \leq & \mathbf{intmp}
\end{array}
$$

Each type variable plays a role in the type of *upon*, which will be described below. The first parameter to *upon* has type α, which has no constraints, and is mentioned in the type context. The two lines which describe the type context indicate that both function parameters, ϕ_{d_1} and ϕ_{d_2}, which are the parameters of *upon* and *fby* respectively, have the type α. The first line indicates that the ordinate is bounded by the type d from below, and unbounded from above. The second line indicates that the ordinate is bounded by **intmp** from below, and **intmp** from above.

The type variable d only has partial information: the value 0 is in its type, along with whatever type information is later supplied by the type variable ζ, which comes from the

guarded constraint when *upon* is later used. The type γ is for the parameter B of *upon*, so the guarded constraints indicate that if B is `false`, then type ζ should have η as a lower bound, and if B is `true`, that ζ should have ϵ as a lower bound. The type η has no constraints, so adds nothing to the type of ζ. Since ϵ has **intmp** as its lower bound, the type **intmp** is propagated to the lower bound of ζ if B has `true` in its type. The only purpose of ζ is to propagate a type to δ, which is used in determining the type of the ordinate of the first parameter to *upon*.

The type β is also unconstrained, as *upon* does nothing with the second parameter, other than to evaluate it in the current context. Therefore, the return type of *upon* is simply the type of the second parameter.

It is impossible to come up with a completely simplified meaningful display type, since the type variable δ does not have a unique bound. Despite that, an expression that fully applies *upon* is still well-typed, and also has a nice display type. For example the expression:

$$(upon.0\ (\#.0)\ (2\ \texttt{mod}\ 0 \equiv 0))\ \texttt{@}\ [0 \leftarrow 5];;$$

has the type:

intmp

with TransLucid context information:

$$\textbf{intmp} \leq \#.0 \leq \textbf{intmp}$$

which simply means that the ordinate of dimension zero is set to an integer in the program, and is constrained to be at most an integer in the program.

7.14 Conclusions

This chapter has demonstrated a type inference algorithm for TransLucid expressions. The type inference algorithm supports parametric polymorphism, and the use of constraints with type variables allows recursive types. It is this ability to infer recursive types that allows the Y-combinator presented above to be typed with the definition given, which is not possible in Haskell.

It is important that some type be inferred for the ordinate of each dimension in the TransLucid runtime context. This is done quite effectively, with this type inference system being able to determine an upper and a lower bound for each dimension that was used in a whole program. Although the limitation that each dimension in the context be required to have the same type throughout a program seems restrictive, it does not in practice limit the programs that can be written. If dimensions with different types are required, it suffices to introduce more dimensions; it is then trivial to write functions that take a dimension as a parameter.

This system, at the moment, is not particularly more powerful than any other type inference system in current use. However, by combining the ideas of the principal type of an object being itself, and subtyping with constraints, it seems reasonable that this

system can be extended to something much more powerful. With other kinds of static analysis, it might be possible to infer better types (for example, finer ranges for integers) for certain expressions, and then the subtyping system presented here would infer a more precise, not necessarily atomic, type for a given expression.

The system is parameterised in three places:

1. the ground types (§7.1);
2. the choice of $\sqcup$ and $\sqcap$ for the non-structural types (§7.2); and
3. the freedom in the values of the dimensions for expressions of the form $\#.E$ and $E \mathbin{@} [E \leftarrow E]$.

Any of the above three items can be changed, and the system will still work, although often points (1) and (2) will go together for the system to make sense. In particular, it seems entirely reasonable that point (3) will be where there is the most room for improvement. With other static analyses, such as Abstract Interpretation [13], it should be possible to improve the approximations of which dimensions are used in a program.

It is envisioned that there are many possible static analyses for a TransLucid program; type inference, and in particular this type inference algorithm, is not the only way to analyse a program. Rather, it can be seen that it is useful to analyse a TransLucid program using any appropriate static analysis, which provides another piece of information about the properties of a program. The interplay between type inference and other static analysis will most likely be a back and forth process, with each analysis providing a little more information to each of the others. Then, that process can stop at either a least fixed point, or some point chosen by the programmer or configurable by the user.

We have not considered the tuple expression in isolation; the only place that the tuple can appear is to the right-hand side of an @ expression. It seems reasonable that a tree-like data structure would be represented by a tuple, although the same could be achieved with intensions, so the utility of a tuple as a data object in TransLucid is questionable. The difficulty in typing the tuple is the possibility of the aliasing of dimensions. Even in what was presented, aliasing had to be taken account of, and is solved with the restriction that dimensions be atomic values, or function parameters.

7.15 Y-combinator type

In this appendix, we present the full simplification of the type of the Y-combinator, as defined in TransLucid. The Y-combinator is defined in concrete syntax as:

$$\lambda^{\mathrm{n}} f \to \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)\ \left(\lambda^{\mathrm{n}} x \to f\ (x\ x)\right)$$

which is translated to the following abstract syntax:

$$\lambda^{\mathrm{v}} \phi_f \to \left(\lambda^{\mathrm{v}} \phi_a \to \downarrow\underline{\phi_f}\ !\ (\downarrow\underline{\phi_a}\ !\ \downarrow\underline{\phi_a})\right)\ !\ \left(\lambda^{\mathrm{v}} \phi_b \to \downarrow\underline{\phi_f}\ !\ (\downarrow\underline{\phi_b}\ !\ \downarrow\underline{\phi_b})\right)$$

The type of the Y-combinator, before any simplification, is $\emptyset \Rightarrow v_1|_G$, where G is given in Figure 7.8. In the following figures, where the constraints are too wide for the page, the line is split to the previous line for the lower bounds and to the next line for the upper bounds.

First, we will canonisation the type scheme. Canonisation is the replacement of every occurrence of $\sqcup V$ and $\sqcap V$ with another type variable, along with the appropriate additional cosntraints. The type is as before, with only the constraint graph changing, which is presented in Figures 7.9 and 7.10. We will examine a few of the type variables so that we can see what is going on.

First look at v_1 before canonisation, it has as its lower bound the type $(\sqcap\{v_2, v_{21}\} \xrightarrow{v} v_{41})$. The $\sqcap$ is replaced with v_{44}, and its corresponding variable for $\sqcup$ is v_{45}, they both require some extra constraints. In Figure 7.6, the appropriate constraints are given by

$$\gamma_V \leq_{G'} \alpha \quad \text{when} \quad \exists \beta \in V \mid \beta \leq_G \alpha$$

so we look for every variable greater than v_2 and v_{21}, and set each of them as greater than v_{44}. We have $v_2 \leq v_3$, and $v_{21} \leq v_{22}$. Therefore, we have $v_{44} \leq v_3, v_{21}$.

The variables v_3 and v_{21} have the upper bounds $\uparrow v_4$ and $\uparrow v_{23}$, respectively. So the upper bound of v_{44} is set to $(\uparrow v_4) \sqcap (\uparrow v_{23})$, which is $\uparrow \sqcap \{v_4, v_{23}\}$. Rather than introducing another $\sqcap$, the type $v_4 \sqcap v_{23}$ is replaced with new variables in the same way, along with the appropriate constraints. So the upper bound of v_{44} becomes $\uparrow v_{59}$, and the constraints on variable v_{59} are constructed in the same manner.

The same occurs for every other instance of $\sqcup V$ and $\sqcap V$ in the graph. For example, by looking at the lower bounds of v_{25} and v_{26}, we can see that every case of $\sqcup\{v_{13}, v_{32}\}$ is replaced by v_{52}. Similarly, the lower bounds of v_{31} and v_{32} indicate that $\sqcap\{v_{25}, v_{29}\}$ is replaced with v_{45}.

Next we look at garbage collecting the type scheme. Here we will run through the complete polarity computation. Initially, variable v_1 is marked as positive. So we have the sets V_0^+ and V_0^- as follows:

$$V_0^+ = \{v_1\}$$
$$V_0^- = \emptyset$$

Because v_1 is positive, we mark its lower bound, $(v_{44} \xrightarrow{v} v_{41})$ as positive, which results in v_{44} being negative and v_{41} being positive. So we now have:

$$V_1^+ = \{v_1, v_{41}\}$$
$$V_1^- = \{v_{44}\}$$

The variable v_{41} has no lower bound, so there is nothing left to mark as positive. However, as v_{44} is negative, we mark its upper bound as negative, which is $\uparrow v_{59}$. Which results in v_{59} being negative. So we now have:

$$V_2^+ = \{v_1, v_{41}\}$$
$$V_2^- = \{v_{44}, v_{59}\}$$

So we now mark the upper bound of v_{59}, which is $(v_{60} \xrightarrow{v} v_{61})$, as negative. The result is that v_{60} is positive, and v_{61} is negative. So we now have:

$$V_3^+ = \{v_1, v_{41}, v_{60}\}$$
$$V_3^- = \{v_{44}, v_{59}, v_{61}\}$$

Variable v_{60} has the lower bound $\uparrow v_{62}$, which we mark as positive, making v_{62} positive. Variable v_{61} has no upper bound, so nothing further is done. Therefore we now have:

$$V_4^+ = \{v_1, v_{41}, v_{60}, v_{62}\}$$
$$V_4^- = \{v_{44}, v_{59}, v_{61}\}$$

The variable v_{62} has no lower bound, so no further work is needed, and V_4^+ and V_4^- are our sets of positive and negative variables.

We can now remove most of the constraints from the graph. The conditions for keeping constraints are reproduced below:

1. $\alpha \leq_G \beta$ if $\alpha \in V^-$ and $\beta \in V^+$.
2. $G^{\downarrow}(\alpha)$ if $\alpha \in V^+$.
3. $G^{\uparrow}(\alpha)$ if $\alpha \in V^-$.
4. $\beta \overset{\alpha \in s}{\leq_G} \gamma$ if $\alpha \in V^-$.

By condition (1), we only need to keep $v_{61} \leq v_{41}$ and $v_{61} \leq v_{62}$. By condition (2), we keep the lower bounds of v_1 and v_{60}. By condition (3) we only keep the upper bounds of v_{44} and v_{59}. Condition (4) does not apply because there are no guarded constraints. Therefore, the resulting constraint graph is:

$$\begin{array}{rcll} (v_{44} \xrightarrow{v} v_{41}) & \leq & v_1 & \\ v_{61} & \leq & v_{41} & \\ & & v_{44} \leq \uparrow v_{59} & \\ & & v_{59} \leq (v_{60} \xrightarrow{v} v_{61}) & \\ \uparrow v_{62} & \leq & v_{60} & \\ & & v_{61} \leq v_{41}, v_{62} & \\ v_{61} & \leq & v_{62} & \end{array}$$

The next step is to minimise the constraint graph. The variables v_{41} and v_{62} are equivalent, because they are both positive, and both have the set $\{v_{61}\}$ as their less than variables. Their lower (resp. upper) bound is the same, which is $\bot$ (resp. $\top$). Therefore, we

merge them by arbitrarily choosing to keep v_{41}, and replacing v_{62} with v_{41}. The resulting graph is:

$$\begin{array}{rcll} (v_{44} \xrightarrow{v} v_{41}) & \leq & v_1 & \\ v_{61} & \leq & v_{41} & \\ & & v_{44} & \leq \uparrow v_{59} \\ & & v_{59} & \leq (v_{60} \xrightarrow{v} v_{61}) \\ \uparrow v_{41} & \leq & v_{60} & \\ & & v_{61} & \leq v_{41} \end{array}$$

This concludes the type simplification process for the type manipulated by the computer. Refer to §7.12 for the details of presenting the type in a manner readable to the user.

