Cartesian Programming:
The TransLucid Programming Language

John Plaice     Blanca Mancilla
School of Computer Science and Engineering
University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{plaice,mancilla}@cse.unsw.edu.au
2009

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 \((\text{identifier}, \text{context})\) 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.

Keywords: 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 [7]. 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 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 [4]. The eductive implementation of Eager TransLucid is given in [5].

The first multithreaded implementation of TransLucid is given in [6]. The TransLucid language is now being developed into a production language, allowing the use of user-defined atomic data types and operations.

The main contribution of this article is to further develop the eductive implementation of TransLucid, which caches intermediate results, ensuring that only dimensions of relevance are stored in the cache. The solution offered here ensures that this minimality is preserved when additional variables are introduced and when lazy tuples are used. Furthermore, the given rules can be transformed into a variety of physical architectures.

This article begins by presenting a small TransLucid example (§2). We continue with the presentation of the syntax for a simple TransLucid system (§3), consisting of a header, libraries, equations and a demand. The denotational semantics (§4) defines the domains manipulated by TransLucid and defines how demands are interpreted. The operational semantics (§5) forms the core of the paper. The conclusions (§6) discuss future work.

## 2 TransLucid example

We show a simple division of two floating-point numbers, where the division uses a higher precision than the original precision of the two numbers.

```plaintext
// Declare for the parser
// the precedence of +.
infixl "+" "operator+" 1;;
%%
// x and y are 64-bit precision
// GNU mp floating-point numbers.
x = floatmp<1.23> ;
y = floatmp<4.56> ;
// The division uses 128-bit precision.
z = (x/y) @ [precision:128] ;
%%
// The result is 64-bit precision.
z @ [precision:64] ;
```

This kind of manipulation of parameters takes place in a huge variety of situations, and it is common for the set of available parameters to be far larger than the set of parameters needed for the current needs of a running program. Examples include the use of Unix-variant environment variables, of fields in C++ templated class instantiations, or of individual sensors in a sensor bank.

In the denotational and operational semantics given below, the focus is on ensuring that no parameters are evaluated if not needed by the equations.

## 3 A simple TransLucid system

A simple TransLucid system $S = (H, L, Q, D)$ is a quadruple 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; and
- $D$ is a demand to be executed.
3.1 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. We do not give the details in this article, as we are focusing on the operational semantics.

3.2 Libraries

A library $\ell$ 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 $\ell$ defines these two interfaces:

\[
\ell_{\text{parse}} : \text{Type} \to \text{Str} \to \text{Ctxt} \to \text{Val} \\
\ell_{\text{print}} : \text{Type} \to \text{Val} \to \text{Ctxt} \to \text{Str}
\]

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

\[
\ell_{\text{op}} : \text{Str} \to \text{Val}^+ \to \text{Ctxt} \to \text{Val}
\]

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, $\ell$ defines:

\[
\ell_{\text{conv}} : \text{Type} \to \text{Val} \to \text{Ctxt} \to \text{Val}
\]

3.3 Expressions

Here is the abstract syntax for expressions:

\[
E ::= \text{id}(x) \\
| \text{const}(\tau, s) \\
| \text{op}(s)(E, \ldots, E) \\
| \text{convert}(\tau) E \\
| \text{istype}(\tau) E \\
| \text{isspecial}(v) E \\
| \text{if}(E, E, E) \\
| \# E \\
| E @[E] \\
| [E : E, \ldots, E : E]
\]

where

- $s$ is a string;
- $x$ is an identifier;
- $\tau$ is a type; and
- $v$ is a special value, defined below.
3.4 Demand

A demand is simply an expression to be evaluated.

4 Semantics

The semantics is defined in a standard way, 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.

4.1 Notation for functions

We define some basic notation on functions (∈ f, g′, g):

The domain of a function f is written dom(f).
If c ∉ dom(f), we will write f(c) = ⊥.
If dom(f) is finite, then we may write f as:

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

meaning that:

\[ f(c_{11}) = c_{12} \]
\[ \ldots \]
\[ f(c_{n1}) = c_{n2} \]

If f and g are two functions, then \( f \uparrow g \) is the perturbation of f by g:

\[ (f \uparrow g)(c) = \begin{cases} g(c), & c \in \text{dom}(g) \\ f(c), & \text{otherwise} \end{cases} \]

4.2 Domains

Type (∈ τ) 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 \( \tau \in \text{Type} \), the set of valid values for \( \tau \) is written V(τ).

