Cartesian Programming: The TransLucid Programming Language

John Plaice and Blanca Mancilla

School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{plaice,mancilla}@cse.unsw.edu.au

Abstract. The TransLucid programming language is a low-level intensional language, designed to be sufficiently rich for it to be the target language for translating the common programming paradigms into it, while still being fully declarative. The objects manipulated by TransLucid, called hyperdatons, are arbitrary-dimensional infinite arrays, indexed by multidimensional tuples of arbitrary types.
We present the syntax, denotational and operational semantics for a simple TransLucid system, consisting of 1) a header detailing how expressions should be parsed, 2) a set of libraries of types, and operations thereon, defined in a host language, 3) a set of TransLucid equations, and 4) a TransLucid demand to be evaluated.
The evaluation of a demand for an (*identifier*, *context*) pair is undertaken using eduction, where previously computed pairs are stored in a cache called a warehouse. The execution ensures that only those dimensions actually encountered during the execution of an expression are taken into account when caching intermediate results.

Key words: Cartesian programming, Lucid language, declarative programming, multidimensional programming, context-aware programming, semantics.

1 Introduction

This paper presents the TransLucid programming language, in which variables define *hyperdatons*, infinite multidimensional arrays of arbitrary dimensionality, indexed by dynamically generated lazy tuples. The infinite nature of the hyperdatons allows the natural encoding of the set of possible states in an imperative language or the set of possible functions in a functional language; it is even possible to encode hyperdatons of functions, thereby providing a simple solution to adding higher-order functions to the Lucid programming language [6]. The lazy tuples — reminiscent of those of Linda [2] — and the declarative nature of the language ensure that an easily written, efficient, multithreaded implementation can be generated.

The multidimensional nature of the hyperdaton supports a *Cartesian* approach to computing. Descartes radically simplified geometry by giving it an

Dagstuhl Seminar Proceedings 08271
Topological and Game-Theoretic Aspects of Infinite Computations
http://drops.dagstuhl.de/opus/volltexte/2008/1654

algebraic basis; the simplicity of the coordinate system that he introduced made previously difficult problems trivial, and laid the foundations for all of modern mathematics and science. Similarly, the hyperdaton means that there is simply no further need to describe the "evolution" of a variable, either through space or time or with respect to some virtual dimensions, since one can simply consider the Cartesian product of all of the possible dimensions—including time and space—to create an index and then to demand the value of that variable at that point. With TransLucid, we are introducing *Cartesian Programming*.

The original version of TransLucid used eager tuples and was presented in [1]—we now call that language Eager TransLucid. The history of the development leading from the original Lucid to TransLucid is presented in [3]. The eductive implementation of Eager is given in [4]. The first multithreaded implementation of TransLucid is given in [5].

In this article, we develop the TransLucid language so that it can be used as a production language, covering the relationship between the concrete and abstract syntaxes and the denotational and operational semantics.

This article begins by presenting a simple TransLucid system (§2), consisting of a header (§3), libraries (§4), equations (§5) and a demand (§6). The section on equations presents both their concrete and abstract syntax. The denotational semantics (§7) defines the domains manipulated by TransLucid and defines how demands are interpreted. The operational semantics (§8) is defined in order to cache intermediate results, ensuring that only dimensions of relevance are stored in the cache. The conclusions (§9) discuss future work.

2 A simple TransLucid system

A *simple TransLucid system* is a quadruple $S = (H, L, Q, D)$, where

- H is a header, defining the arities and precedences of operators appearing in the equations, thereby allowing the translation of concrete syntax into abstract syntax;
- L ($\ni \ell$) is a set of libraries, defining the available types and functions;
- Q is a set of TransLucid equations;
- D is a demand to be executed.

3 The header

TransLucid is designed as a coordination language, which means that one can use types, constants and operators defined in another language, and then manipulate these. The header consists of a number of declarations defining how to parse expressions. In particular, the arity and precedence of operators are defined, as are the delimiters for user-defined types.

3.1 Infix operators

There are five kinds of declaration for infix binary or variadic operators. In each of the cases enumerated below, we are defining an operator symbol named *op* of precedence level n, which will be translated into an operator called *name* in the abstract syntax.

- `infixn`: infix non-associative binary operator.

  `infixn ustring<`*op*`> ustring<`*name*`> int<`*n*`> ;;`

- `infixl`: infix left-associative binary operator.

  `infixl ustring<`*op*`> ustring<`*name*`> int<`*n*`> ;;`

- `infixr`: infix right-associative binary operator.

  `infixr ustring<`*op*`> ustring<`*name*`> int<`*n*`> ;;`

- `infixm`: infix variadic operator.