$$
\begin{array}{rrcll}
(\sqcap\{v_2, v_{21}\} \xrightarrow{v} v_{41}) & v_{43} & \leq & v_1 & \leq \top \\
\bot & & \leq & v_2 & \leq v_3 \quad \uparrow v_4 \\
\bot & v_2 & \leq & v_3 & \leq \uparrow v_4 \\
\bot & & \leq & v_4 & \leq v_5 \quad (v_{17} \xrightarrow{v} v_{19}) \\
\bot & v_4 & \leq & v_5 & \leq (v_{17} \xrightarrow{v} v_{19}) \\
\uparrow v_{39} & v_{40} & \leq & v_6 & \leq v_7 \quad \uparrow v_8 \\
\uparrow v_{39} & v_6, v_{40} & \leq & v_7 & \leq \uparrow v_8 \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{39} & \leq & v_8 & \leq v_9 \quad (v_{14} \xrightarrow{v} v_{16}) \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_8, v_{39} & \leq & v_9 & \leq (v_{14} \xrightarrow{v} v_{16}) \\
\uparrow v_{39} & v_{40} & \leq & v_{10} & \leq v_{11} \quad \uparrow v_{12} \\
\uparrow v_{39} & v_{10}, v_{40} & \leq & v_{11} & \leq \uparrow v_{12} \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{39} & \leq & v_{12} & \leq v_{13}, v_{27}, v_{28}, v_{31}, v_{32} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{12}, v_{39} & \leq & v_{13} & \leq v_{27}, v_{28}, v_{31}, v_{32} \quad (v_{33} \xrightarrow{v} v_{35}) \\
\uparrow v_{13} & & \leq & v_{14} & \leq v_{25}, v_{26}, v_{29}, v_{30} \quad \uparrow \sqcap \{v_{27}, v_{31}\} \\
\bot & v_{16}, v_{37}, v_{38} & \leq & v_{15} & \leq \top \\
\bot & v_{37}, v_{38} & \leq & v_{16} & \leq v_{15} \quad \top \\
\uparrow v_{15} & & \leq & v_{17} & \leq \top \\
\bot & v_{19} & \leq & v_{18} & \leq v_{41}, v_{42} \quad \top \\
\bot & & \leq & v_{19} & \leq v_{18}, v_{41}, v_{42} \quad \top \\
(\sqcap\{v_6, v_{10}\} \xrightarrow{v} v_{18}) & & \leq & v_{20} & \leq (v_{40} \xrightarrow{v} v_{42}) \\
\bot & & \leq & v_{21} & \leq v_{22} \quad \uparrow v_{23} \\
\bot & v_{21} & \leq & v_{22} & \leq \uparrow v_{23} \\
\bot & & \leq & v_{23} & \leq v_{24} \quad (v_{36} \xrightarrow{v} v_{38}) \\
\bot & v_{23} & \leq & v_{24} & \leq (v_{36} \xrightarrow{v} v_{38}) \\
\uparrow \sqcup \{v_{13}, v_{32}\} & v_{14}, v_{33} & \leq & v_{25} & \leq v_{26} \quad \uparrow v_{27} \\
\uparrow \sqcup \{v_{13}, v_{32}\} & v_{14}, v_{25}, v_{33} & \leq & v_{26} & \leq \uparrow v_{27} \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{12}, v_{13}, v_{31}, v_{32}, v_{39} & \leq & v_{27} & \leq v_{28} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & & & & \\
v_{12}, v_{13}, v_{27}, v_{31}, v_{32}, v_{39} & & \leq & v_{28} & \leq (v_{33} \xrightarrow{v} v_{35}) \\
\uparrow \sqcup \{v_{13}, v_{32}\} & v_{14}, v_{33} & \leq & v_{29} & \leq v_{30} \quad \uparrow v_{31} \\
\uparrow \sqcup \{v_{13}, v_{32}\} & v_{14}, v_{29}, v_{33} & \leq & v_{30} & \leq \uparrow v_{31} \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{12}, v_{13}, v_{32}, v_{39} & \leq & v_{31} & \leq v_{27}, v_{28}, v_{32} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & v_{12}, v_{13}, v_{31}, v_{39} & \leq & v_{32} & \leq v_{27}, v_{28}, v_{31} \quad (v_{33} \xrightarrow{v} v_{35}) \\
\uparrow v_{32} & & \leq & v_{33} & \leq v_{25}, v_{26}, v_{29}, v_{30} \quad \uparrow \sqcap \{v_{27}, v_{31}\} \\
\bot & v_{35}, v_{37}, v_{38} & \leq & v_{34} & \leq \top \\
\bot & v_{37}, v_{38} & \leq & v_{35} & \leq v_{34} \quad \top \\
\uparrow v_{34} & & \leq & v_{36} & \leq \top \\
\bot & v_{38} & \leq & v_{37} & \leq v_{15}, v_{16}, v_{34}, v_{35} \quad \top \\
\bot & & \leq & v_{38} & \leq v_{15}, v_{16}, v_{34}, v_{35}, v_{37} \quad \top \\
(\sqcap\{v_{25}, v_{29}\} \xrightarrow{v} v_{37}) & & \leq & v_{39} & \leq v_8, v_9, v_{12}, v_{13}, v_{27}, v_{28}, v_{31}, v_{32} \\
 & & & & (\sqcup\{v_{14}, v_{33}\} \xrightarrow{v} \sqcap\{v_{16}, v_{35}\}) \\
\uparrow v_{39} & & \leq & v_{40} & \leq v_6, v_7, v_{10}, v_{11} \quad \uparrow \sqcap \{v_8, v_{12}\} \\
\bot & v_{18}, v_{19}, v_{42} & \leq & v_{41} & \leq \top \\
\bot & v_{18}, v_{19} & \leq & v_{42} & \leq v_{41} \quad \top \\
(\sqcap\{v_2, v_{21}\} \xrightarrow{v} v_{41}) & & \leq & v_{43} & \leq v_1 \quad \top
\end{array}
$$

Figure 7.8: Y-combinator constraint graph before simplification

$$
\begin{array}{rrcll}
(v_{44} \xrightarrow{v} v_{41}) & v_{43} & \leq & v_1 & \leq \top \\
 & \bot & \leq & v_2 & \leq v_3 \quad \uparrow v_4 \\
\bot & v_2, v_{44} & \leq & v_3 & \leq \uparrow v_4 \\
 & \bot & \leq & v_4 & \leq v_5 \quad (v_{17} \xrightarrow{v} v_{19}) \\
\bot & v_4, v_{59} & \leq & v_5 & \leq (v_{17} \xrightarrow{v} v_{19}) \\
 & \uparrow v_{39} & v_{40} \leq & v_6 & \leq v_7 \quad \uparrow v_8 \\
\uparrow v_{39} & v_6, v_{40}, v_{49} & \leq & v_7 & \leq \uparrow v_8 \\
(v_{45} \xrightarrow{v} v_{37}) & v_{39} & \leq & v_8 & \leq v_9 \quad (v_{14} \xrightarrow{v} v_{16}) \\
(v_{45} \xrightarrow{v} v_{37}) & v_8, v_{39}, v_{57} & \leq & v_9 & \leq (v_{14} \xrightarrow{v} v_{16}) \\
 & \uparrow v_{39} & v_{40} \leq & v_{10} & \leq v_{11} \quad \uparrow v_{12} \\
\uparrow v_{39} & v_{10}, v_{40}, v_{49} & \leq & v_{11} & \leq \uparrow v_{12} \\
(v_{45} \xrightarrow{v} v_{37}) & v_{39} & \leq & v_{12} & \leq v_{13}, v_{27}, v_{28}, v_{31}, v_{32}, v_{48}, v_{52} \\
 & & & & \quad (v_{33} \xrightarrow{v} v_{35}) \\
(v_{45} \xrightarrow{v} v_{37}) & v_{12}, v_{39}, v_{57} & \leq & v_{13} & \leq v_{27}, v_{28}, v_{31}, v_{32}, v_{48}, v_{52} \quad (v_{33} \xrightarrow{v} v_{35}) \\
 & \uparrow v_{13} & \leq & v_{14} & \leq v_{25}, v_{26}, v_{29}, v_{30}, v_{46} \quad \uparrow v_{47} \\
\bot & v_{16}, v_{37}, v_{38}, v_{55}, v_{61} & \leq & v_{15} & \leq \top \\
\bot & v_{37}, v_{38}, v_{61} & \leq & v_{16} & \leq v_{15}, v_{62} \quad \top \\
 & \uparrow v_{15} & \leq & v_{17} & \leq \top \\
\bot & v_{19}, v_{61} & \leq & v_{18} & \leq v_{41}, v_{42} \quad \top \\
 & \bot & \leq & v_{19} & \leq v_{18}, v_{41}, v_{42} \quad \top \\
 & (v_{49} \xrightarrow{v} v_{18}) & \leq & v_{20} & \leq (v_{40} \xrightarrow{v} v_{42}) \\
 & \bot & \leq & v_{21} & \leq v_{22} \quad \uparrow v_{23} \\
\bot & v_{21}, v_{44} & \leq & v_{22} & \leq \uparrow v_{23} \\
 & \bot & \leq & v_{23} & \leq v_{24} \quad (v_{36} \xrightarrow{v} v_{38}) \\
\bot & v_{23}, v_{59} & \leq & v_{24} & \leq (v_{36} \xrightarrow{v} v_{38}) \\
\uparrow v_{52} & v_{14}, v_{33}, v_{53} & \leq & v_{25} & \leq v_{26} \quad \uparrow v_{27} \\
\uparrow v_{52} & v_{14}, v_{25}, v_{33}, v_{45}, v_{53} & \leq & v_{26} & \leq \uparrow v_{27} \\
 & (v_{45} \xrightarrow{v} v_{37}) & & & \\
v_{12}, v_{13}, v_{31}, v_{32}, v_{39}, v_{47}, v_{51}, v_{57} & & \leq & v_{27} & \leq v_{28} \quad (v_{33} \xrightarrow{v} v_{35}) \\
 & (v_{45} \xrightarrow{v} v_{37}) & & & \\
v_{12}, v_{13}, v_{27}, v_{31}, v_{32}, v_{39}, v_{47}, v_{51}, v_{57} & & \leq & v_{28} & \leq (v_{33} \xrightarrow{v} v_{35}) \\
\uparrow v_{52} & v_{14}, v_{33}, v_{53} & \leq & v_{29} & \leq v_{30} \quad \uparrow v_{31} \\
\uparrow v_{52} & v_{14}, v_{29}, v_{33}, v_{45}, v_{53} & \leq & v_{30} & \leq \uparrow v_{31} \\
(v_{45} \xrightarrow{v} v_{37}) & v_{12}, v_{13}, v_{32}, v_{39}, v_{47}, v_{51}, v_{57} & \leq & v_{31} & \leq v_{27}, v_{28}, v_{32}, v_{48}, v_{52} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(v_{45} \xrightarrow{v} v_{37}) & v_{12}, v_{13}, v_{31}, v_{39}, v_{47}, v_{51}, v_{57} & \leq & v_{32} & \leq v_{27}, v_{28}, v_{31}, v_{48}, v_{52} \quad (v_{33} \xrightarrow{v} v_{35}) \\
 & \uparrow v_{32} & \leq & v_{33} & \leq v_{25}, v_{26}, v_{29}, v_{30}, v_{46} \quad \uparrow v_{47} \\
\bot & v_{35}, v_{37}, v_{38}, v_{55}, v_{61} & \leq & v_{34} & \leq \top \\
\bot & v_{37}, v_{38}, v_{61} & \leq & v_{35} & \leq v_{34}, v_{62} \quad \top \\
 & \uparrow v_{34} & \leq & v_{36} & \leq \top \\
\bot & v_{38}, v_{61} & \leq & v_{37} & \leq v_{15}, v_{16}, v_{34}, v_{35}, v_{56}, v_{62} \quad \top \\
 & \bot & \leq & v_{38} & \leq v_{15}, v_{16}, v_{34}, v_{35}, v_{37}, v_{56}, v_{62} \quad \top \\
 & (v_{45} \xrightarrow{v} v_{37}) & \leq & v_{39} & \leq v_8, v_9, v_{12}, v_{13}, v_{27}, v_{28}, v_{31}, v_{32} \\
 & & & & \quad v_{48}, v_{52}, v_{58} \quad (v_{54} \xrightarrow{v} v_{55}) \\
 & \uparrow v_{39} & \leq & v_{40} & \leq v_6, v_7, v_{10}, v_{11}, v_{50} \quad \uparrow v_{57} \\
\bot & v_{18}, v_{19}, v_{42}, v_{61} & \leq & v_{41} & \leq \top \\
\bot & v_{18}, v_{19}, v_{61} & \leq & v_{42} & \leq v_{41} \quad \top \\
 & (v_{44} \xrightarrow{v} v_{41}) & \leq & v_{43} & \leq v_1 \quad \top
\end{array}
$$

Figure 7.9: Y-combinator constraint graph after canonisation (1)

$$
\begin{array}{rcccl}
\bot & \leq & v_{44} & \leq & v_3, v_{22} \quad \uparrow v_{59} \\
\bot & \leq & v_{45} & \leq & v_{26}, v_{30} \quad \uparrow v_{47} \\
\uparrow v_{52} \quad v_{14}, v_{33}, v_{53} & \leq & v_{46} & \leq & \top \\
\bot & \leq & v_{47} & \leq & v_{27}, v_{28}, v_{31}, v_{32}, v_{48}, v_{52} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(v_{45} \xrightarrow{v} v_{37}) & & & & \\
v_{12}, v_{13}, v_{31}, v_{32}, v_{39}, v_{47}, v_{51}, v_{57} & \leq & v_{48} & \leq & \top \\
\bot & \leq & v_{49} & \leq & v_7, v_{11} \quad \uparrow v_{57} \\
\uparrow v_{39} \quad v_{40} & \leq & v_{50} & \leq & \top \\
\bot & \leq & v_{51} & \leq & v_{27}, v_{28}, v_{31}, v_{32}, v_{48}, v_{52} \quad (v_{33} \xrightarrow{v} v_{35}) \\
(v_{45} \xrightarrow{v} v_{37}) & & & & \\
v_{12}, v_{13}, v_{31}, v_{32}, v_{39}, v_{47}, v_{51}, v_{57} & \leq & v_{52} & \leq & \top \\
\bot & \leq & v_{53} & \leq & v_{25}, v_{26}, v_{29}, v_{30}, v_{46} \quad \uparrow v_{47} \\
\uparrow v_{52} & \leq & v_{54} & \leq & \top \\
\bot & \leq & v_{55} & \leq & v_{15}, v_{34}, v_{62} \quad \top \\
\bot \quad v_{37}, v_{38}, v_{61} & \leq & v_{56} & \leq & \top \\
\bot & \leq & v_{57} & \leq & v_9, v_{13}, v_{27}, v_{28}, v_{31}, v_{32}, v_{48}, v_{52} \\
 & & & & (v_{54} \xrightarrow{v} v_{55}) \\
(v_{45} \xrightarrow{v} v_{37}) \quad v_{39} & \leq & v_{58} & \leq & \top \\
\bot & \leq & v_{59} & \leq & v_5, v_{24} \quad (v_{60} \xrightarrow{v} v_{61}) \\
\uparrow v_{62} & \leq & v_{60} & \leq & \top \\
\bot & \leq & v_{61} & \leq & v_{15}, v_{16}, v_{18}, v_{34}, v_{35}, v_{37}, v_{41}, v_{42}, v_{56}, v_{62} \quad \top \\
\bot \quad v_{16}, v_{35}, v_{37}, v_{38}, v_{55}, v_{61} & \leq & v_{62} & \leq & \top
\end{array}
$$

Figure 7.10: Y-combinator constraint graph after canonisation (2)

Chapter 8

The TransLucid/C++ System, or Concrete TransLucid

This chapter shows how we move from TransLucid, with its abstract syntax and denotational semantics, to a full, concrete language, with concrete syntax and concrete data types of another (host) language. To undertake this task, we have two choices: 1) to fix the set of atomic values, or 2) to provide the means to access the atomic values (which is not a fixed set) of another language. We choose the second path, and use as the host language `C++11`, along with the GNU `mp` integer and floating-point libraries, and IBM's `icu` for Unicode support.