Value (∈ v) is the set of values found in the system.

\[ \text{Value} = \bigcup \{ V(\tau) \mid \tau \in \text{Type} \} \]

TypedValue (∈ τ⟨v⟩) is the set of properly typed values. It is a subset of Type × Value.

\[ \text{TypedValue} = \bigcup \{ \tau(\langle v \rangle) \mid \tau \in \text{Type} \wedge v \in V(\tau) \} \]

When we do not need to distinguish the type and value, we will write c (for constant).

Bool = V(bool) is the set of Boolean values. The possible values are:

\[ \text{Bool} = \{ \text{false}, \text{true} \} \]

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>undecl</td>
<td>Undeclared identifier</td>
</tr>
<tr>
<td>multidecl</td>
<td>Multiply declared identifier</td>
</tr>
<tr>
<td>undef</td>
<td>Undefined definition</td>
</tr>
<tr>
<td>multidef</td>
<td>Multiple definition</td>
</tr>
<tr>
<td>access</td>
<td>Accessibility error</td>
</tr>
<tr>
<td>loop</td>
<td>Infinite loop</td>
</tr>
<tr>
<td>dim</td>
<td>Undefined dimension</td>
</tr>
<tr>
<td>type</td>
<td>Type error</td>
</tr>
<tr>
<td>arith</td>
<td>Arithmetic operation error</td>
</tr>
<tr>
<td>string</td>
<td>String operation error</td>
</tr>
<tr>
<td>eod</td>
<td>End of data</td>
</tr>
</tbody>
</table>

\[
\text{Tuple} = V(\text{tuple}\_t) (\ni \kappa)
\]

is the set of tuples, used to hold contexts as well as complex types:

\[
\text{Tuple} = \text{TypedValue} \to (\text{TypedValue} \mid \text{Demand})
\]

The domain of a tuple (its \emph{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.

\textbf{Demand} is the set for \emph{demands}, which encapsulate unevaluated expressions with their static and dynamic environments.

\[
\text{Demand} = \text{System} \times \text{Ctxt} \times \text{Expr}
\]

A demand is written \emph{demand}(\xi, \kappa)E, where \(\xi \in \text{System}, \ \kappa \in \text{Tuple}\) and \(E \in \text{Expr}\).

\textbf{SimpleSystem} (\ni \xi) is the semantic counterpart of a TransLucid simple system \(S\) (Section 3).

A system \(\xi\) contains the following components:

\[
\begin{align*}
\xi_{\text{parse}} : & \text{Type} \to \text{Str} \to \text{Ctxt} \to \text{Val} \\
\xi_{\text{print}} : & \text{Type} \to \text{Val} \to \text{Ctxt} \to \text{Str} \\
\xi_{\text{op}} : & \text{Str} \to \text{Val}^+ \to \text{Ctxt} \to \text{Val} \\
\xi_{\text{conv}} : & \text{Type} \to \text{Val} \to \text{Ctxt} \to \text{Val} \\
\xi_{\text{eqn}} : & \text{Str} \to \text{Expr}
\end{align*}
\]

The system \(\xi\) derived from \(S = (H, L, Q, D)\) is:

\[
\begin{align*}
\xi_{\text{parse}} &= \bigcup_{\ell \in L} \ell_{\text{parse}} \\
\xi_{\text{print}} &= \bigcup_{\ell \in L} \ell_{\text{print}} \\
\xi_{\text{op}} &= \bigcup_{\ell \in L} \ell_{\text{op}} \\
\xi_{\text{conv}} &= \bigcup_{\ell \in L} \ell_{\text{conv}} \\
\xi_{\text{eqn}} &= \text{translate}(Q, H) \\
\xi_{\text{demand}} &= \text{translate}(D, H)
\end{align*}
\]

(\text{translate} converts concrete into abstract syntax).
4.3 Expressions

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

\[ [E] \xi \kappa \]

which means that given a system \( \xi \) and a context \( \kappa \), expression \( E \) evaluates to a typed value \( c = \tau \langle v \rangle \).