  `infixm ustring<`*op*`> ustring<`*name*`> int<`*n*`> ;;`

  An example use of `infixm` would be for the set union operator, where we consider the union to be a single operation, as in:

  $$A \cup B \cup C \cup D \cup E \;=\; \bigcup\{A, B, C, D, E\}$$

- `infixp`: infix variadic comparison operator.

  `infixp ustring<`*op*`> ustring<`*name*`> int<`*n*`> ;;`

  An example use of `infixp` would be for the repeated $<$ operator, as in:

  $$A < B < C < D < E \;=\; (A < B) \wedge (B < C) \wedge (C < D) \wedge (D < E)$$

3.2 Unary operators

There are two kinds of unary operators, prefix and postfix.

- `prefix`: prefix unary operator.

  `prefix ustring<`*op*`> ustring<`*name*`> ;;`

  All prefix operators are assumed to have the same precedence level, which is higher than that of all the infix operators.
- `postfix`: postfix unary operator.

  `postfix ustring<`*op*`> ustring<`*name*`> ;;`

  All postfix operators are assumed to have the same precedence level, which is higher than that of all the prefix operators.

3.3 Typed constants

Normally a constant of type τ will be written τ`<`s`>` in the source code, where both τ and s are UTF-8–encoded strings. However, programs can become very difficult to read if there are many typed constants. We therefore provide specialised syntax for a default integer type, a default floating-point type, as well as for the parenthesisation of constants of given types.

- `defaultinttype`: default integer type.

  `defaultinttype type<`*name*`> ;;`

  defines the default integer type to be *name*. Integers of that type may simply appear as numbers. There may only be one such declaration in the header.
- `defaultfloattype`: default floating-point type.

  `defaultfloattype type<`*name*`> ;;`

  defines the default floating-point type to be *name*. Floating-point numbers of that type may simply appear as numbers. There may only be one such declaration in the header.
- `delimiters`: delimiters for a type.

  `delimiters type<`*name*`> uchar<`*left*`> uchar<`*right*`> ;;`

  states that constants of type *name* need simply be placed between the *left*...*right* pair. Common values for *left* and *right* would be brackets and quote characters of various kinds. Unicode contains many such pairs. For each type, there may only be one such declaration in the header.

3.4 Dimensions

In TransLucid, any ground value may be used as a dimension. It is also useful to have identifiers as dimensions.

- `dimension`: declare a dimension identifier.

  `dimension ustring<`*name*`> ;;`

  name may be used as dimension without it being evaluated.

3.5 Library

Finally, the header needs to provide an interface for learning about new kinds of type and operator, beyond the ones provided by default by any implementation. This is done as follows:

- `library`: declare a library of types and functions.

  `library ustring<`*name*`> ;;`

states that the library called *name* should be loaded. Libraries are defined in Section 4.

4 Libraries

A library ℓ defines the following information:

- *types*: These are ground data types. For each type, a context-dependent parse function and a context-dependent print function are defined. Therefore a library ℓ defines these two interfaces:

$$\ell_{\text{parse}} : \mathbf{Type} \to \mathbf{Str} \to \mathbf{Ctxt} \to \mathbf{Val}$$
$$\ell_{\text{print}} : \mathbf{Type} \to \mathbf{Val} \to \mathbf{Ctxt} \to \mathbf{Str}$$

- *operators*: An operator *name* is context-dependent and may be overloaded. Therefore, ℓ defines:

$$\ell_{\text{op}} : \mathbf{Str} \to \mathbf{Val}^{+} \to \mathbf{Ctxt} \to \mathbf{Val}$$

  For each type, an equality function must be defined.
- *type conversion operators*: These are used to cast typed values of one type to another. There is no implicit casting. Therefore, ℓ defines:

$$\ell_{\text{conv}} : \mathbf{Type} \to \mathbf{Val} \to \mathbf{Ctxt} \to \mathbf{Val}$$

5 Equations

We present the concrete syntax, then the abstract syntax, then the conversion from the former to the latter.

5.1 Concrete syntax

Here is the concrete syntax for equations (*eqn*) and expressions (*expr*):

eqn ::= *ident* `=` *expr* `;;`

expr ::= `(` *expr* `)`
| *const*
| *ident*
| *expr postfix*
| *prefix expr*
| *expr infix expr*
| *ident* `(` *expr* `,` ... `,` *expr* `)`
| `convert` `<`*ident*`>` *expr*
| `istype` `<`*ident*`>` *expr*
| `isspecial` `<`*str*`>` *expr*

| `if` *expr* `then` *expr* (`elsif` *expr* `then` *expr*)* `else` *expr* `fi`
| `#` *expr*
| *expr* `@` *expr*
| `[` *pair* `,` ... `,` *pair* `]`

pair ::= *expr* `:` *expr*

ident ::= `[a-zA-Z][a-zA-Z0-9_]*`

const ::= *ident* `<`*str*`>`
| *ldelim str rdelim*
| *int*
| *float*

int ::= `[0-9][a-zA-Z0-9]*`

float ::= *int*`.`*int*
| *int*`^[+-]?`*int*
| *int*`.`*int*`^[+-]?`*int*

5.2 Operator precedence

The unary operators bind with higher precedence than the binary and variadic operators. In increasing order, here is the precedence of all of the operators:

- '`!`', '`;`', binary, left-associative, least precedence;
- '`@`', binary, left-associative;
- infix, binary or variadic, of varying precedence and associativity;
- prefix, unary;
- postfix, unary;
- '`#`', unary, highest precedence.