The denotational semantic rules—presented in Chapter 2—are of the form $[\![E]\!]\iota\zeta\kappa$, where E is an expression, ι is the interpretation of the constants, ζ is an environment, and κ is a context. Implicit in all of this is the set D of atomic objects, which is the basis for the semantic domains, with $\iota({}^0c) \in D$ and $\iota({}^mc) \in (D^m \to D), m > 0$.

When we move from TransLucid to Concrete TransLucid, we need, of course, to define a concrete syntax (§8.1) for expressions E. But, more importantly, the host environment provided by `C++11` and the core libraries in the concrete language, corresponds to the pair (D, ι) of Core TransLucid. In this chapter, we make everything concrete: concrete atomic types to define D, user-defined constructible types, concrete syntax for E, including the means to add new unary or binary operators, and the full use of Unicode characters, amongst other things.

The host environment provides a set of atomic types, whose union forms the set D (§8.2). We assume at the very least that Booleans, integers, and Unicode characters and strings are provided. As for the ι, it corresponds to the union of the parsers for each of the atomic types. Since we are dealing with an interpreter, we also need to define what is effectively ι^{-1}, which corresponds to the union of the printers for each of the atomic types.

We start with the syntax for Concrete TransLucid (§8.1), without explaining how certain constructs are recognised by the parser, then present the infrastructure for interacting with the host environment. Then we can focus on the concrete syntax for variable declarations (§8.5), for function declarations (§8.6) and the user-defined data types (§8.7), all of which require the use of a new type of conditional expression, called *bestfitting* (§8.4),

in which the branch to be chosen is the best choice depending on the current context. We then present operator declarations (§8.8), which, using the host environment, enable prefix, postfix and infix notation, and inform the parser how to recognise symbols presented in §8.1. In addition, all of this infrastructure can be extended with *libraries* (§8.9), which add new objects to D, and functions to manipulate them.

Finally, we present the TransLucid system (§8.10), which takes all of the declarations and infrastructure presented in this chapter, and produces a programming environment that allows the user to add declarations and evaluates expressions. Then we extend that system to evaluate expressions through time (§8.11), taking new declarations and expressions to evaluate at each *instant*, taking real data as input, and producing real data as output at each instant (§8.12).

8.1 Concrete TransLucid

Concrete TransLucid is a set of *declarations*, and a set of concrete expressions to be evaluated. The remainder of this chapter presents the recognised declarations, with their syntax, and the syntax of expressions. In this section, we focus on the syntax.

8.1.1 Expressions

Concrete TransLucid expressions can appear standalone or within declarations. Each expression is a sequence of *lexemes*, which is transformed by a Concrete TransLucid parser to a TransLucid expression. There are four kinds of lexemes:

- *punctuation symbols*, which define TransLucid constructs, and correspond closely to the symbols from the abstract syntax;
- *literals*, which are the lexemes that are translated by ι to elements of D;
- *identifiers*, which are unaltered in the abstract syntax; and
- *operator symbols*, which are translated to function applications.

Each of these lexemes is transformed by the lexical analyser into a tuple which describes the meaning of the token so that it can be used by the parser to build an expression.

8.1.2 Punctuation

The TransLucid punctuation is as follows:

- declaration punctuation:

  ```
  |      =      :=      ;;
  ```

- tuple manipulation:

  ```
  [      :      <-      ]
  ```

- context manipulation:

  `#` `@`

- intension manipulation:

  ↑ ↓ `{` `}`

- functional abstraction:

  `\` `\\` `->`

- functional application:

  `!` `.`

- general grouping:

  `(` `,` `)`

  Each punctuation symbol s is transformed into a tuple of the form:

$$[kind \leftarrow punctuation, symbol \leftarrow s]$$

8.1.3 Literals

The syntax of the literals recognised by the lexical analyser is presented in the following sections.

Boolean literals

The Boolean literals are of type `bool`, and consist of the two constants `true` and `false`. A Boolean symbol b is transformed by the lexical analyser to the following tuple:

$$[kind \leftarrow literal, type \leftarrow bool, value \leftarrow b]$$

Integer literals

The integer literals are of type `intmp`, implemented using GNU `mp` integers. An integer symbol n is transformed by the lexical analyser to the following tuple:

$$[kind \leftarrow literal, type \leftarrow intmp, value \leftarrow n]$$

Their syntax is outlined below.

- Negative integers are an integer literal preceded by character `~`.
- Any integer starting with characters `1` through `9` is interpreted as base 10.
- The character `0` by itself corresponds to the value 0.
- An integer beginning with `01` followed by n more `1`s is base-1 notation for the number n.

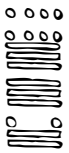
- An integer beginning `0` followed by a character in the range `[2-9A-Za-z]` uses that second character as base-designator as follows:
  - `2` through `9` mean bases 2 through 9, respectively;
  - `A` through `Z` mean bases 10 through 35, respectively;
  - `a` through `z` mean bases 36 through 61, respectively.

  The subsequent characters are interpreted as digits in that base. For a number in base n, only 'digits' from 0 to $n-1$ may be used.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
W	X	Y	Z	a	b	c	d	e	f	g	h	i	j	k	l
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
m	n	o	p	q	r	s	t	u	v	w	x	y	z		
48	49	50	51	52	53	54	55	56	57	58	59	60	61		

For example, the number 39912 becomes

- `021001101111101000` in binary (base 2);
- `08115750` in octal (base 8);
- `0A39912` in decimal (base 10);
- `0G9BE8` in hexadecimal (base 16);
- `0K4JFC` in vigesimal (base 20), as used by the Mayans:



- `0yB5C` in sexagesimal (base 60), as used by the Babylonians:

Character literals

A character literal is implemented using a Unicode 32-bit character literal (`UCS-4`, `http://www.unicode.org`), and is written as a single *cooked character* surrounded by single quotes (`'c'`). A character literal c is transformed by the lexical analyser to the following tuple:

$$[kind \leftarrow literal, type \leftarrow uchar, value \leftarrow c]$$

The cooked character c is either a single character, or an escape sequence. The valid escape sequences are:

`\n` for a newline (`000A`);

`\r` for a carriage return (`000D`);

`\t` for a horizontal tab (`0009`);

`\'` for a single quote (`0027`);

`\"` for a double quote (`0022`);

`\\` for a backslash (`005C`);

`\uXXXX` where `XXXX` are four hex digits, for a Unicode character in the Basic Multilingual Plane, range `0000`–`FFFF`;

`\UXXXXXXXX` where `XXXXXXXX` are eight hex digits, for a Unicode character not in the Basic Multilingual Plane, range `10000`–`10FFFF`;

`\xXX` for a valid one-byte UTF-8 sequence, designating a Unicode character in the range `0000`–`007F`;

`\xXX\xXX` for a valid two-byte UTF-8 sequence, designating a Unicode character in the range `0080`–`07FF`;

`\xXX\xXX\xXX` for a valid three-byte UTF-8 sequence, designating a Unicode character in the range `0800`–`FFFF`;

`\xXX\xXX\xXX\xXX` for a valid four-byte UTF-8 sequence, designating a Unicode character in the range `10000`–`10FFFF`.

String literals

String literals are a sequence of n cooked characters, described in the previous paragraph, surrounded by double quotes (`"`$c_0 \cdots c_{n-1}$`"`), or a raw string literal, which is a sequence of bytes interpreted as UTF-8 placed between back quotes (‘$c_0 \cdots c_{n-1}$‘). A string literal s is transformed by the lexical analyser to the following tuple:

$$[kind \leftarrow literal, type \leftarrow ustring, value \leftarrow s]$$

Generic literals

A generic literal is a single lexeme made up of an identifier T followed by a string literal, either cooked or raw. Therefore, its syntax is either T`"`s`"` or T‘s‘. It is translated by the lexical analyser to the tuple

$$[kind \leftarrow literal, type \leftarrow generic, typeid \leftarrow T, value \leftarrow s]$$

8.1.4 Special values

In the concrete language, we have chosen that there be no undefined computations. The semantics leaves certain things undefined, such as the meaning of $\kappa(\delta)$ when $\delta \notin \mathrm{dom}(\kappa)$, for some context κ and some dimension δ. For these "error" cases, we return an object from the **special** type, rather than the other option of crashing (gracefully or otherwise), or worse, continuing in some erroneous state. The special values are all written `sp`*value* for some special value describing the error *value*. The special values and their uses are summarised in the following table:

Special value	Description
`sptypeerror`	Function application error
`spundef`	Undefined identifier ($x \notin \mathrm{dom}(\zeta)$)
`spdim`	Undefined dimension ($d \notin \mathrm{dom}(\kappa)$)
`spmultidef`	Multiple definitions (§8.4)
`spaccess`	Access error (§8.11)
`sploop`	Loop in cache (§6.6)

8.1.5 Identifiers

An identifier (x) in TransLucid is of the following form:

$$x ::= (\ \textit{Letter} \mid \_\ )\ (\ \textit{Letter} \mid \textit{Number} \mid \_\ )^{+}$$

where *Letter* stands for the entire Unicode class `Letter`, which is any kind of "letter" from any script (including most Chinese characters), and *Number* stands for the Unicode class `Number`, which is any kind of numeric character in any script. Examples of uses of unusual characters in identifiers are $\mathtt{H_2O}$ (water) and $\mathtt{\_^3_2He}$ (helium-3).

The following TransLucid keywords are reserved.

- declaration introductions:

  `data` `dim` `fun` `hd`
  `op` `var` `assign` `host`

- conditional expressions:

  `if` `then` `elsif` `else` `fi`

- local declarations:

  `where` `end`

- Boolean values:

  `true` `false`

- bestfitting keywords (§8.4):

  `is` `imp` `bestof`

The declaration-introduction symbols x are transformed to the following tuple:

$$[kind \leftarrow declaration, value \leftarrow x]$$

The other keywords x are transformed to the following tuple:

$$[kind \leftarrow keyword, value \leftarrow x]$$

8.1.6 Operator symbols

An operator (op) in TransLucid has the following form:

$$op ::= (\ Symbol\ |\ \texttt{!}\ |\ \texttt{\%}\ |\ \texttt{*}\ |\ \texttt{-}\ |\ \texttt{.}\ |\ \texttt{\&}\ |\ \texttt{/}\ |\ \texttt{:}\ )^{+}$$

where *Symbol* stands for the Unicode character class `Symbol`. The six exceptions are the symbols that are TransLucid punctuation:

`=` `:` `|` `!` `.` `//`

Operator symbols can be used to improve the clarity of programs, rather than using verbose prefix function-call notation for every expression. An operator can be declared by the user to be unary postfix, unary prefix, or binary infix, and using those declarations the resulting expression is transformed by the parser to the appropriate function-call expression. The mechanism for declaring these and their transformation by the lexical analyser to a tuple is explained in §8.8. Nevertheless, we can present the way in which the symbols will be used without presenting how they get there. Each of the operators is mapped to a function name, and can stand for a call-by-name or call-by-value function application. The binary infix operators also have an associativity and precedence. A unary postfix operator is translated to the following tuple:

$$[kind \leftarrow operator, type \leftarrow postfix, function \leftarrow id, call\_type \leftarrow \texttt{cbn} \mid \texttt{cbv}]$$

A unary prefix operator is translated to the following tuple:

$$[kind \leftarrow operator, type \leftarrow prefix, function \leftarrow id, call\_type \leftarrow \texttt{cbn} \mid \texttt{cbv}]$$

A binary infix operator is translated to the tuple:

$$\begin{aligned}[&kind \leftarrow operator,\\ &type \leftarrow infix,\\ &function \leftarrow id,\\ &call\_type \leftarrow \texttt{cbn} \mid \texttt{cbv},\\ &precedence \leftarrow intmp,\\ &assoc \leftarrow \texttt{AssocNon} \mid \texttt{AssocRight} \mid \texttt{AssocLeft}]\end{aligned}$$

The precedence of unary postfix operators is highest, then unary prefix, then binary infix.

8.2 Types and the host system

In any programming language, there is a notion of the *type* of an object, i.e., it is an integer, a string, and so on. Then, the operations over objects are restricted by the type of those objects, since some operations do not make sense for certain types of objects. For example, usually, it makes no sense to add a string and an integer, or at least if it does make sense, it is still only by converting the integer to a string and concatenating the result. For the types in TransLucid, we take a slightly different approach to the norm. The intuition behind data types in TransLucid comes from Shamir and Wadge's 1977 *ICALP* paper, "Data Types as Objects" [40], in which types are both sets of values and objects themselves, as are all of the subsets of these types perceived as *desirable*, *interesting* or *relevant* by the users of a system.

Despite considering any object or set of objects as a type, there is still a practical consideration here. Every object manipulated by the system is a set of bits that are understood to represent that object, and are interpreted in a manner suitable for that type of object. Therefore, regardless of how we view types, it must still be consistent with each object having a physical type.

As presented in §7.1, we suppose that the atomic objects manipulated by the system are made up of distinct sets of object, such as strings, integers and so on, each of which is given a name. Each of those sets we call a *type*. In addition, so that we can have data types as objects, we require that each type itself also be an object manipulable by the system. The type objects are then members of the **type** type, including **type** itself. In addition, the concrete Translucid system manipulates ranges over integers, so we require objects representing negative infinity and positive infinity, for the ends of unbounded ranges of integers, along with a type for ranges of integers.

In fact, when mapping the concrete system to the semantics presented in the chapters up to now, the set of objects just mentioned is actually the set D used by the semantics, from §2.2.2, Chapter 2 on. As a result, the set D of atomic objects is formed as the union of

1. the set $\{\mathbf{type}, \mathbf{range}\} \cup \{t_1, t_2, \ldots, t_n\}$, where $t_1, t_2, \ldots, t_n$ are the concrete types in the system; this list must include **bool**, **intmp**, **uchar**, **ustring** and **special**;
2. $D_{t_1} \cup D_{t_2} \cup \ldots \cup D_{t_n}$, where each D_t is the set of elements corresponding to type t;
3. $\{-\infty, \infty\}$, special elements used for designating the ends of unbounded ranges of integers; and
4. $\{\emptyset\} \cup \{(-\infty, n]\} \cup \{[m, n] \mid m \leq n\} \cup \{[m, \infty)\} \cup \{(-\infty, \infty)\}$, where $m, n \in \mathbb{Z}$, which are the possible ranges of integers.