4.3.1 Conventions

If in the right-hand part of a rule, there is an occurrence of \( \tau_{\alpha} \), then this type can be calculated through the following convention:

\[ \tau_{\alpha}\langle v_{\alpha} \rangle = \text{eval}_{1}(\{ [E_{\alpha}] \xi \kappa \}) \]

where:

\[ \text{eval}_{1}(\tau \langle v \rangle) = \tau \langle v \rangle \]
\[ \text{eval}_{1}(\text{demand}(\xi, \kappa)E) = [E] \xi \kappa \]

If in the right-hand part of a rule, there is an occurrence of \( c_{\alpha} \), then the constant must be fully evaluated, as given by:

\[ c_{\alpha} = \tau_{\alpha}\langle v_{\alpha} \rangle = \text{eval}([E_{\alpha}] \xi \kappa) \]

where:

\[ \text{eval}(\tau \langle v \rangle) = \tau \langle v \rangle, \quad \tau \neq \text{tuple} \]
\[ \text{eval}(\text{tuple}(c_{i} \mapsto \ell_{i})) = \text{tuple}(c_{i} \mapsto \text{eval}(\ell_{i})) \]
\[ \text{eval}(\text{demand}(\xi, \kappa)E) = \text{eval}([E] \xi \kappa) \]

In any rule, \( i \) and \( j \) take on the values from 1 to \( n \).
4.3.2 Rules

\[
\begin{align*}
[\text{id}(s)]\xi\kappa &= \xi_{\text{eqn}}(s)(\xi)(\kappa) \\
[\text{const}(\tau,s)]\xi\kappa &= \xi_{\text{parse}}(\tau)(s)(\kappa) \\
[\text{op}(s)(E_i)]\xi\kappa &= \begin{cases} 
\min\{v_j \mid \tau_j = \text{sp}\}, & \exists j, \tau_j = \text{sp} \\
\xi_{\text{op}}(s)(c_1, \ldots, c_n)(\kappa), & \text{otherwise}
\end{cases} \\
[\text{convert}(\tau)E_i]\xi\kappa &= \begin{cases} 
\text{bool}(\text{true}), & \tau_1 = \tau \\
\text{bool}(\text{false}), & \text{otherwise}
\end{cases} \\
[\text{istype}(\tau)E_i]\xi\kappa &= \begin{cases} 
\text{bool}(\text{true}), & c_1 = \text{sp}(\tau) \\
\text{bool}(\text{false}), & \text{otherwise}
\end{cases} \\
[\text{isspecial}(v)E_i]\xi\kappa &= \begin{cases} 
\text{bool}(\text{false}), & \text{otherwise}
\end{cases} \\
[\text{if}(E_1, E_2, E_3)]\xi\kappa &= \begin{cases} 
c_1, & \tau_1 = \text{sp} \\
E_2\xi\kappa, & c_1 = \text{bool}(\text{true}) \\
E_3\xi\kappa, & c_1 = \text{bool}(\text{false}) \\
\text{sp}(\text{type}), & \text{otherwise}
\end{cases} \\
[#E_i]\xi\kappa &= \begin{cases} 
c_1, & \tau_1 = \text{sp} \\
c(\kappa), & c_1 \in \text{dom} \kappa \\
\text{sp}(\text{dim}), & \text{otherwise}
\end{cases} \\
[E_2 \circ E_1]\xi\kappa &= \begin{cases} 
c_1, & \tau_1 = \text{sp} \\
\text{sp}(\text{type}), & \tau_1 \neq \text{tuple} \\
\text{sp}(\text{access}), & \neg\text{accessible}(\kappa', \kappa) \\
E_2[\xi(\kappa \uparrow v_1)], & \text{otherwise}
\end{cases} \\
[[E_1; E_2]]\xi\kappa &= \begin{cases} 
\min\{v_j \mid \tau_j = \text{sp}\}, & \exists j, \tau_j = \text{sp} \\
\text{tuple}(c_1 \mapsto \text{demand}(\xi, \kappa)E_{i2}), & \text{otherwise}
\end{cases}
\end{align*}
\]

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

4.4 Demands

The semantics of a system \(\xi\) is given by evaluating the demand therein and then by printing out the result:

\[
\text{let } c = \tau(v) = \left[ \xi_{\text{demand}} \right](\xi)(\emptyset)
\text{in } \xi_{\text{print}}(\tau)(c)(\kappa)
\]

5 Operational Semantics

The operational semantics are designed to cache intermediate results for each demand for the calculation of an \((\text{identifier}, \text{context})\) pair \((x, \kappa)\). 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.