Operators of the same precedence but different associativity cannot be used together without parentheses, nor can different variadic operators of the same precedence.

5.3 Abstract syntax

The abstract syntax for expressions is simpler than the concrete syntax.

$$\begin{aligned} E ::= {} & \texttt{id}\langle x \rangle \\ | \; & \texttt{const}\langle \tau, s \rangle \end{aligned}$$

$$
\begin{array}{l}
\mid\ \texttt{op}\langle s\rangle\ (E, \ldots, E) \\
\mid\ \texttt{convert}\langle \tau\rangle\ E \\
\mid\ \texttt{istype}\langle \tau\rangle\ E \\
\mid\ \texttt{isspecial}\langle v\rangle\ E \\
\mid\ \texttt{if}\ (E, E, E) \\
\mid\ \texttt{\#}\ E \\
\mid\ E\ \texttt{@}\ E \\
\mid\ \texttt{[}\ E\ \texttt{:}\ E,\ \ldots,\ E\ \texttt{:}\ E\ \texttt{]}
\end{array}
$$

Note that:

- The concrete *expr* becomes the abstract E.
- The concrete *ident* becomes the abstract *id*.
- The concrete *prefix*, *postfix* and *infix* all become the abstract *op*.
- The concrete `if`–`then`–`else`–`elsif`–`fi` all use the abstract `if`.

5.4 From concrete to abstract syntax

Given a set of equations Q and a header H, we write $\texttt{translate}(Q, H)$ for the conversion from concrete to abstract syntax of Q, taking into account the information provided by H. The process is straightforward, except for the handling of the infix operators. Since there are five different forms of associativity, and precedence levels can be any unsigned number, parsing these structures and creating the correct parse tree cannot be done using static tables. The following C++-style code will do the trick:

```
Expression*
infix_expr_build ( OperatorInfix* op,
                   Expression* left,
                   Expression* right )
  if (left->is_unary)
    return new ExpressionOpInfix (op, left, right);
  if (!left->is_infix)
    left->add_right (op, right);
    return left;
  // Now we know that we have two infix operators
  left_prec  = left->op->prec;
  left_assoc = left->op->assoc;
  if (left_prec < op->prec)
    left->add_right (op, right);
    return left;
  // Now we know that left_prec == op->prec
  if (left_assoc != op->assoc)
    throw "Parser error";
  if (left_assoc == ASSOC_NON)
```

```
    throw "Parser error";
  if (left_assoc == ASSOC_LEFT)
    return new ExpressionOpInfix (op, left, right);
  if (left_assoc == ASSOC_RIGHT)
    left->add_right (op, right);
    return left;
  // Now we know that assoc == ASSOC_VARIABLE or ASSOC_COMPARISON
  if (left_op != op)
    throw "Parser error";
  // Now we know that we have the same operator
  left->add_leaf (right);
  return left;
```

where `left->add_right` will replace the right-hand element of the left operand with an expression:

```
void
ExpressionOpInfix:add_right (OperatorInfix* op, Expression* right)
  args[1] = infix_expr_build (op, args[1], right);
```

and where `left->add_leaf` will add an additional argument to the node:

```
void
ExpressionOpInfix:add_leaf (Expression* right)
  args.push_back (right);
```

6 Demand

A demand is simply an expression to be evaluated.

7 Semantics

The semantics is standard, defined according to the structure of the expressions to be evaluated. What will be different will be the use of a dynamic context of evaluation. We begin by presenting some notation and the domains, then we give the semantics for evaluating expressions.

7.1 Notation for functions

We define some basic notation on functions ($\ni f, g', g$):

- The domain of a function f is written $\mathrm{dom}(f)$.
- If $c \notin \mathrm{dom}(f)$, we will write $f(c) = \bot$.