Because we are working with an interpreter, it is necessary for there to be a mechanism to both read values as input, and write values as output. Therefore, each concrete type in the system must have both a constructor and a printer. The constructors are functions of the form:

$$\texttt{construct\_}t : \mathbf{ustring} \to t$$

where $t \in \{\mathbf{type}, \mathbf{range}, \mathbf{special}, t_1 \ldots, t_n\}$. As for the printers, they are functions of the form:

$$\texttt{print\_}t : t \to \mathbf{ustring}$$

The mechanism for adding these functions is presented in §8.6.

8.3 Host functions

A Concrete TransLucid system provides a declaration for informing the system about a function that is accessed through a C-style function pointer, with the purpose of carrying out low-level operations on data types provided in the system. A declaration informs the system of the existence of a function x, its arity m, and its *address* in memory. The function residing at that address should take as argument m TransLucid atomic objects, and return an atomic object. Functions are declared using the `hostfun` declaration, whose syntax is as follows:

$$\texttt{hostfun}\ x\ m\ \mathit{address}\ \texttt{;;}$$

where m and *address* are non-negative integers. In the case $m = 0$, this means a constant function, in the case *address* $= 0$, this is usually a runtime error, typically a segmentation fault. Note that normally the user will not use this declaration, since the address of a function is not usually known, and might change depending on the dynamic linker. Rather, it is used internally to declare the built-in functions, and can be used by libraries (§8.9) which can add to the set of types and functions in the system.

These functions are in fact defining the concrete C/C++ implementation of the interpretation ι of constant symbols (§2.2.3). These functions, along with the black-box lexical analyser and parser, are the basis for the C/C++ interface to the TransLucid system.

Although there is only a small number of functions that are required by the system, to produce a complete system, it is necessary to provide all of the standard functionality for each of the types. The functions presented in the following sections map one or more atomic objects to atomic objects.

8.3.1 Integer functions

The integer functions all operate over the GNU `mp` integers, which have the type **intmp**, and include the standard arithmetic and comparison functions.

$$\begin{aligned}
\mathit{intmp_plus} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{intmp} \\
\mathit{intmp_minus} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{intmp} \\
\mathit{intmp_times} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{intmp} \\
\mathit{intmp_divide} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{intmp} \\
\mathit{intmp_modulus} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{intmp} \\
\mathit{intmp_lte} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool} \\
\mathit{intmp_lt} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool}
\end{aligned}$$

$$\begin{aligned}
\mathit{intmp_gte} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool} \\
\mathit{intmp_gt} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool} \\
\mathit{intmp_eq} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool} \\
\mathit{intmp_ne} &: (\mathbf{intmp}, \mathbf{intmp}) \to \mathbf{bool} \\
\mathit{intmp_uminus} &: \mathbf{intmp} \to \mathbf{intmp}
\end{aligned}$$

8.3.2 Floating-point functions

The integer functions all operate over the GNU `mp` floating-point numbers, which have the type **floatmp**, and include the standard arithmetic and comparison functions.

$$\begin{aligned}
\mathit{floatmp_plus} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{floatmp} \\
\mathit{floatmp_minus} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{floatmp} \\
\mathit{floatmp_times} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{floatmp} \\
\mathit{floatmp_divide} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{floatmp} \\
\mathit{floatmp_modulus} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{floatmp} \\
\mathit{floatmp_lte} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_lt} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_gte} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_gt} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_eq} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_ne} &: (\mathbf{floatmp}, \mathbf{floatmp}) \to \mathbf{bool} \\
\mathit{floatmp_sqrt} &: \mathbf{floatmp} \to \mathbf{floatmp} \\
\mathit{floatmp_abs} &: \mathbf{floatmp} \to \mathbf{floatmp} \\
\mathit{floatmp_uminus} &: \mathbf{floatmp} \to \mathbf{floatmp} \\
\mathit{floatmp_convert_intmp} &: \mathbf{floatmp} \to \mathbf{intmp}
\end{aligned}$$

8.3.3 Boolean functions

The only Boolean function required is equality.

$$\mathit{bool\_eq} : (\mathbf{bool}, \mathbf{bool}) \to \mathbf{bool}$$

8.3.4 Character functions

For the Unicode character type, we provide a number of functions implemented by IBM's `icu` library.

$$\begin{aligned}
\mathit{is_printable} &: \mathbf{uchar} \to \mathbf{bool} \\
\mathit{code_point} &: \mathbf{uchar} \to \mathbf{intmp}
\end{aligned}$$

$$\textit{code\_point\_4} : \textbf{uchar} \to \textbf{ustring}$$
$$\textit{code\_point\_8} : \textbf{uchar} \to \textbf{ustring}$$

8.3.5 String functions

For the moment, we provide a restricted set of functions for strings:

$$\textit{ustring\_concatenate} : (\textbf{ustring}, \textbf{ustring}) \to \textbf{ustring}$$
$$\textit{ustring\_substr} : (\textbf{ustring}, \textbf{intmp}, \textbf{intmp}) \to \textbf{ustring}$$

8.3.6 Range functions

The range functions allow the user to create a range that is either bounded or unbounded in either direction. The range unbounded in both directions, *make_range_infinite*, which is equivalent to the type **intmp**, is in fact a constant.

$$\textit{make\_range} : (\textbf{intmp}, \textbf{intmp}) \to \textbf{range}$$
$$\textit{make\_range\_infty} : \textbf{intmp} \to \textbf{range}$$
$$\textit{make\_range\_neginfty} : \textbf{intmp} \to \textbf{range}$$
$$\textit{make\_range\_infinite} : \textbf{range}$$

8.3.7 Concluding remarks

The functions above provide the standard operations for each data type. However, they would not normally be used directly by the user, as the syntax becomes unwieldy. For convenience, §8.8 describes how the user can declare unary postfix, unary prefix and binary infix operator symbols, so that these can be translated to function calls using the above functions.

8.4 Bestfitting

This section presents a new type of conditional expression called *bestfitting*, never previously added to any Lucid-like language. Bestfitting allows an expression E to be defined as a choice from among a set of subexpressions, where each of these subexpressions is *guarded* by a context region, which is a set of contexts. When expression E is to be evaluated in a particular context κ, the subexpression to be evaluated depends on which context region the current context κ happens to be inside. Should κ turn out to be more inside more than one of these context regions, then the most specific, or *bestfit*, region is chosen.

The ideas behind bestfitting were first presented in the seminal 1993 *IEEE TOSE* article "A New Approach to Version Control" [30] by Plaice and Wadge, which presented bestfitting as a means to select a different version of a piece of software, or the components of a piece of software, based on some context. Their ideas were developed over the following fifteen years, the results of which are summarised in the 2008 *Mathematics in Computer Science* paper "Possible Worlds Versioning" [25] by Mancilla and Plaice. Uses of bestfitting

included intensional HTML [45], a context-aware sequential programming language [41], and an intensional mapping server [24].

Bestfitting was used to guard a definition with a context, so that only the definitions applicable to the current context would be chosen. Then, out of those applicable, the *bestfit* definition would be chosen. Previously (in previous work), the definitions of an object were restricted to being guarded by a single context. Then, the bestfit definition was the one whose context was *more refined* than those of the other definitions. Context κ refined context κ' if κ defined *at least* the dimensions of κ', and the ordinates of their corresponding dimensions were the same.

Here, we extend the idea of bestfitting so that definitions can be guarded by *sets* of contexts, rather than just a single context. A definition is considered valid if the current context is inside the guard for that definition. The bestfit definition out of several is still the one whose guard is more refined than the others, but we extend the definition of refinement so that a region k is more refined than another region k' if k defines at least the dimensions of k', and each ordinate of k for a corresponding dimension is a subset of the corresponding ordinate of k'.

Determining both which region a context is inside and which region out of several is the most refined is, in general, undecidable. Therefore, we must restrict the way in which regions can be specified so that bestfitting is still decidable. As a result, we present bestfitting in two parts: the first is the semantics, which is in general, undecidable, and the second is some practical restrictions to the sets allowed so that bestfitting can be decidable.

We begin by adding to the abstract syntax for the denotational semantics (Figure 2.1):

$$\texttt{bestof}\ E_{i0}\ \texttt{|}\ E_{i1} \to E_{i2}\ \texttt{end}$$

For example, consider the following possible definition of the factorial function (the new syntax ":" and `is` are explained below):

$$\begin{array}{l}
\lambda^{\mathrm{v}} n \to F \\
\texttt{where} \\
\quad \texttt{dim}\ d \leftarrow n \\
\quad \texttt{var}\ F = \texttt{bestof} \\
\qquad\qquad\qquad [d\ \texttt{is}\ 0] \to 1 \\
\qquad\qquad\qquad [d : 1..\texttt{infty}] \to \texttt{\#}.d \times \big(F\ \texttt{@}\ [d \leftarrow \texttt{\#}.d - 1]\big) \\
\qquad\qquad \texttt{end} \\
\texttt{end}
\end{array}$$

First we allocate a new dimension d, and initialise its ordinate to the parameter n. We can then create an array F that varies in dimension d, and whose entries are the factorial of the index of the entry. The first choice in the `bestof` says that when the ordinate of dimension d is the value 0, the value of the expression is 1. The second choice states that when the ordinate is an integer in the range 1..`infty`, the value of the expression is the

index of the ordinate multiplied by the array F at the previous entry.

The denotational semantics for bestfitting would then be as follows:

$$
\begin{aligned}
&[\![\texttt{bestof } E_{i0} \texttt{ | } E_{i1} = E_{i2} \texttt{ end}_{i=1..m}]\!]\iota\zeta\kappa = \\
&\quad \text{let } k_i = [\![E_{i0}]\!]\iota\zeta({}_i\kappa) \\
&\qquad\; b_i = [\![E_{i1}]\!]\iota\zeta({}_{(i+m)}\kappa) \\
&\qquad\; d_i = [\![E_{i2}]\!]\iota\zeta({}_{(i+2m)}\kappa) \\
&\qquad\; \mathit{valid} = \{i \in 1..n \mid \kappa \in k_i \wedge b_i \equiv \texttt{true}\} \\
&\qquad\; \mathit{best} = \{i \in \mathit{valid} \mid \nexists j \in \mathit{valid} \text{ s.t. } k_j \subset k_i\} \\
&\quad \text{in } \bigoplus\{d_i \mid i \in \mathit{best}\}
\end{aligned}
$$

It is important to realise that this semantics is not syntactic sugar on top of existing infrastructure, like the constructs presented in Chapter 3. The `bestof` construct is, in fact, a new primitive.

What is going on here is that for each guarded expression E_{i2}, the expression E_{i0} evaluates to a region k_i, E_{i1} evaluates to a Boolean value b_i, and E_{i2} evaluates to some value d_i. Then, if b_i is `true`, and the current context κ is in the region k_i, definition i is considered valid. Then, from all the valid definitions, the one whose region is not a superset of any other region is chosen. If multiple regions fit that criterion, then they are combined with the operator $\bigoplus$.

Currently, the definition of $\bigoplus$ is that it returns the special value `spmultidef` to indicate that there were multiple definitions for the best region. However, it is feasible that this could be changed in another system. Depending on what the user wants, the system could behave in several ways, so long as $\bigoplus$ is associative and commutative: unicity (the current solution), identity, sum, product, maximum, minimum, union, intersection, ...

There are two parts to the semantics that are undecidable if we do not make restrictions to the allowable sets: 1) the test $\kappa \in k_i$, which specifies that a context κ must be inside the region k_i; 2) the test $k_j \subset k_i$, which tests whether a region k_j is a subset of the region k_i. For these two operations, their decidability is determined by the possible regions that can be defined. These choices are fairly arbitrary, and as long as they are decidable, different decisions can be made for any particular implementation. The choices made for the current interpreter are to be pragmatic whilst still being useful.

A region is defined as a set of contexts. We define each set of contexts by defining a set of values for the ordinate of each dimension in that region. Therefore, a region can be understood as a mapping from dimensions to sets

$$k = \{\delta_i \mapsto s_i\}.$$

Let us look at how $\kappa \in k$ and $k \subset k'$ are specified, and see how their definitions lead naturally to an implementation. They are as follows:

$$
\begin{aligned}
\kappa \in k &\iff \text{dom } k \subseteq \text{dom } \kappa \wedge \forall \delta_i \in \text{dom } k,\ \kappa(\delta_i) \in s_i \\
k \subset k' &\iff \text{dom } k' \subseteq \text{dom } k \wedge \forall \delta \in \text{dom } k',\ k(\delta) \subseteq k'(\delta) \wedge \exists \delta_i \in \text{dom } k' \text{ s.t. } k(\delta_i) \subset k'(\delta_i).
\end{aligned}
$$

There are two components in the above two cases that must be restricted to make bestfitting decidable, $\kappa(\delta_i) \in s_i$ and $k(\delta) \subseteq k'(\delta)$: in other words, is an atomic value in some set, and is one set a subset of another. Therefore, to make bestfitting decidable, it is the definition of the set of values for an ordinate in a region that must be restricted.

Our solution is to restrict an ordinate to being one of the following three cases:

1. a single atomic value,
2. a type, or
3. a set whose containment procedure is decidable.

The concrete syntax for regions is similar to that for tuples, except that the left-arrow symbol is replaced with the type of containment being specified. There is a different symbol for each of the three cases above. The syntax is as follows:

$$\begin{aligned} \textit{region} &::= \texttt{[}E_{i0}\ \textit{containment}\ E_{i1}\texttt{]} \\ \textit{containment} &::= \texttt{is} \\ &\mid \texttt{imp} \\ &\mid \texttt{:} \end{aligned}$$

The three cases for *containment* correspond to the three cases above. For the third case, there are only two sets that can be specified

1. a range over integers, with the addition of the values `infty` and `negintfy`; and
2. another region.