When the pair \( (x, \kappa) \) is being executed to produce a value \( c \), a hierarchy \( H = \langle C_0, \ldots, C_{n-1} \rangle \) will be built. A hierarchy \( H \) is written \( H = \langle C_0, \ldots, C_{n-1} \rangle \), where each \( C_i \) is a set of dimensions. When \( H \) appears in a set of rules, it means that to evaluate an expression, first 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. Each warehouse \( W \) is a function:

\[
W : \text{Id} \times \text{Ctxt} \to \text{Val}^+ \cup \text{Val}
\]

After adding these entries to warehouse \( W \), the following will hold:

\[
\begin{align*}
W(x, \emptyset) &= C_0 \? \\
W(x, \kappa \mid C_0) &= C_1 \? \\
W(x, \kappa \mid (C_0 \cup C_1)) &= C_2 \? \\
&\quad \vdots \\
W(x, \kappa \mid (C_0 \cup \cdots \cup C_{n-2})) &= C_{n-1} \? \\
W(x, \kappa \mid (C_0 \cup \cdots \cup C_{n-1})) &= c
\end{align*}
\]

where \( C_i \? \) means a demand for the dimensions in \( C_i \).

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.

5.1 Basic operations

Here we define operations on hierarchies. The ‘\(:\)’ is the “cons” operator, and the ‘\(\langle\rangle\)’ is the empty list.

\[
\begin{align*}
\text{restrict}(\langle\rangle, C) &= \langle\rangle \\
\text{restrict}(C_0 : H, C) &= \begin{cases} \\
\text{restrict}(H, C), & C_0 - C = \emptyset \\
(C_0 - C) : \text{restrict}(H, C), & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{merge}(H, \langle\rangle) &= H \\
\text{merge}(\langle\rangle, H') &= H' \\
\text{merge}(C_0 : H, C_0' : H') &= (C_0 \cup C_0') : \text{merge(restrict(H, C_0'), restrict(H', C_0'))}
\end{align*}
\]

\[
\begin{align*}
\text{collapse}(\langle\rangle) &= \emptyset \\
\text{collapse}(C_0 : H) &= C_0 \cup \text{collapse}(H)
\end{align*}
\]

8
\[
\begin{align*}
\text{add}(\mathcal{H}, \mathcal{H}') &= \text{append}\left(\mathcal{H}, \text{restrict}(\mathcal{H}', \text{collapse}(\mathcal{H}))\right) \\
in(\langle \rangle, c) &= \text{false} \\
in(C_0 : \mathcal{H}, c) &= \begin{cases} 
\text{true}, & c \in C_0 \\
in(\mathcal{H}, c), & \text{otherwise}
\end{cases} \\
\text{addone}(\mathcal{H}, c) &= \begin{cases} 
\mathcal{H}, & \text{in}(\mathcal{H}, c) = \text{true} \\
\text{add}\left(\mathcal{H}, \langle \{c\} \rangle\right), & \text{otherwise}
\end{cases}
\end{align*}
\]

5.2 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 \(\varnothing\) 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 \text{mergerhs} and \text{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 \text{mergewarehouses} and \text{addtowarehouse} operators are straightforward.