– If $\mathrm{dom}(f)$ is finite, then we may write f as:

$$f = \{c_{11} \mapsto c_{12},\ \ldots,\ c_{n1} \mapsto c_{n2}\}$$

meaning that:

$$\begin{aligned} f\,(c_{11}) &= c_{12} \\ &\ldots \\ f\,(c_{n1}) &= c_{n2} \end{aligned}$$

– If f and g are two functions, then $f \dagger g$ is the *perturbation* of f by g:

$$(f \dagger g)\,(c) = \begin{cases} g\,(c),\ c \in \mathrm{dom}(g) \\ f\,(c), \text{otherwise} \end{cases}$$

7.2 Domains

– **Type** $(\ni \tau)$ is the set of types found in the system. The set **Type** may vary, but must contain at least the types `sp`, `bool` and `tuple`. For each $\tau \in$ **Type**, the set of valid values for τ is written $V(\tau)$.
– **Value** $(\ni v)$ is the set of values found in the system.

$$\mathbf{Value} = \bigcup \{V(\tau) \mid \tau \in \mathbf{Type}\}$$

– **TypedValue** $\left(\ni \tau\langle v\rangle\right)$ is the set of properly typed values. It is a subset of **Type** $\times$ **Value**.

$$\mathbf{TypedValue} = \bigcup \{\tau\langle v\rangle \mid \tau \in \mathbf{Type} \wedge v \in V(\tau)\}$$

When we do not need to distinguish the type and value, we will write c.
– **Bool** $= V($`bool`$)$ is the set of Boolean values. The possible values are:

$$\mathbf{Bool} = \{\texttt{false}, \texttt{true}\}$$

– **Special** $= V($`sp`$)$ is the set of special values, to ensure that all operations in TransLucid are fully defined, no matter what the values of the passed arguments. The set must be partially ordered with the greatest lower bound property. In the current implementation, the following values are defined, in increasing order:

Value	Meaning
`undecl`	Undeclared identifier
`multidecl`	Multiply declared identifier
`undef`	Undefined definition
`multidef`	Multiple definition
`access`	Accessibility error
`loop`	Infinite loop
`dim`	Undefined dimension
`type`	Type error
`arith`	Arithmetic operation error
`string`	String operation error
`eod`	End of data

– **Tuple** $= V(\texttt{tuple\_t})$ $(\ni \kappa)$ is the set of tuples, used to hold contexts as well as complex types:

$$\textbf{Tuple} = \textbf{TypedValue} \to (\textbf{TypedValue} \mid \textbf{Demand})$$

The domain of a tuple (its *dimensions*) must always be computable, while the values associated with these dimensions need not yet be calculated. In the current implementation, the domain of tuples must always be finite.

– **Demand** is the set for *demands*, which encapsulate unevaluated expressions with their static and dynamic environments.

$$\textbf{Demand} = \textbf{System} \times \textbf{Ctxt} \times \textbf{Expr}$$

A demand is written $\texttt{demand}\langle \xi, \kappa \rangle E$, where $\xi \in \textbf{System}$, $\kappa \in \textbf{Tuple}$ and $E \in \textbf{Expr}$.

– **SimpleSystem** $(\ni \xi)$ is the semantic counterpart of a TransLucid simple system S (Section 2). A system ξ contains the following components:

$$\begin{aligned}
\xi_{\text{parse}} &: \textbf{Type} \to \textbf{Str} \to \textbf{Ctxt} \to \textbf{Val} \\
\xi_{\text{print}} &: \textbf{Type} \to \textbf{Val} \to \textbf{Ctxt} \to \textbf{Str} \\
\xi_{\text{op}} &: \textbf{Str} \to \textbf{Val}^{+} \to \textbf{Ctxt} \to \textbf{Val} \\
\xi_{\text{conv}} &: \textbf{Type} \to \textbf{Val} \to \textbf{Ctxt} \to \textbf{Val} \\
\xi_{\text{eqn}} &: \textbf{Str} \to \textbf{Expr}
\end{aligned}$$

The system ξ defined from $S = (H, L, Q, D)$ is given by:

$$\begin{aligned}
\xi_{\text{parse}} &= \bigcup_{\ell \in L} \ell_{\text{parse}} \\
\xi_{\text{print}} &= \bigcup_{\ell \in L} \ell_{\text{print}} \\
\xi_{\text{op}} &= \bigcup_{\ell \in L} \ell_{\text{op}} \\
\xi_{\text{conv}} &= \bigcup_{\ell \in L} \ell_{\text{conv}} \\
\xi_{\text{eqn}} &= \texttt{translate}(Q, H) \\
\xi_{\text{demand}} &= \texttt{translate}(D, H)
\end{aligned}$$

7.3 Expressions

The evaluation rules for expressions, given below, are of the form:

$$[\![E]\!]\xi\kappa$$

which means that given a system ξ and a context κ, expression E evaluates to a typed value $c = \tau\langle v \rangle$.