One can imagine more types of sets for the third case, and more cases of containment, as long as they are decidable. For example, a regular expression type of containment could be implemented, so that strings matching a particular pattern could be bestfit against. One could also imagine implementing pattern matching with data types in a manner similar to Haskell. However, in such cases, one would need to be careful about checking $k_j \subset k_i$ should there be multiple valid regions.

8.5 Variable declarations

Here we extend the variable-declaration component of the `where` clause (§3.6) by adding bestfitting syntax. Several declarations can be made for the same identifier, each with a guard, and when the system is evaluated (§8.10), all of the definitions for each identifier are combined into one declaration whose expression is a `bestof` clause.

The syntax for variable declarations is:

$$\textit{vardecl} ::= \texttt{var}\ x\ \textit{guard}\ \texttt{=}\ E\ \texttt{;;}$$

where *guard* is the syntax E_0 `|` E_1 from §8.4.

For example, variable F from the factorial function example in §8.4 could be written as the following two variable declarations:

$$\begin{aligned}&\texttt{var } F\ [0\ \texttt{is}\ 0] = 1\\&\texttt{var } F\ [0 : 1..\texttt{infty}] = \texttt{\#}.0 \times (F\ \texttt{@}\ [0 \leftarrow \texttt{\#}.0 - 1])\end{aligned}$$

which are transformed, of course, into the form below:

$$\begin{aligned}\texttt{var } F = \texttt{bestof}\\ &[0\ \texttt{is}\ 0] \to 1\\ &[0 : 1..\texttt{infty}] \to \texttt{\#}.0 \times \big(F\ \texttt{@}\ [0 \leftarrow \texttt{\#}.0 - 1]\big)\\ \texttt{end}\end{aligned}$$

In general, a set of declarations

$$\texttt{var } x\ E_{i0} \,|\, E_{i1} = E_{i2}$$

is transformed into the following declaration:

$$\texttt{var } x = \texttt{bestof } E_{i0}\ \texttt{|}\ E_{i1} \to E_{i2}\ \texttt{end}$$

8.6 Function declarations

We extend function declarations in a manner similar to variable declarations, by allowing bestfitting to be built-in to the declaration. The syntax for a function declaration is:

$$\textit{fundecl} ::= \texttt{fun } x\ p_i\ x_i\ \textit{guard} \texttt{ = } E\ \texttt{;;}$$

where each p_i is one of '.', '!' or ' ', as in §3.7. As for variable declarations, which can have multiple declarations for each identifier, function declarations are rewritten to a single declaration with an appropriate expression, which defines a function, and additionally, we require that the parameters used in each declaration for the same identifier be consistent. We make use of the infrastructure provided by Operational TransLucid (Chapter 5), which replaces function parameters with ϕ dimensions, and allow function parameters to appear on the left-hand sides of the regions guarding each expression. Then, a series of n declarations gives ($i = 1..m, j = 1..n$):

$$\texttt{fun } x\ p_i\ x_i\ E_{j0} \,|\, E_{j1} = E_{j2}$$

is rewritten to the single variable declaration:

$$\texttt{var } x\ = \texttt{bestof } E_{j0}[x_i/\underline{\phi_{x_i}}]\ \texttt{|}\ E_{j1} \to \big(\mathcal{F}(p_1)\ x_1 \to \cdots \to \mathcal{F}(p_m)\ x_m \to E_{j2}\big)\ \texttt{end}$$

where $\mathcal{F}$ is defined in Figure 3.2, p.37.

8.7 Data types

The user can declare data constructors, which are a shorthand notation for a function that builds a tuple with specific fields present. There are two declarations used to define a data type with constructors: `data` for the type, and `constructor` to declare each data constructor.

The syntax of the `data` and `constructor` declarations is

$$\textit{datadecl} ::= \texttt{data}\ x\ \texttt{;;}$$
$$\textit{constructordecl} ::= \texttt{constructor}\ x\ x^*\ \textit{guard}\ \texttt{=}\ x\ \texttt{;;}$$

To define a data type T with constructors, the data type should be declared as

$$\texttt{data}\ T$$

then the constructors should be declared with the x to the right of the equals sign being T. Each constructor can be declared with as many arguments as desired, and can also be guarded, so that only certain types or sets of values can be used to construct a value.

A data object is just a tuple, and the data and constructor declarations use a number of dimensions in the resulting tuple to describe a data object. Every data object will always have the *type* and *cons* dimensions present, and they will be the name of the data type and the name of the constructor respectively. Then, if a constructor has m arguments, those arguments will be stored in the fields `arg0` through to `arg`$(m-1)$.

The data type itself can then be used as an identifier, because a `data` declaration for type T results in a declaration

$$\texttt{var}\ T = [\textit{type}\ \texttt{is}\ \text{“}T\text{”}]$$

Then, that type can be used with bestfitting, to match any argument that is a list whose *type* dimension is set to “T”.

For example, a list type could be declared as follows

$$\texttt{data}\ \textit{list}$$
$$\texttt{constructor}\ \textit{Nil} = \textit{list}$$
$$\texttt{constructor}\ \textit{Cons}\ a\ b\ [b : \textit{list}] = \textit{list}$$

Those three declarations would result in the following three variable declarations:

$$\begin{array}{l}\texttt{var}\ \mathit{list} = [\mathit{type}\ \texttt{is}\ \text{“}\mathit{list}\text{”}] \\ \texttt{var}\ \mathit{Nil} = [\mathit{type} \leftarrow \text{“}\mathit{list}\text{”}, \mathit{cons} \leftarrow \text{“}\mathit{Nil}\text{”}] \\ \texttt{var}\ \mathit{Cons} = \lambda^{\mathrm{v}} a \rightarrow \lambda^{\mathrm{v}} b \rightarrow \\ \quad \texttt{bestof} \\ \quad\quad [b : \mathit{list}] \rightarrow [\mathit{type} \leftarrow \text{“}\mathit{list}\text{”}, \mathit{cons} \leftarrow \text{“}\mathit{Cons}\text{”}, \mathit{arg0} \leftarrow a, \mathit{arg1} \leftarrow b] \\ \quad \texttt{end}\end{array}$$

We can then define the standard list functions *head* and *tail* as follows, with a guard to check that the list is not empty:

$$\begin{array}{l}\texttt{fun}\ \mathit{head}.l\ \big[l : [\mathit{type}\ \texttt{is}\ \text{“}\mathit{list}\text{”}, \mathit{cons}\ \texttt{is}\ \text{“}\mathit{Cons}\text{”}]\big] = l.\mathit{arg0} \\ \texttt{fun}\ \mathit{tail}.l\ \big[l : [\mathit{type}\ \texttt{is}\ \text{“}\mathit{list}\text{”}, \mathit{cons}\ \texttt{is}\ \text{“}\mathit{Cons}\text{”}]\big] = l.\mathit{arg1}\end{array}$$

With the *list* type, we can then write a length function that uses bestfitting to check that its argument is a list:

$$\begin{array}{l}\texttt{fun}\ \mathit{list\_length}.l\ [l\ \texttt{is}\ \mathit{Nil}] = 0 \\ \texttt{fun}\ \mathit{list\_length}.l\ [l : \mathit{list}] = 1 + \mathit{list\_length}.(\mathit{tail}.l)\end{array}$$

Below is an example of a function that flattens a list of lists, of arbitrary depth, to a single list:

$$\begin{array}{l}\texttt{fun}\ \mathit{flatten}.l\ [l\ \texttt{is}\ \mathit{Nil}] = l \\ \texttt{fun}\ \mathit{flatten}.l\ [l : \mathit{list}] = \mathit{concat}.\big(\mathit{flatten}.(\mathit{head}.l)\big).\big(\mathit{flatten}.(\mathit{tail}.l)\big) \\ \texttt{fun}\ \mathit{flatten}.l = \mathit{Cons}.l.\mathit{Nil}\end{array}$$

8.8 Operator declarations

For convenience, postfix, prefix, and binary infix operators can be defined by the user, along with their precedence and associativity. An operator is declared using the `op` declaration.

The syntax for an `op` declaration is:

$$
\begin{aligned}
\textit{opdecl} &::= \texttt{op}\ \textit{op}\ \texttt{=}\ \textit{oparg}\ \texttt{;;} \\
\textit{oparg} &::= \texttt{OpPostfix}.\textit{string}.\textit{call\_type} \\
&\quad \mid \texttt{OpPrefix}.\textit{string}.\textit{call\_type} \\
&\quad \mid \texttt{OpInfix}.\textit{string}.\textit{call\_type}.\textit{assoc}.\textit{intmp} \\
\textit{assoc} &::= \texttt{AssocNon} \\
&\quad \mid \texttt{AssocLeft} \\
&\quad \mid \texttt{AssocRight} \\
\textit{call\_type} &::= \texttt{cbv} \\
&\quad \mid \texttt{cbn} \\
\textit{op} &::= (\ \textit{Symbol}\ \mid \texttt{!} \mid \texttt{\%} \mid \texttt{*} \mid \texttt{-} \mid \texttt{.} \mid \texttt{\&} \mid \texttt{/} \mid \texttt{:}\ )^{+}
\end{aligned}
$$

The symbols `OpPostfix`, `OpPrefix` and `OpInfix` are constructors whose data type is `OpType`; the symbols `AssocNon`, `AssocLeft` and `AssocRight` are nullary constructors whose data type is `Assoc`; and the symbols `cbv` and `cbn` are nullary constructors whose data type is `CallType`. Their full definition is in §9.1.

The parser uses the information provided by the operator declarations to build a parse tree using the precedence and associativity declared by the user. Take as example the following expression. We write op_n to mean an operator symbol of precedence n, and assume that the operators are left-associative. If the input is of the form:

$$E_0\ op_1\ E_1\ op_2\ E_2$$

then after recognising the operator symbols and looking up their precedence, the parser would parse this as:

$$E_0\ op_1\ (E_1\ op_2\ E_2)$$

Supposing that the operators are both for call-by-value functions f and g respectively, then that expression would be subsequently rewritten to:

$$f\ !\ E_0\ !\ (g\ !\ E_1\ !\ E_2)$$

which is the final TransLucid abstract syntax.

8.8.1 Postfix and prefix operators

The postfix and prefix operators are defined similarly, using the `OpPostfix` and `OpPrefix` functions. They both take two arguments: the first being the function to map the operator to, and the second being `true` to map to call-by-name, and `false` to map to call-by-value, allowing the user to choose whether the arguments to an operator are evaluated lazily or eagerly.

If we have defined an operator like so

$$\texttt{op}\ s\ = \texttt{OpPrefix}.f.\texttt{true}$$

then any occurrences of

$$s\ E$$

will be translated to

$$f\ E$$

and the following function declaration is expected

$$\texttt{fun}\ f\ X = E'$$

The case is similar for call-by-value and postfix operators.

8.8.2 Binary infix operators

In addition to the first two parameters, which are the same as for postfix and prefix operators, the associativity and precedence of binary operators are defined by the user.

For associativity, there are three cases:

`AssocNon` the operator is parsed as non-associative, meaning that it is an error for more than two non-associative operators to appear next to each other;

`AssocLeft` the operator is parsed as left-associative, meaning that more than one operator of the same precedence next to each other will be grouped from the left;

`AssocRight` the operator is parsed as right-associative, meaning that more than one operator of the same precedence next to each other will be grouped from the right.

The precedence of operators is specified as an `intmp` value, meaning that any integer representable on the host system is available as a precedence. The parser groups operators based on their precedence if multiple operators of the same associativity appear next to each other, then, as by the standard rules, operators are grouped based on which has higher precedence.

The ability to define operators as either call-by-value or call-by-name has an interesting implication, in that certain operations, such as conditionals, can be lazy or eager as the user desires. For example, the TransLucid standard library has the following declarations for the `||` and `&&` operators:

$$\texttt{op ||} = \texttt{OpInfix."bool\_or".true.AssocLeft}.15$$

$$\texttt{op \&\&} = \texttt{OpInfix."bool\_and".true.AssocLeft}.20$$

$$\texttt{fun}\ \textit{bool\_or}\ X\ Y = \texttt{if}\ X\ \texttt{then true else}\ Y\ \texttt{fi}$$

$$\texttt{fun}\ \textit{bool\_and}\ X\ Y = \texttt{if}\ X\ \texttt{then}\ Y\ \texttt{else false fi}$$

Since both of these are call-by-name, the second argument to both will not be evaluated unless the first is true in the case of `||`, and false in the case of `&&`.

In addition to operators like `||` and `&&` that can be implemented completely within the language, there are a number of operators that are mapped to host functions §8.3 through the call-by-value functions that they are mapped to by the `op` declaration. For each operator that is mapped to a call-by-value function, that function can have multiple definitions, guarded by the appropriate types. This allows multiple data types to use the same operator symbol for an operation. For example, the TransLucid standard library has the following declaration for addition:

$$\texttt{op} + = \texttt{OpInfix."plus".false.AssocLeft}.100$$

For *intmp* addition, there is the following definition of the function *plus*:

$$\texttt{fun}\ plus\,!\,a\,!\,b\,[a\ \texttt{imp}\ intmp, b\ \texttt{imp}\ intmp] = intmp\_plus.(a, b)$$

8.9 External libraries

The TransLucid programming environment can be extended through the use of external libraries loaded at runtime. An external library can add to the system in two ways: 1) by adding new data types and functions, which extends the language by adding to the set D and the function ι; and, 2) by adding any of the declarations described in this chapter.

For an external library to work, there must be a programming API for each of the declarations, and the host environment (the programming language, the operating system and the compilation model) must have the ability to load dynamic libraries at runtime. In the current implementation, which is written in C++, and runs in a Unix environment, libraries are opened using the `libtool` library, and if the library's name is `tllib`, then it must provide a function `lib_tllib_init`, which initialises the library and adds its declarations to the system.