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

\[
\begin{align*}
E &= \xi_{\text{eqn}}(s) \\
\mathcal{W}, \mathcal{K} \vdash E : c, \mathcal{H}, \mathcal{K}', \mathcal{W}' \\
\mathcal{W}'' &= \text{addtowarehouse}(\mathcal{W}', s, c, \mathcal{H}, \mathcal{K}') \\
\mathcal{W}, \mathcal{K} \vdash \text{id}(s) : c, \mathcal{H}, \mathcal{K}', \mathcal{W}'' \\
c &= \xi_{\text{parse}}(\tau)(s) \\
\mathcal{K'}, \mathcal{W} \vdash \text{const}(\tau, s) : c, \langle \rangle, \mathcal{K}, \mathcal{W} \\
\mathcal{K}, \mathcal{W} \vdash E_i : c_i, \mathcal{H}_i, \mathcal{K}_i, \mathcal{W}_i \\
c &= \xi_{\text{op}}(s)(E_i) \\
\mathcal{H}' &= \text{merge}(\mathcal{H}_i) \\
\mathcal{K}' &= \text{mergerhs}(\mathcal{K}_i) \\
\mathcal{W}' &= \text{mergewarehouses}(\mathcal{W}_i) \\
\mathcal{K}, \mathcal{W} \vdash \text{op}(s)(E_i) : c, \mathcal{H}', \mathcal{K}', \mathcal{W}'
\end{align*}
\]
\[ K, W \vdash E_1 : c_1, H_1, K_1, W_1 \]
\[ c = \xi_{\text{conv}}(\tau)(c_1) \]
\[ K, W \vdash \text{convert}(\tau) E_1 : c, H_1, K_1, W_1 \]
\[ K, W \vdash E_1 : \tau_1(v_1), H_1, K_1, W_1 \]
\[ \tau = \tau_1 \]
\[ K, W \vdash \text{istype}(\tau) E_1 : \text{bool}(\text{true}), H_1, K_1, W_1 \]
\[ K, W \vdash E_1 : \tau_1(v_1), H_1, K_1, W_1 \]
\[ \tau \neq \tau_1 \]
\[ K, W \vdash \text{istype}(\tau) E_1 : \text{bool}(\text{false}), H_1, K_1, W_1 \]
\[ K, W \vdash E_1 : \text{bool}(\text{true}), H_1, K_1, W_1 \]
\[ K_1, W_1 \vdash E_2 : c_2, H_2, K_2, W_2 \]
\[ H' = \text{add}(H_1, H_2) \]
\[ K' = \text{mergerhs}(K_1, K_2) \]
\[ W' = \text{merge warehouses}(W_1, W_2) \]
\[ K, W \vdash \text{if}(E_1, E_2, E_3) : c_2, H', K', W' \]
\[ K, W \vdash E_1 : \text{bool}(\text{false}), H_1, K_1, W_1 \]
\[ K_1, W_1 \vdash E_3 : c_3, H_3, K_3, W_3 \]
\[ H' = \text{add}(H_1, H_3) \]
\[ K' = \text{mergerhs}(K_1, K_3) \]
\[ W' = \text{merge warehouses}(W_1, W_3) \]
\[ W, K \vdash \text{if}(E_1, E_2, E_3) : c_3, H', K', W' \]
\[ K, W \vdash E : c, H, K', W' \]
\[ \kappa = \text{find}(K', c) \]
\[ \kappa' = \text{eval rhs}(\kappa, c) \]
\[ K'' = \text{replace}(K', \kappa, \kappa') \]
\[ K, W \vdash \#E : \kappa'(c), \text{add one}(H, c), K'', W'' \]
\[ K, W \vdash E_1 : \kappa_1, H_1, K_1, W_1 \]
\[ \kappa_1 : K_1, W_1 \vdash E_2 : c_2, H_2, K_2, W_2 \]
\[ W, K \vdash E_2 \oplus E_1 \]
\[ : c, \text{add}(H_1, \text{restrict}(H_2, \text{dom } \kappa_1)), K_2, W_2 \]
\[ K, W \vdash E_{i1} : c_i, H_i, K_i, W_i \]
\[ E'_{i2} = \text{demand}(\xi, K) E_{i2} \]
\[ H' = \text{merge}(H_i) \]
\[ K' = \text{mergerhs}(K_i) \]
\[ W' = \text{merge warehouses}(W_i) \]
\[ K, W \vdash [E_{i1} : E_{i2}] : [c_i : E'_{i2}], H', K', W' \]

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

6 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 [3].

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

References