Conventions

– If in the right-hand part of a rule, there is an occurrence of τ_α, then this type can be calculated through the following convention:

$$\tau_\alpha\langle v_\alpha\rangle = \mathtt{eval}_1\big([\![E_\alpha]\!]\xi\kappa\big)$$

where:

$$\begin{aligned}\mathtt{eval}_1\big(\tau\langle v\rangle\big) &= \tau\langle v\rangle\\ \mathtt{eval}_1\big(\mathtt{demand}\langle\xi,\kappa\rangle E\big) &= [\![E]\!]\xi\kappa\end{aligned}$$

– If in the right-hand part of a rule, there is an occurrence of c_α, then the constant must be fully evaluated, as is given by:

$$c_\alpha = \tau_\alpha\langle v_\alpha\rangle = \mathtt{eval}\big([\![E_\alpha]\!]\xi\kappa\big)$$

where:

$$\begin{aligned}\mathtt{eval}\big(\tau\langle v\rangle\big) &= \tau\langle v\rangle, \qquad \tau \neq \mathtt{tuple}\\ \mathtt{eval}\big(\mathtt{tuple}\langle c_i \mapsto \ell_i\rangle\big) &= \mathtt{tuple}\big\langle c_i \mapsto \mathtt{eval}(\ell_i)\big\rangle\\ \mathtt{eval}\big(\mathtt{demand}\langle\xi,\kappa\rangle E\big) &= \mathtt{eval}\big([\![E]\!]\xi\kappa\big)\end{aligned}$$

– In the rules below, i and j take on the values from 1 to n.

Rules

$$\begin{aligned}[\![\mathtt{id}\langle s\rangle]\!]\xi\kappa &= \xi_{\mathrm{eqn}}(s)(\xi)(\kappa)\\ [\![\mathtt{const}\langle\tau, s\rangle]\!]\xi\kappa &= \xi_{\mathrm{parse}}(\tau)(s)(\kappa)\\ [\![\mathtt{op}\langle s\rangle(E_i)]\!]\xi\kappa &= \begin{cases}\min\{v_j \mid \tau_j = \mathtt{sp}\}, & \exists j, \tau_j = \mathtt{sp}\\ \xi_{\mathrm{op}}(s)(c_1,\ldots,c_n)(\kappa), \text{ otherwise}\end{cases}\\ [\![\mathtt{convert}\langle\tau\rangle E_1]\!]\xi\kappa &= \begin{cases}c_1, & \tau_1 = \mathtt{sp}\\ \xi_{\mathrm{conv}}(\tau)(c_1)(\kappa), \text{ otherwise}\end{cases}\\ [\![\mathtt{istype}\langle\tau\rangle E_1]\!]\xi\kappa &= \begin{cases}\mathtt{bool}\langle\mathtt{true}\rangle, & \tau_1 = \tau\\ \mathtt{bool}\langle\mathtt{false}\rangle, \text{ otherwise}\end{cases}\\ [\![\mathtt{isspecial}\langle v\rangle E_1]\!]\xi\kappa &= \begin{cases}\mathtt{bool}\langle\mathtt{true}\rangle, & c_1 = \mathtt{sp}\langle v\rangle\\ \mathtt{bool}\langle\mathtt{false}\rangle, \text{ otherwise}\end{cases}\\ [\![\mathtt{if}\ (E_1, E_2, E_3)]\!]\xi\kappa &= \begin{cases}c_1, & \tau_1 = \mathtt{sp}\\ [\![E_2]\!]\xi\kappa, & c_1 = \mathtt{bool}\langle\mathtt{true}\rangle\\ [\![E_3]\!]\xi\kappa, & c_1 = \mathtt{bool}\langle\mathtt{false}\rangle\\ \mathtt{sp}\langle\mathtt{type}\rangle, \text{ otherwise}\end{cases}\\ [\![\#E_1]\!]\xi\kappa &= \begin{cases}c_1, & \tau_1 = \mathtt{sp}\\ \kappa(c_1), & c_1 \in \mathrm{dom}\ \kappa\\ \mathtt{sp}\langle\mathtt{dim}\rangle, \text{ otherwise}\end{cases}\end{aligned}$$

$$[\![E_2 \texttt{ @ } E_1]\!]\xi\kappa = \begin{cases} c_1, & \tau_1 = \texttt{sp} \\ \texttt{sp}\langle\texttt{type}\rangle, & \tau_1 \neq \texttt{tuple} \\ \texttt{sp}\langle\texttt{access}\rangle, & \neg\text{accessible}\,(\kappa', \kappa) \\ [\![E_2]\!]\xi(\kappa \dagger v_1), & \text{otherwise} \end{cases}$$

$$[\![\texttt{[}E_{i1} \texttt{ : } E_{i2}\texttt{]}]\!]\xi\kappa = \begin{cases} \min\{v_{j1} \mid \tau_{j1} = \texttt{sp}\}, & \exists j, \tau_{j1} = \texttt{sp} \\ \texttt{tuple}\langle c_{i1} \mapsto \texttt{demand}\langle\xi, \kappa\rangle E_{i2}\rangle, & \text{otherwise} \end{cases}$$

The line accessible (κ', κ) refers to the possibility of moving from context κ to κ'. By default, this is always true; in situations where contexts may have physical interpretations, then this relation will be more complex.