A library is declared using the `library` declaration, which has the following syntax

$$libdecl ::= \texttt{library}\ id\ \texttt{;;}$$

The declaration

$$\texttt{library}\ l\ \texttt{;;}$$

opens the library whose name is l, and whose executable can be referred to as `lib`l, and is found in the operating system's standard search path. The function `lib_`l`_init` is then run, which should add the libraries declaration to the system.

8.10 The TransLucid system

A TransLucid system is a set of declarations, which are provided with the purpose of giving the definitions of variables, functions, data types, and operators so that a set of

expressions can be evaluated. To illustrate this point, we present a small session in the `tltext` command line interface to the TransLucid interpreter [5]:

```
op ** = OpInfix."exponent".cbv.AssocRight.400 ;;

fun exponent!a!b [a imp intmp, b imp intmp] = exp.b.a ;;

fun exp.n = P
where
  dim d <- n ;;
  var P = fby.d (\_s -> 1) (\_ {d} s -> s * P.s) ;;
end ;;

%%

2**10 ;;
```

Here, we define an operator `**`, which stands for the function *exponent*, which is provided as a suitable interface to the *exp* function presented in §4.13. This is defined so that the expression `2**10` is equivalent to the function call `exp.10.2`, and computes 2 raised to the 10th power. The symbol `%%` separates the declarations from the expressions to be computed.

The presentation above glossed over the fact that several of the declarations require the evaluation of other declarations for the parser to even work. For example, to parse an operator symbol *op*, it is necessary to evaluate the function application *operator*.*op*. The definition of *operator* in turn uses several data declarations. In addition, to compile a variable declaration for an identifier x, it is necessary that all of the variable declarations for x be present, so that it can be combined into one definition. So how does all of this work so that declarations can be parsed and processed in the appropriate order and have the system still work?

The syntax is designed so that all of the declarations can be recognised without requiring them to be parsed. This way, they can all collected without being processed, and then they are parsed in a completely lazy manner. The parser starts by parsing the expressions to be evaluated, and then only parses declarations as they are required. Furthermore, once evaluation starts, declarations are still only parsed as they are required. This has the side effect that declarations that are never needed are not even parsed.

From this point it is a trivial matter to map the concrete system into Operational TransLucid. Conceptually, the declarations presented in this chapter, along with any dimension declarations (allowing global dimensions), are grouped together into one (potentially very large) `where` clause, whose body is the expression to be evaluated. Should there be more than one expression to evaluate, each expression is evaluated as though it is the body of the aforementioned `where` clause.

8.11 Time

The system presented in the previous section is still not the complete story. The complete Translucid interpreter, in fact, is a programming environment—a synchronous reactive system—that evaluates expressions using declarations presented in a sequence of time *instants*. Everything presented up to now is for one instant, and is still valid for any single instant, with some rules about the interaction with previous and future instants. The extension to multiple time instants is achieved by the system simulating a sequence of discrete time instants, and by the addition of a `time` dimension whose ordinate is the current instant—a natural number, starting from 0—being simulated.

At each time instant being simulated, the system takes as input all the declarations for that instant, and evaluates any expressions requested. Then, the clock is incremented and the process repeats until something tells the system to stop.

To allow the user to access information about which instant the system is currently running, we add a `time` dimension, whose ordinate when evaluating a demand will always be the number denoting the current instant. Consider the following input to the `tltext` program to see the use of the `time` dimension:

```
%%
#.time ;;
$$
%%
#.time ;;
```

the output is:

```
1
2
```

The current instant is terminated using the `$$` token, which tells the system to process the demands and declarations from the current instant, increments the clock and waits for more input. The only difference between instants is the `time`-ordinate.

Some restrictions must be made to the use of the `time` dimension. To be consistent with time in the real world, we must not be able to change the past, we can only access the past if the information from the past still exists, and we cannot access the future. Therefore, we make the following restrictions in how the `time` dimension is used:

1. Declarations only affect instants from when they were made, and no declaration can be made that changes a past declaration.

2. A computation looking into the past can be carried out, and the system will compute whatever it can based on the declarations that it still has. If some declarations have been deleted, then the computation will fail.

3. The `time`-ordinate cannot be increased using a context change expression. Attempting to do so will produce an `spaccess` special value.

At any given instant, the set of declarations used by the system is the set of the most recent declarations for each variable. In addition, a declaration can have the `time` dimension mentioned in its guard, however, this is orthogonal to whether a definition is included in the current set of declarations. Bestfitting is about choosing the most specific definition during evaluation, after the appropriate declarations for an instant have been chosen.

For example, if the following declaration for variable A were made at instant 5:

$$\texttt{var}\ A\ [\texttt{time is}\ 3] = E\ \texttt{;;}$$

it would never be used, because it would not be seen at instant 3.

8.12 Input and output

Finally, to conclude this chapter, we present a means for the system to take *finite* multidimensional arrays as input, and to produce *finite* multidimensional arrays as output. Up to this point, except for demand-driven evaluation, the discussion has been entirely about the infinite, but at some point, it is necessary for finite objects to be passed around—either by the system being required to compute some finite amount of data, or by there being some finite amount of input to the system that the system is required to transform.

As alluded to in Chapter 6, an input is just a cache, that has the relevant entries already filled-in. This was necessary because the cached semantics uses ξ, a mapping from identifiers to *expressions*, instead of ζ, a mapping from identifiers to *intensions*. Any input to the system can be assumed to be in ζ, because it is a mathematical object. But this doesn't work for ξ, because it specifically maps to expressions, and the inputs to the system are most likely not presented as expressions. Therefore, it is necessary to initialise a cache with every entry in any inputs.

In Chapter 6, using a cache for inputs was the natural choice, given that the environment mapped to expressions. In moving to the concrete system, it is, in fact, the correct solution that all inputs be initialised as a cache. This is consistent with the intuitive point of view—a cache is just a store of values, as is an input.

We add two declarations to the concrete system so that it can accept multidimensional arrays as input: `indim` and `invar`. It is necessary to declare dimensions that will be in the domain of input variables so that the system knows that it should treat these as special cases. The syntax of the `indim` declaration is:

$$\texttt{indim}\ x\ \texttt{;;}$$

The input variable declaration is similar to the variable declaration, including with bestfitting, except that each guarded definition is a `URL` to a source location, which is a multidimensional array. This allows a single input variable to have multiple sources and to be grouped together so that a whole array of inputs can be referred to by one variable. In addition, an input variable can be initialised with an expression so that some part of

the input variable has a default value. The syntax for the `invar` declaration is:

$$\texttt{invar}\ x\ guard \leftarrow source\ \texttt{;;}$$
$$\texttt{invar}\ x\ guard = E\ \texttt{;;}$$

For the second version of the definition, E should either be constant or other inputs, with no dependency loops between variables.

Output variables are declared similarly, and are also informing the system about a finite multidimensional array. However, the semantics is entirely different. Writing to an output variable is the only part of the whole TransLucid system that produces a side effect. Up to this point, everything presented has been declarations, since TransLucid is a declarative language. Other than evaluating single expressions, there has been no means to make the system actually do anything. Despite the fact that writing to output is an operation with a side effect, it is still done declaratively. Since the TransLucid system evaluates with respect to a `time` dimension, output declarations are made per-instant, and these have to be consistent. Only if a whole output request is defined, and is consistent with all other declarations, is the output sent off at the end of an instant.

There are three keywords for declaring outputs: `outdim`, `outvar` and `outvardest`. The reasons for needing `outdim` are analogous to those for `indim`. Its syntax is:

$$\texttt{outdim}\ x\ \texttt{;;}$$

The `outvardest` declaration provides the system with the location of an output variable, which is similar to an input variable, but it is understood that the location is a multidimensional sink. The syntax for the `outvardest` declaration is:

$$\texttt{outvardest}\ x\ guard \rightarrow dest\ \texttt{;;}$$

The syntax for the `outvar` declaration is:

$$\texttt{outvar}\ x\ guard\ \texttt{:=}\ E\ \texttt{;;}$$

The reason for having a guard for both `outvardest` and `outvar` is that the location being written to could be defined over several regions, and the data being written could be defined by several expressions over different regions.

8.13 Conclusion

This chapter has presented a complete Concrete TransLucid system, which is completely declarative, with physical inputs and outputs, yet is reactive and responds to requests to do work. Furthermore, none of this required another language to describe—all of these features can be described in the same language, simply by extending the declaration mechanism, and using the existing expression evaluation infrastructure.

This Concrete TransLucid system is implemented as a `C++` library, and the `tltext` interface (§8.10) is just a textual interface to that library. In fact, as part of the source distribution, a web-based text interface is provided, which is similar to the `tltext` interface; a version of this interface is running at `http://translucid.web.cse.unsw.edu.au/tlweb`. This means that the inputs and outputs to `tltext` can come directly from `C++`, or from any location that the user can write down.

Not all of the input and output mechanism presented is actually implemented, rather, the declarations presented in §8.12 are only implemented internally. The `tltext` interface uses the `C++` system API to add declarations and demands for computation internally; in addition, arguments can be passed on the command line which are added as an input variable. However, there is currently no mechanism for the user to add input from arbitrary sources.

What is missing from all of this is a *system abstraction*, which would be a TransLucid system that can be created inside another system, and passed around just like any other abstraction. This would require a clock dimension that can be passed as a parameter, so that a system abstraction can exist inside another system at a single instant, but have its own clock, accessible by the outer system. The clock of the internal system could then run infinitely faster than the outer system, and in one outer tick, the outer system could pull out the result of calculations after the internal system has run several ticks of its internal clock.

Also required is a semantics for multiple systems interacting with each other, adding input to and requesting demands from each other. But for all this the focus would be the semantics of time, synchronisation of clocks, distribution of computation, atomicity of communication and all of the other difficulties inherent in distributed computation.

The author, with John Plaice and Blanca Mancilla, designed a mechanism for a system abstraction to be created internally to an outer system, along with a clock dimension, presented in a 2013 *Spatial Computing Workshop* article [33]. It is necessary to pass finite multidimensional arrays to internal systems, and to be able to accept finite multidimensional arrays as the output from those systems. So part of that mechanism includes a *block demand* expression, written $E\,\$\,[\cdots]$, which forces the evaluation of E over the specified region, so that the resulting finite array can be passed around. The key idea of the article was multiple systems sitting in a systolic array, all carrying out a small part of the overall computation. However, this lacked proper semantics and has not been implemented. Nevertheless, the approach is correct, and if the issues brought up by this chapter were resolved, systolic arrays of systems would be trivial in TransLucid.

The open problems left by this chapter are substantial, and are probably worth another PhD in themselves. With these problems resolved, TransLucid would become suitable for various methods of distributed programming, all within a declarative framework, and all described by the one language, with no need for extra formalisms, languages or systems to coordinate the processing nodes.

Chapter 9

The TransLucid Standard Library

This chapter describes the TransLucid standard library, which is defined in the standard header to `tltext`, found in the file `src/tltext/header.tl` in the TransLucid distribution [5].

9.1 Data types

The TransLucid standard library defines several data types, defined using the `data` and `constructor` declarations. These are described below.

9.1.1 Associativity

The data type describing the three types of associativity of operators: left, right and non-associative is declared as:

$$\texttt{data}\ Assoc$$

and has the following members:

$$\begin{aligned}\texttt{constructor}\ AssocNon &= Assoc\\ \texttt{constructor}\ AssocLeft &= Assoc\\ \texttt{constructor}\ AssocRight &= Assoc\end{aligned}$$

The associativity data objects are used in defining the TransLucid operators (§9.2).

9.1.2 Call type

The data type used to describe how a function is called is defined as follows:

$$\texttt{data}\ CallType$$

and has the following members:

$$\texttt{constructor}\ cbv = CallType$$
$$\texttt{constructor}\ cbn = CallType$$

9.1.3 Operator types

The data type describing the operator types: prefix, postfix and infix, which are used to define each TransLucid operator (§9.2), is declared as:

$$\texttt{data}\ OpType$$

and has the following members:

$\texttt{constructor}\ OpPostfix\ a\ b\ [a\ \texttt{imp}\ ustring, b\ \texttt{imp}\ bool] = OpType$

$\texttt{constructor}\ OpPrefix\ a\ b\ [a\ \texttt{imp}\ ustring, b\ \texttt{imp}\ bool] = OpType$

$\texttt{constructor}\ OpInfix\ a\ b\ c\ d\ [a : ustring, b : CallType, c : Assoc, d : intmp] = OpType$

9.2 Operators

The infix binary operators are defined in Table 9.1. Their definitions are in the standard header in the form:

$$\texttt{op}\ Symbol = \texttt{OpInfix}.string.bool.\texttt{Assoc}.intmp$$

Table 9.1: TransLucid infix operators

Operator	Precedence	Associativity	cbn/cbv	Function
+	100	left	cbv	*plus*
-	100	left	cbv	*minus*
*	200	left	cbv	*times*
/	200	left	cbv	*divide*
%	200	left	cbv	*modulus*
<	50	non	cbv	*lt*
<=	50	non	cbv	*lte*
>	50	non	cbv	*gt*
>=	50	non	cbv	*gte*
==	25	non	cbv	*eq*
!=	25	non	cbv	*ne*
&&	20	left	cbn	*bool_and*
||	15	left	cbn	*bool_or*
>>	100	left	cbv	*concatenate*
..	0	non	cbv	*range_construct*

The unary prefix operators are described in Table 9.2. They are declared in the standard header as:

$$\texttt{op}\ Symbol = \texttt{AssocPrefix}.string.bool$$

Table 9.2: TransLucid prefix operators

Operator	cbv/cbn	function
$\neg$	cbv	*negate*
$\surd$	cbv	*sqrt*

9.3 Variables

There are several pre-defined variables in the standard library; these are described in the following sections.

9.3.1 Ranges

There are several pre-defined ranges over integers; these are as follows:

$$\texttt{var}\ \ pos = 1..infty$$
$$\texttt{var}\ \ nat = 0..infty$$
$$\texttt{var}\ \ int = neginfty..infty$$
$$\texttt{var}\ \ neg = neginfty..\tilde{}1$$

9.3.2 Types

Each of the predefined types has an associated identifier; these are as follows:

$$\texttt{var}\ \ intmp = \texttt{type"}intmp\texttt{"}$$
$$\texttt{var}\ \ uchar = \texttt{type"}uchar\texttt{"}$$
$$\texttt{var}\ \ ustring = \texttt{type"}ustring\texttt{"}$$
$$\texttt{var}\ \ floatmp = \texttt{type"}floatmp\texttt{"}$$
$$\texttt{var}\ \ bool = \texttt{type"}bool\texttt{"}$$
$$\texttt{var}\ \ special = \texttt{type"}special\texttt{"}$$

9.3.3 Special values

None of the special values are recognised by the parser, and need to be defined using type literal syntax. For convenience, we declare a variable for each special value as follows:

$$\texttt{var}\ \ spdim = \texttt{special"}dim\texttt{"}$$
$$\texttt{var}\ \ spaccess = \texttt{special"}access\texttt{"}$$
$$\texttt{var}\ \ sptypeerror = \texttt{special"}typeerror\texttt{"}$$
$$\texttt{var}\ \ spundef = \texttt{special"}undef\texttt{"}$$

```
var spmultidef = special"multidef"
var sploop = special"loop"
```

9.4 Atomic functions

There are a number of functions that operate only on atomic values. These are described in the following sections.

9.4.1 `min` and `max`

The function *min.a.b* (resp. *max.a.b*) returns the smaller (resp. greater) of two values a and b.

```
fun min.a.b = if a < b then a else b fi
fun max.a.b = if a < b then b else a fi
```

9.4.2 `ilog`

The function *ilog.n* computes $\lceil \log_2(n+1) \rceil$.

```
fun ilog.n = asa.d (#.d) (double ≥ n)
where
  dim d ← 0
  var double = fby.d 1 (double × 2)
end
```

9.5 Intensional functions

There are several standard functions that manipulate intensions in particular directions. These are presented in the subsections below.

9.5.1 `at`

The function *at.d.n X* is provided as a convenience for changing the context with a single dimension.

```
fun at.d.n X = X @ [d ← n]
```

9.5.2 `first`

The function *first.d X* returns element zero of the intension X, in direction d.

```
fun first.d X = at.d.0 X
```

9.5.3 wvr

The function *wvr.d X Y* takes two intensions, X and Y, both varying in dimension d, and defines an intension which has the values of X at the positions where Y is `true`.

```
fun wvr.d X Y = at.d.T X
where
  var T = fby.d U (at.d.(T + 1) U)
  var U = if Y then #.d else next.d U fi
end
```

9.5.4 upon

The function *upon.d X Y* takes as input two intensions, and creates a stream that repeats the current element of X as long as Y is `false`, only giving the next element of X when Y is `true`.

```
fun upon.d X Y = at.d.T X
where
  var T = fby.d 0 (if Y then T + 1 else T fi)
end
```

9.5.5 merge

The function *merge.d X Y* merges the two sorted infinite arrays X and Y.

```
fun merge.d X Y = if X' ≤ Y' then X' else Y' fi
where
  var X' = upon.d X (X' ≤ Y')
  var Y' = upon.d Y (Y' < X')
end
```

9.5.6 asa

The function *asa.d X Y* returns the first entry of X for which the corresponding entry of Y is `true`.

```
fun asa.d X Y = first.d (wvr.d X Y)
```

9.5.7 rotate and transpose

The function *rotate.d.d′ X* takes as input two dimensions and an intension, and produces an intension having the values of X which varies in dimension d' instead of d.

```
fun rotate.d.d' X = at.d.(#.d) X
```

The function *transpose.d.d′ X* takes as input two dimensions d and d', and an intension X, and swaps dimensions d and d' for X.

$$\texttt{fun}\ \mathit{transpose}.d.d'\ X = X\ @\ [d \leftarrow \#.d', d' \leftarrow \#.d]$$

9.5.8 Default values

There are one- and two-dimensional functions for surrounding values from an intension in a sea of default values. These both take as input an intension X and the region over which to use values from X. They are defined as follows:

$$\begin{array}{l}
\texttt{fun}\ \mathit{default}_1.d.m.n.v\ X = Y \\
\texttt{where} \\
\quad \texttt{var}\ Y\ [d : m..n] = X \\
\quad \texttt{var}\ Y\ [d : nat] = v \\
\texttt{end} \\
\\
\texttt{fun}\ \mathit{default}_2.d_1.m_1.n_1.d_2.m_2.n_2.v\ X = Y \\
\texttt{where} \\
\quad \texttt{var}\ Y[d_1 : m_1..n_1, d_2 : m_2..n_2] = X \\
\quad \texttt{var}\ Y[d_1 : nat, d_2 : nat] = v \\
\texttt{end}
\end{array}$$

9.5.9 Divide-and-conquer functions

There are two functions for tournament-style computation: one for computations in one dimension, another for computations in two dimensions. A tournament computation in one dimension collapses elements next to each other using a user-specified function, and in two dimensions collapses groups of four elements next to each other.

The two functions *lPair.d X* and *rPair.d X* give the left and right elements of the input array X in direction d.

$$\begin{array}{l}
\texttt{fun}\ \mathit{lPair}.d\ X = X\ @\ [d \leftarrow \#.d \times 2] \\
\texttt{fun}\ \mathit{rPair}.d\ X = X\ @\ [d \leftarrow \#.d \times 2 + 1]
\end{array}$$

There are four functions for producing the elements from the appropriate corners of a two dimensional array for a two-dimensional tournament computation. These are defined as follows:

$$\begin{array}{l}
\texttt{fun}\ \mathit{NWofQuad}.d_1.d_2\ X = X\ @\ [d_1 \leftarrow \#.d_1 \times 2, d_2 \leftarrow \#.d_2 \times 2] \\
\texttt{fun}\ \mathit{SWofQuad}.d_1.d_2\ X = X\ @\ [d_1 \leftarrow \#.d_1 \times 2 + 1, d_2 \leftarrow \#.d_2 \times 2] \\
\texttt{fun}\ \mathit{NEofQuad}.d_1.d_2\ X = X\ @\ [d_1 \leftarrow \#.d_1 \times 2, d_2 \leftarrow \#.d_2 \times 2 + 1] \\
\texttt{fun}\ \mathit{SEofQuad}.d_1.d_2\ X = X\ @\ [d_1 \leftarrow \#.d_1 \times 2 + 1, d_2 \leftarrow \#.d_2 \times 2 + 1]
\end{array}$$

The two functions $\mathit{tournamentOp}_1.d.n.g\ X$ and $\mathit{tournamentOp}_2.d_1.d_2.n.g\ X$ carry out

a tournament computation in one and two dimensions respectively. The parameter n is for how many elements to use for the computation, and g is a function that combines the elements at each level.

```
fun tournamentOp_1.d.n.gX  = first.d Y
where
  dim t ← ilog.n
  var Y = fby.t X (g ! (LofPair.d Y) ! (RofPair.d Y))
end

fun tournamentOp_2.d_1.d_2.n.gX = first.d_1 (first.d_2 Y)
where
  dim t ← ilog.n
  var Y = fby.t X (g ! (NWofQuad.d_1.d_2 Y) ! (NEofQuad.d_1.d_2 Y)
                     ! (SWofQuad.d_1.d_2 Y) ! (SEofQuad.d_1.d_2 Y))
end
```

Chapter 10

Conclusions

This thesis presented the TransLucid programming language, from its denotational semantics in Chapter 2, right through to its concrete implementation in Chapter 8. In presenting TransLucid, this thesis set out to solve several open problems left in the history of Lucid, and added a few questions of its own. These were:

- Dimensions as first-class values, atomic values as dimensions, and contexts as first-class values;
- The semantics and implementation of higher-order functions over intensions;
- The semantics and implementation of a cached evaluator;
- The semantics and implementation of a concrete TransLucid system;
- Static analysis with the assumption that the principal type of an object is itself.

The first solved problems are higher-order functions with first-class dimensions and contexts; these are solved in Chapters 2 and 3. In addition, there is an underlying goal of making the language implementable. The presentation starts with the syntax and denotational semantics of a basic function language, called Core TransLucid. Its syntax (Figure 2.1, p.23) is specified by adding only four new syntactic elements to those of standard functional programming languages: the context (`#`), the tuple constructor $[\cdots]$, the context perturbator (`@`), and the `wheredim` clause. Then, the denotational semantics of a variable is an intension, which maps contexts to the semantic domain $\mathbf{D}$ (Definition 1, p.20); the semantics of an expression is a mapping from environments to intensions, where an environment maps variables to intensions. This semantic domain is based on the set D of atomic objects, whose members, as explained in Chapter 8, can be chosen by an implementation.

The first version of the denotational semantics, called non-deterministic (Definition 12, p.24), gives the meaning of an expression E in an environment ζ and a runtime context κ. Identifiers in the environment are lexically bound, whilst the entities in the context are dynamically bound, *permeating* the entire program, in a manner similar to UNIX environment variables. However, unlike for UNIX environment variables, the control of the TransLucid context is extremely fine-grained. This runtime context is used as an index

into the intension defined by E in the environment ζ. The `wheredim` clause allows unique local dimensions to be allocated, so that an array used for a subcomputation can be made orthogonal to every other array in a program. These dimensions are allocated from an infinite set of available dimensions (Figure 2.3, p.25), which is split at each branch in the evaluation tree, ensuring that each dimension allocation is unique. However, since this is not implementable, the deterministic semantics (Definition 13, p.26) takes the first step in moving TransLucid towards an implementation by making the allocation of dimensions deterministic, by threading a list through the ρ ordinate of the running context κ, which represents the evaluation stack of the current expression. Then, the `wheredim` clause in Figure 2.4 (p.26) allocates dimensions by taking values from a set of dimensions χ^i_ν, indexed by the list $\nu = \kappa(\rho)$ and a natural number i.

Whilst the Core TransLucid language can completely express higher-order intensional programming, expressing higher-order functions and writing canonical TransLucid programs in Core TransLucid is verbose. Therefore, Chapter 3 presents TransLucid, which is the language in which the user writes expressions. TransLucid adds several function abstractions to the core language, and combines the `wherevar` and `wheredim` clauses into a single `where` clause, all without changing the core language, but by defining syntactic transformations from TransLucid to Core TransLucid. In some sense, TransLucid is to Core TransLucid as `C++` is to `C`; they are equally powerful, yet TransLucid produces clearer and more concise programs. The example in §3.6 (p.36) demonstrates the reduction in verbosity by using a single `where` clause. The abstractions of TransLucid are implemented using Core TransLucid by passing the appropriate context around when abstractions are created and applied. For example, the intension abstraction expression $(\uparrow E)$ creates a function that takes a context as parameter, and evaluates its body in that context $(\lambda\kappa \to E \mathbin{@} \kappa)$, then the intension application expression $(\downarrow E)$ simply applies the current context to its body (E.`#`). The other abstractions are defined similarly.

The choice of function abstractions in TransLucid is made by recognising the different ways that abstractions can behave with respect to their argument and the context. There are five options available for function abstractions, a function can:

1) take no argument (intensions, with item (4));

2) take one argument (base, call-by-value and call-by-name functions), for these there are two more options:

   (a) their argument is evaluated when the function is applied (base functions, with item (3), and call-by-value functions, with item (4)),

   (b) their argument is evaluated at the context in which it is used inside the body of the function (call-by-name functions, with item (4));

3) have their body evaluated without respect to the runtime context; or

4) have their body evaluated with respect to the runtime context.

Combining options (1) and (3) would give a constant intension, and has little use outside the semantics (see Figure 2.2, item (2.5), p.24). Option (2)(b) with (3) would give

call-by-name functions that are not evaluated with respect to the current context, which also seems to have little point. In summary, the valid options for abstractions are: i) the intension, which takes no parameter and is evaluated with respect to the context at which it is applied; ii) the base function, which takes one parameter which is evaluated when applied, and whose body is evaluated with no context; iii) the call-by-value function, which takes one parameter which is evaluated when applied, and whose body is evaluated with respect to the context at which it is applied; and iv) the call-by-name function, which takes one parameter that is evaluated with respect to the context at which it is used inside the function, and whose body is evaluated with respect to the context at which it is applied.

With the semantics of higher-order functions solved, the presentation moves towards an implementation. The key problem in implementing TransLucid is the use of the environment for abstractions. For the semantics of base functions and call-by-value functions (Figure 2.2 item (2.5) and equation (3.4), p.35), both pass their function parameters by modifying the environment with a constant intension. The abstraction created needs a closure over the entire environment to function correctly. Chapter 5 presents the way in which uses of the environment are transformed to uses of the context, leaving the only manipulation of the environment to the `wherevar` clause. This way, no closure over parts of the environment is required, and the exact dimensions that need to be retained for the closure to be correct are made explicit in the Φ parameter that is part of each abstraction (see §5.3, p.70).

Chapter 6 continues the goal of moving towards an implementation by presenting *eduction*, which evaluates a TransLucid expression in a single context, and repeatedly evaluates any resulting demands for (x, κ) pairs until the result is reached. However, this naive model leads to huge amounts of repeated computation, and is not effective. To alleviate this, results are *cached*. The difficulty with caching is that the dimensions of relevance are not known until one begins evaluating an expression. To solve this, §6.4 (p.82) describes a back-and-forth interaction between the evaluator and each cache node, with §6.6 (p.88) giving an operational semantics of that procedure. This solves a problem existing in Lucid-like languages ever since multidimensional streams, which goes back to the original pLucid interpreter [43]; that feature was, in fact, present, but undocumented because of the lack of a solution. Furthermore, multidimensionality became explicit in Indexical Lucid [9] in 1995, further compounding the problems, and since then there has been no complete solution to the problem of caching.