7.4 Demands

The semantics of a system ξ is given by evaluating the demand therein and then by printing out the result:

$$\begin{array}{l} \text{let } c = \tau\langle v\rangle = [\![\, \xi_{\text{demand}} \,]\!]\,(\xi)\,(\emptyset) \\ \text{in } \xi_{\text{print}}(\tau)(c)(\kappa) \end{array}$$

8 Operational Semantics

The operational semantics are designed to cache intermediate results for each demand for the calculation of an (*identifier*, *context*) pair (x, κ). However, it is often the case that the current context includes information about dimensions that are not needed for the calculation of a particular expression. Therefore, it is necessary to keep track of a *hierarchy* of dimensions, which is a list of sets of dimensions.

A hierarchy $\mathcal{H}$ is written $\mathcal{H} = \langle C_0, \ldots, C_{n-1}\rangle$, where each C_i is a set of dimensions. When $\mathcal{H}$ appears in a set of rules, it means that to evaluate an expression, first all of the dimensions in C_0 need to be known. Once these dimensions are known, then the dimensions in C_1 need to be known, and so on.

Hierarchies are used to build *warehouses*. A warehouse $\mathcal{W}$ is a function:

$$\mathcal{W} : \mathbf{Id} \times \mathbf{Ctxt} \to \mathbf{Val}^+ \cup \mathbf{Val}$$

When the pair (x, κ) is being executed to produce a value c, a hierarchy $\mathcal{H} = \langle C_0, \ldots, C_{n-1}\rangle$ will be built. After adding these entries to warehouse W, the following will hold:

$$\begin{aligned} \mathcal{W}(x, \emptyset) &= C_0? \\ \mathcal{W}(x, \kappa \mid C_0) &= C_1? \\ \mathcal{W}\big(x, \kappa \mid (C_0 \cup C_1)\big) &= C_2? \\ &\cdots \\ \mathcal{W}\big(x, \kappa \mid (C_0 \cup \cdots \cup C_{n-2})\big) &= C_{n-1}? \\ \mathcal{W}\big(x, \kappa \mid (C_0 \cup \cdots \cup C_{n-1})\big) &= c \end{aligned}$$

To simplify the building of hierarchies, the operational semantics rules maintain a stack—a list—of contexts, built up through the successive use of the `@` operator. Rather than perturbing the current context, the use of the stack makes it easier to keep track of the hierarchies being built.

8.1 Basic operations

Here we define operations on hierarchies. The ':' is the "cons" operator, and the '$\langle\rangle$' is the empty list.

$$\begin{aligned}
\texttt{restrict}\big(\langle\rangle, C\big) &= \langle\rangle \\
\texttt{restrict}(C_0 : \mathcal{H}, C) &= \begin{cases} \texttt{restrict}(\mathcal{H}, C), & C_0 - C = \emptyset \\ (C_0 - C) : \texttt{restrict}(\mathcal{H}, C), & \text{otherwise} \end{cases} \\
\\
\texttt{merge}\big(\mathcal{H}, \langle\rangle\big) &= \mathcal{H} \\
\texttt{merge}\big(\langle\rangle, \mathcal{H}'\big) &= \mathcal{H}' \\
\texttt{merge}(C_0 : \mathcal{H}, C_0' : \mathcal{H}') &= (C_0 \cup C_0') : \texttt{merge}\big(\texttt{restrict}(\mathcal{H}, C_0'), \texttt{restrict}(\mathcal{H}', C_0)\big) \\
\\
\texttt{collapse}\big(\langle\rangle\big) &= \emptyset \\
\texttt{collapse}(C_0 : \mathcal{H}) &= C_0 \cup \texttt{collapse}(\mathcal{H}) \\
\\
\texttt{add}(\mathcal{H}, \mathcal{H}') &= \texttt{append}\Big(\mathcal{H}, \texttt{restrict}\big(\mathcal{H}', \texttt{collapse}(\mathcal{H})\big)\Big) \\
\\
\texttt{in}\big(\langle\rangle, c\big) &= \mathit{false} \\
\texttt{in}(C_0 : \mathcal{H}, c) &= \begin{cases} \mathit{true}, & c \in C_0 \\ \texttt{in}(\mathcal{H}, c), & \text{otherwise} \end{cases} \\
\\
\texttt{addone}(\mathcal{H}, c) &= \begin{cases} \mathcal{H}, & \texttt{in}(\mathcal{H}, c) = \mathit{true} \\ \texttt{add}\Big(\mathcal{H}, \big\langle\{c\}\big\rangle\Big), & \text{otherwise} \end{cases}
\end{aligned}$$

8.2 Rules

The operational semantics rules are of the form:

$$\mathcal{K}, \mathcal{W} \vdash E : c, \mathcal{H}', \mathcal{K}', \mathcal{W}'$$

where:

- E is the expression being evaluated.
- c is the calculated value.