Completing the goal of moving towards an implementation, Chapter 8 presents Concrete TransLucid, which builds a complete TransLucid system that takes real input and produces real output. Concrete TransLucid specifies that the atomic objects provided by the host system are the base set D for the semantic domain $\mathbf{D}$, and that the operations provided by the host system are the interpretation of constants ι. These are provided by the concrete system as the set of atomic objects that can exist in the host environment, and the set of functions that can operate over them. For these to be accessible inside the language, concrete syntax is provided for writing down atomic objects in §8.1 (p.143), along with concrete syntax for expressions. Some of the functionality of the concrete sys-

tem is built in TransLucid itself, so a minimum set of objects in D is required for the system to work (§8.2, p.149); these are the Booleans, integers, Unicode characters, and Unicode strings.

Concrete TransLucid contains a number of declarations, which are all syntactic sugar, to enrich the programming environment with more useful notation. Many declarations can be guarded by *context regions*, and then only the definition applicable will be chosen, so first a new kind of conditional called *bestfitting* is defined (§8.4, p.152). Then the different kinds of declarations are presented: the variable declaration (§8.5, p.155), the function declaration (§8.6, p.156), the data type declarations (§8.7, p.157), and the operator declarations (§8.8, p.158), which make the syntax for expressions more friendly to the user. In addition, the system has both a library interface and a textual interface, so that it can be used by both `C++` programs and from the command line.

Finally, the concrete system was extended by adding time (§8.11, p.163), and input and output (§8.12, p.164), to produce a *synchronous reactive system*. This system allows arbitrary multidimensional input to be given to the system, and for the system to produce multidimensional arrays as output, furthermore, this can be done as the system simulates clock ticks. The deficiencies in input and output are that in the current implementation, the means to specify the location of an input or output is quite limited. But the infrastructure is there for this to be extended.

It is at this point that the development of Lucid comes full circle. The original idea for Lucid, in 1974, was that a variable was an infinite stream changing through time. Whilst the semantics of a variable evolved to a multidimensional intension, the semantics of a system is exactly a set of streams varying through time, which obey the prefix order. The whole system itself with the relevant inputs and outputs all change through time, the semantics of which is constrained by the fact that time in this universe continually advances. The result being that one can only look into the past if it is remembered, the past cannot be changed, and it is impossible to look into the future. The implications for input and output are clear, because the order of operations is fixed by the progression of time, so an input that arrives at time t necessarily arrives before an input at time $t + 1$, similarly for producing output.

The problems left by the development of Lucid have been completely solved, with the implementation of a concrete TransLucid system allowing the user to evaluate TransLucid expressions, and with the cached evaluator providing an effective and efficient mechanism for evaluating in a completely dynamic system. This work has left several of its own problems, and ideas for future work.

The cached semantics left an open problem: by changing the way that dimensions are allocated to allow the cache to work, not all programs produce the same result as the denotational semantics. However, we do not consider this to be a problem, since it is suspected that only programs that would otherwise produce an error, or at least that are so bizzare that no one would write them, will not be evaluated correctly. Nevertheless, it is desirable that in the future there be some decidable procedure, i.e., a static semantics, to determine if a program can be run correctly using the cached semantics.

Chapter 8 leaves open the problem of a complete semantics of a system abstraction, with input, output, synchronisation and all that distributed computing entails. Additionally, the idea of a system changing through time, or other *physical dimensions*, leaves open numerous possibilities. Some of these are:

- A system becomes a new kind of abstraction, and `time` is not a special dimension, but rather, a *clocked dimension* passed as input from the system's caller, possibly declared therein.
- In each instant, there could be a set of *environment variables*, which are fixed for that instant, but could take on different values in future instants.
- The exact parameters of numeric data types could differ from one instant to another, or from one place to another. These parameters include the default precision of fixed-precision integers or floating-point numbers, whether or not to use overflow-protected integers, and how to handle floating-point exceptions.
- Multiple systems could be composed into a larger one. A particular case is that of *systolic algorithms*. Ideas in this direction were presented by the author with John Plaice and Blanca Mancilla in [33].
- Assuming advances in the semantics of timed systems, we can envisage more complicated, non-synchronous, compositions of systems.

Finally, it should be stated that since the key semantic and implementation issues of TransLucid are resolved, attention can be focused more on methodological and parallel implementation issues. We believe that programming with multidimensional, infinite data structures in an intensional manner allows programming to be viewed from a *completely* new perspective. This intuition will only be confirmed through experimentation with real problems, such as with multidimensional databases and simulations. With the right analysis, it should not be long before parallel programming becomes mainstream, with the programmer writing down the problem, and staying as far from the machine as possible, so that efficient code can be generated no matter the architecture on which the program happens to be running.

A glimpse of this is seen in Chapter 4, which presents a geometrical view of intensional programming, and it is here that the implications of Cartesian Programming can be seen. In TransLucid, a variable denotes an array, and many common programming problems can be solved by flattening the data structures used to represent the problem, and iterating over the resulting (non-hierarchical) arrays. By flattening out the data, and allowing every part of a computation to be indexed, we have made the data involved in a computation explicit. In fact this is the purpose of higher-order functions in TransLucid: to *structure* data. They are rarely used in the way that a language such as Haskell uses them, which is to describe computation. So, with the appropriate analysis, data and task parallelism should become apparent, allowing programs to be reasonably distributed across parallel computing networks.

For example, merge sort (§4.12.4, p.63), which is often implemented recursively, in TransLucid uses no functional recursion, but, instead, recurrence relations to define the variables used in its definition. It is this explicit definition of the structure of a merge sort that will allow an optimal implementation to be produced for it, including distribution over a parallel computing network. Furthermore, there is a vast body of work in the analysis of dependencies and the generation of iterative and distributed code from systems of recurrence equations (see [44, 19, 20, 10]). With the appropriate static analysis, it should not be difficult to implement such a scheme and produce incredibly fast code for evaluating TransLucid expressions on a wide variety of architectures. There is also a vast body of work in minimising the memory required to evaluate a system of recurrence equations (see [36, 6]), so despite the fact that the TransLucid solution creates an array of size $\Omega(n^2)$, an optimal implementation will only use $\Omega(n)$ memory. The key to all of this is to determine the rank of each variable in a program, and the dependencies between variables.

By examining the examples presented in Chapter 4, it can be seen that the infinite is still very relevant to Cartesian programming. Rather than limiting the length of intensions, programs can be defined much more naturally if one assumes that an intension is being defined for an infinite number of entries. Then, it is only the demands made of the system that limits what is computed. In fact, if the demands that will be made of a system can be determined ahead of time, or at least constrained, then only code relevant to the demands can be generated.

This should lead naturally to just-in-time compilation for TransLucid. The demand-driven computation of the current interpreter becomes demand-driven over regions, and then code specific to the computation of the requested regions can be generated on the fly. This way, parts of programs that are never demanded, never even have their code generated. In fact, key to generating efficient code will be the interaction between a completely static system, a dynamic system with just-in-time compilation, and the cache which ties them both together. An efficient implementation can be seen as a cache which knows something about the structure of a program in advance. If the hierarchy of dimensions required and the dependencies between variables can be determined in advance, caches with both more efficient memory layout and garbage collection schemes can be generated. An example of this is an array whose elements are dependent on the previous element, such as is the case when *fby* is used: to compute an element of such an array, only the previous element need be retained. In this case, the cache is a single element, and garbage collection throws out the old value immediately.

Chapter 7 presents a start on the question of static analysis, and is the most experimental chapter, by attempting to produce a type inference algorithm that is consistent with the idea that the principal type of an object is itself. This idea is, in fact, a fundamental idea in the design of TransLucid, and bestfitting (§8.4, p.152) is implemented with that in mind, and any future static analysis should also operate based on that principle. The type inference algorithm presented determines types by generating subtyping constraints, rather than unification requirements, as is standard in type-inferring functional languages. This allows the type of an object to be itself, rather than its atomic type, and for inferred

types to be more precise, by allowing specific union types, such as ranges over integers.

Any static analysis will necessarily not be complete with respect to the denotational semantics. Therefore, the goal for static analysis is to be as permissive as possible so that most *normal* programs pass, but to be strict enough that enough information is gathered to do something useful with it, such as generate faster code. The type inference chapter has left a large volume of future work. Rather than making TransLucid a static language, with a fixed type system, we have started with a completely dynamic system, and future work is to determine suitable constraints on programs so that static analysis works for reasonable cases, and can produce extremely efficient parallel code for most reasonable programs likely to be entered by the user.

Bibliography

[1] Data Parallel Haskell. `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`.

[2] Simon Peyton Jones—Haskell is useless. `http://youtube.com/watch/?v=iSmkqocn0oQ`.

[3] `Control.Concurrent`. `https://hackage.haskell.org/package/base-4.7.0.1/docs/Control-Concurrent.html`.

[4] `Control.Parallel`. `http://hackage.haskell.org/package/parallel-2.2.0.1/docs/Control-Parallel.html`.

[5] TransLucid source distribution. `http://github.com/jarro2783/TransLucid`.

[6] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. *SIGPLAN Not.*, 42:73–82, June 2007.

[7] E. A. Ashcroft and W. W. Wadge. Lucid—A formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3):336–354, September 1976.

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

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

[10] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Jarryd P. Beck, John Plaice, and William W. Wadge. Multidimensional Infinite Data in the Language Lucid. *Mathematics in Computer Science*, 25(2), April 2015. In press.

[12] Rudolph Carnap. *Meaning and Necessity*. University of Chicago Press, 1956. Seventh Impression, 1975.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

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

[15] 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.

[16] Anthony A. Faustini and Rangaswamy Jagannathan. Indexical Lucid. In *Proceedings of the 4th ISLIP*, pages 19–34, Menlo Park, USA, Apr 1991. SRI International. `http://plaice.web.cse.unsw.edu.au/archive/ISLIP/1991/islip1991_019.pdf`.

[17] Anthony A. Faustini and Rangaswamy Jagannathan. Multidimensional Problem Solving in Lucid. Technical Report SRI-CSL-93-03, Computer Science, SRI International, Menlo Park, USA, 1993. `http://www.csl.sri.com/papers/sri-csl-93-03/sri-csl-93-03.pdf`.

[18] Antony A. Faustini and William W. Wadge. An eductive interpreter for Lucid. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques, 1987, St. Paul, Minnesota, USA, June 24–26, 1987*, pages 86–91. ACM, 1987. `http://doi.acm.org/10.1145/29650.29659`.

[19] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem: Part I, One Dimensional Time. *International Journal of Parallel Programming*, 21(5), October 1992.

[20] Paul Feautrier. Some efficient solutions to the affine scheduling problem: Part II, Multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992. 10.1007/BF01379404.

[21] Stuart I. Feldman and Alan C. Kay. A conversation with Alan Kay. *ACM Queue*, 2(9):20–30, 2004. `http://queue.acm.org/detail.cfm?id=1039523`.

[22] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford, CA, USA, 1971. `http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf`.

[23] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. North-Holland, 1977.

[24] Blanca Mancilla. *Intensional Infrastructure for Collaborative Mapping*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2004. `http://mancilla.web.cse.unsw.edu.au/thesis/thesis.pdf`.

[25] Blanca Mancilla and John Plaice. Possible Worlds Versioning. *Mathematics in Computer Science*, 2(1):63–83, 2008.

[26] Calvin B. Ostrum. *The Luthid 1.0 Manual*. Department of Computer Science, University of Waterloo, Ontario, Canada, 1981.

[27] Joey Paquet. *Intensional Scientific Programming*. PhD thesis, Department of Computer Science, Laval University, Québec, Canada, April 1999.

[28] Joey Paquet and Peter G. Kropf. The GIPSY architecture. *Distributed Communities on the Web, LNCS*, 1830:144–153, 2000.

[29] Joey Paquet and John Plaice. The semantics of dimensions as values. In *Intensional Programming II*, pages 259–273, Singapore, 2000. World-Scientific.

[30] J. Plaice and W. W. Wadge. A new approach to version control. *IEEE Trans. Softw. Eng.*, 19(3):268–276, March 1993.

[31] John Plaice. Multidimensional Lucid: Design, Semantics and Implementation. *Distributed Communities on the Web, LNCS*, 1830:154–160, 2000.

[32] John Plaice. Cartesian Programming. University of Grenoble, France, 2010. HDR (Habilitation à diriger des recherches), `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/1101.pdf`.

[33] John Plaice, Jarryd P. Beck, and Blanca Mancilla. Declarative Multidimensional Spatial Programming. In *Proceedings of the 6th International Workshop on Spatial Computing*, pages 41–46, 2013. `http://www.spatial-computing.org/_media/scw13/corrected_prooceedings-workshop-9_spatial-computing.pdf`.

[34] François Pottier. Type inference in the presence of subtyping: From theory to practice. Research Report 3483, INRIA, September 1998. `http://hal.inria.fr/docs/00/07/32/05/PDF/RR-3483.pdf`.

[35] François Pottier. Subtyping-constraint-based type inference with conditional constraints: Algorithms and proofs, 1999. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.1625`.

[36] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.*, 22:773–815, September 2000.

[37] Toby Rahilly and John Plaice. A Multithreaded Implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1272–1277. IEEE Computer Society, 2008.

[38] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1978. `http://publications.csail.mit.edu/lcs/specpub.php?id=773`.

[39] P. Rondogiannis and W. W. Wadge. Higher-order functional languages and intensional logic. *J. Funct. Program.*, 9(5):527–564, September 1999.

[40] Adi Shamir and William W. Wadge. Data Types as Objects. In Arto Salomaa and Magnus Steinby, editors, *Automata, Languages and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 465–479, 1977. `http://dx.doi.org/10.1007/3-540-08342-1_36`.

[41] Paul Swoboda. Practical languages for intensional programming. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1999.

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

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

[44] Yoav Yaacoby and Peter. R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):115–125, 1989.

[45] Taner Yıldırım. Intensional HTML. Master's thesis, Department of Computer Science, University of Victoria, Canada, 1997.