– $\mathcal{H}'$ is the dependency hierarchy built while evaluating E.
– $\mathcal{K}$ and $\mathcal{K}'$ are the before and after states of a *context stack*, which is simply a list of partially evaluated contexts, i.e., with demands in the right-hand sides of entries. Each time that an `@` is encountered, the context stack grows. The difference between $\mathcal{K}'$ and $\mathcal{K}$ is that $\mathcal{K}'$ will contain more evaluated right-hand sides. The `mergerhs` and `evalrhs` operators are straightforward.
– $\mathcal{W}$ and $\mathcal{W}'$ are the before and after states of a *warehouse*. The difference between $\mathcal{W}'$ and $\mathcal{W}$ is that $\mathcal{W}'$ will contain more entries. The `mergewarehouses` and `addtowarehouse` operators are straightforward.

To simplify the presentation of the rules below, we will assume that $\xi_{\texttt{parse}}$, $\xi_{\texttt{op}}$ and $\xi_{\texttt{conv}}$ are not context-dependent. We will also not consider the handling of special cases and values. Adding these features is straightforward.

$$\frac{\begin{array}{c}E = \xi_{\texttt{eqn}}(s) \\ \mathcal{W}, \mathcal{K} \vdash E : c, \mathcal{H}, \mathcal{K}', \mathcal{W}' \\ \mathcal{W}'' = \texttt{addtowarehouse}(\mathcal{W}', s, c, \mathcal{H}, \mathcal{K}')\end{array}}{\mathcal{W}, \mathcal{K} \vdash \texttt{id}\langle s\rangle : c, \mathcal{H}, \mathcal{K}', \mathcal{W}''}$$

$$\frac{c = \xi_{\texttt{parse}}(\tau)(s)}{\mathcal{K}, \mathcal{W} \vdash \texttt{const}\langle \tau, s\rangle : c, \langle\rangle, \mathcal{K}, \mathcal{W}}$$

$$\frac{\begin{array}{c}\mathcal{K}, \mathcal{W} \vdash E_i : c_i, \mathcal{H}_i, \mathcal{K}_i, \mathcal{W}_i \\ c = \xi_{\texttt{op}}(s)(E_i) \\ \mathcal{H}' = \texttt{merge}(\mathcal{H}_i) \\ \mathcal{K}' = \texttt{mergerhs}(\mathcal{K}_i) \\ \mathcal{W}' = \texttt{mergewarehouses}(\mathcal{W}_i)\end{array}}{\mathcal{K}, \mathcal{W} \vdash \texttt{op}\langle s\rangle(E_i) : c, \mathcal{H}', \mathcal{K}', \mathcal{W}'}$$

$$\frac{\begin{array}{c}\mathcal{K}, \mathcal{W} \vdash E_1 : c_1, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ c = \xi_{\texttt{conv}}(\tau)(c_1)\end{array}}{\mathcal{K}, \mathcal{W} \vdash \texttt{convert}\langle \tau\rangle\ E_1 : c, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1}$$

$$\frac{\begin{array}{c}\mathcal{K}, \mathcal{W} \vdash E_1 : \tau_1\langle v_1\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ \tau = \tau_1\end{array}}{\mathcal{K}, \mathcal{W} \vdash \texttt{istype}\langle \tau\rangle\ E_1 : \texttt{bool}\langle\texttt{true}\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1}$$

$$\frac{\begin{array}{c}\mathcal{K}, \mathcal{W} \vdash E_1 : \tau_1\langle v_1\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ \tau \neq \tau_1\end{array}}{\mathcal{K}, \mathcal{W} \vdash \texttt{istype}\langle \tau\rangle\ E_1 : \texttt{bool}\langle\texttt{false}\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1}$$

$$\frac{\begin{array}{c}\mathcal{K},\mathcal{W} \vdash E_1 : \texttt{bool}\langle\texttt{true}\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ \mathcal{K}_1, \mathcal{W}_1 \vdash E_2 : c_2, \mathcal{H}_2, \mathcal{K}_2, \mathcal{W}_2 \\ \mathcal{H}' = \texttt{add}(\mathcal{H}_1, \mathcal{H}_2) \\ \mathcal{K}' = \texttt{mergerhs}(\mathcal{K}_1, \mathcal{K}_2) \\ \mathcal{W}' = \texttt{mergewarehouses}(\mathcal{W}_1, \mathcal{W}_2)\end{array}}{\mathcal{K},\mathcal{W} \vdash \texttt{if}(E_1, E_2, E_3) : c_2, \mathcal{H}', \mathcal{K}', \mathcal{W}'}$$

$$\frac{\begin{array}{c}\mathcal{K},\mathcal{W} \vdash E_1 : \texttt{bool}\langle\texttt{false}\rangle, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ \mathcal{K}_1, \mathcal{W}_1 \vdash E_3 : c_3, \mathcal{H}_3, \mathcal{K}_3, \mathcal{W}_3 \\ \mathcal{H}' = \texttt{add}(\mathcal{H}_1, \mathcal{H}_3) \\ \mathcal{K}' = \texttt{mergerhs}(\mathcal{K}_1, \mathcal{K}_3) \\ \mathcal{W}' = \texttt{mergewarehouses}(\mathcal{W}_1, \mathcal{W}_3)\end{array}}{\mathcal{W},\mathcal{K} \vdash \texttt{if}(E_1, E_2, E_3) : c_3, \mathcal{H}', \mathcal{K}', \mathcal{W}'}$$

$$\frac{\begin{array}{c}\mathcal{K},\mathcal{W} \vdash E : c, \mathcal{H}, \mathcal{K}', \mathcal{W}' \\ \kappa = \texttt{find}(\mathcal{K}', c) \\ \kappa' = \texttt{evalrhs}(\kappa, c) \\ \mathcal{K}'' = \texttt{replace}(\mathcal{K}', \kappa, \kappa')\end{array}}{\mathcal{K},\mathcal{W} \vdash \texttt{\#}E : \kappa'(c), \texttt{addone}(\mathcal{H}, c), \mathcal{K}'', \mathcal{W}'}$$

$$\frac{\begin{array}{c}\mathcal{K},\mathcal{W} \vdash E_1 : \kappa_1, \mathcal{H}_1, \mathcal{K}_1, \mathcal{W}_1 \\ \kappa_1 : \mathcal{K}_1, \mathcal{W}_1 \vdash E_2 : c_2, \mathcal{H}_2, \kappa_2 : \mathcal{K}_2, \mathcal{W}_2\end{array}}{\mathcal{W},\mathcal{K} \vdash E_2 \texttt{@} E_1 : c, \texttt{add}\big(\mathcal{H}_1, \texttt{restrict}(\mathcal{H}_2, \operatorname{dom} \kappa_1)\big), \mathcal{K}_2, \mathcal{W}_2}$$

$$\frac{\begin{array}{c}\mathcal{K},\mathcal{W} \vdash E_{i1} : c_i, \mathcal{H}_i, \mathcal{K}_i, \mathcal{W}_i \\ E'_{i2} = \texttt{demand}\langle\xi, \mathcal{K}\rangle\ E_{i2} \\ \mathcal{H}' = \texttt{merge}(\mathcal{H}_i) \\ \mathcal{K}' = \texttt{mergerhs}(\mathcal{K}_i) \\ \mathcal{W}' = \texttt{mergewarehouses}(\mathcal{W}_i)\end{array}}{\mathcal{K},\mathcal{W} \vdash [E_{i1} : E_{i2}] : [c_i : E'_{i2}], \mathcal{H}', \mathcal{K}', \mathcal{W}'}$$

The above rules can naturally be transformed into an efficient system for demand-driven evaluation, using a sequential or a multi-threaded approach.

9 Conclusions

We have presented a simple TransLucid system, and given an outline of the concrete and abstract syntaxes as well as the denotational and operational semantics. The current TransLucid interpreter implements the language as it is defined in this paper.

Future work involves transforming TransLucid into a reactive system, in which the set of equations evolves over time, through the use of a `time` dimension, and through the use of multiple *threads*, each making demands of the reactive system at each instant.

Envisaged applications of TransLucid are the development of Cartesian languages for functional and imperative programming, using TransLucid as implementation target.

References

1. Gabriel Ditu. *The Programming Language TransLucid*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, March 2007.
2. David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
3. John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and Cartesian programming. *Journal of Mathematics in Computer Science*, In press. 2008.
4. John Plaice, Blanca Mancilla, Gabriel Ditu, and William W. Wadge. Sequential demand-driven evaluation of Eager TransLucid. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 1266–1271, Turku, Finland, 28 July – 1 August 2008.
5. Toby Rahilly and John Plaice. A multithreaded implementation for TransLucid. In *32nd Annual IEEE International Computer Software and Applications Conference*, pages 1272–1277, Turku, Finland, 28 July – 1 August 2008.
6. William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.