Sequential Demand-driven Evaluation of Eager TransLucid*

John Plaice  Blanca Mancilla  Gabriel Ditu  William W. Wadge

Abstract

We present the Eager TransLucid language, an intensional programming language in which the value of a variable is a function mapping multidimensional contexts — the “possible worlds” of intensional logic — to ground values or, equivalently, that variables define multidimensional arrays of arbitrary dimensionality.

The Eager TransLucid language is a natural generalization of Wadge and Ashcroft’s Lucid dataflow language. Given a specific set of equations and a context, the operational semantics determines the value taken by a variable in that context, which may depend both on the values of dimensions within the context and the values of variables in other contexts. The contexts correspond to tags in tagged-token dataflow systems.

The key contribution of the paper is to prove that it is possible to create a warehouse caching the values of already computed (identifier, context) pairs in such a way as to ensure that no reference is made to unnecessary dimensions. The method consists of storing demands for relevant dimensions in the current context as these are needed.

1 Introduction

This paper presents a solution to the problem of demand-driven computation in a multidimensional context, for which the behavior of expressions depends on the values held by some of the dimensions in the current context. When working with these context-dependent expressions, it is common for the set of dimensions in the current context to be far larger than the set of dimensions needed to do a specific computation in that context. The presented solution ensures that only the relevant dimensions, i.e., those explicitly referred to in the computation, will be used. An efficient operational semantics is thus provided for the Eager TransLucid language, defined below.

In many areas of computer science, multidimensional data are common, and play an increasingly important rôle. Typically, there are large numbers of dimensions, but the latter are only used sparsely — with respect to the number of dimensions — in any particular situation. We need computation techniques that do not examine data points unnecessarily or dimensions unnecessarily. Some of this unnecessary computation or examination can be avoided with static analysis, but that is complex and incomplete. Moreover, there are situations whereby examining more than the relevant dimensions will yield nonterminating — or at least overlong — computations.

We are therefore proposing a computation model that at runtime automatically limits itself to only requests of data or dimensions that are necessary for producing the results. In so doing, intermediate values are automatically cached, labeled only with dimensions of relevance.

The origins of this work come from the field of intensional programming, in which a program runs in an implicit multidimensional context and can explicitly manipulate the dimensions, or coordinates, of that context, using a (lazy) demand-driven execution model called education. The term intensional programming was first coined by Antony Faustini and author Wadge [1], in the context of research in the Lucid programming language, and corresponds to the fact that the semantics of intensional programs use the possible-worlds semantics of intensional logic. For a full

discussion of the history and development of Lucid, intensional programming and eduction, we refer the reader to [2].

Developments in intensional programming have always been reflected by continued developments in the Lucid programming language, and the present work is no exception. In this paper, we present Eager TransLucid, which allows any ground value to be used as a dimension, and which clearly separates the definition of identifiers through a system of mutually recursive equations from the demands for computations to be undertaken. Although the basic operational semantics is simple, efficient implementation of the language is non-trivial: traditional eduction techniques, using a static list of dimensions, cannot be applied, since the set of possible dimensions can only be known at run-time.

The solution provided in this paper is purely sequential, in the sense that expressions are evaluated in a left-to-right, depth-first manner. Furthermore, it is assumed that contexts are fully evaluated; it is for this reason that we call the language Eager TransLucid. It is only with a multithreaded interpreter, as described in [3], that contexts can be allowed to be lazy. That interpreter, however, is derived from the one described below.

The way that the sequential interpreter works is that when a identifier-context pair \((x, \kappa)\) is to be evaluated, the warehouse is queried for a cached value for \((x, \emptyset)\), i.e., using the empty context. The warehouse can then respond in one of three ways: 1) there is no information for this pair, in which case the expression defining \(x\) must be evaluated in context \(\kappa\); 2) a value is stored, in which case that value is returned; or 3) a dimensional query, a query for a set of dimensions \(C\) is stored; in this case the warehouse is requiered for a cached value for \((x, \kappa | C)\), and the process begins anew. The dimensional queries are added to the warehouse dynamically, upon encountering dimensional queries while evaluating expressions. It is for this reason that the interpreter described below can guarantee that all dimensions referred to in the warehouse are actually needed for evaluating the expressions.

We present the syntax and definitions (§2) of Eager TransLucid, then a warehouse-based eductive semantics (§3). Future work is discussed in the concluding remarks (§4).

2 Definitions

We define the semantic domains:

- **Typ** \((\ni \tau)\) is the set of types. The types `sp` (special), `bool` and `tuple` must always exist.

- **Str** \((\ni s, x, y)\) is the set of strings. The letters \(x\) and \(y\) are only used when the string is for an identifier.

- **Const** \((\ni c, \subseteq C)\) is for typed constants. A constant \(c\) of value \(v\) and type \(\tau\) is written \(c = \tau(v)\).

- **Ctx**\((\ni \kappa) = \text{Const} \rightarrow \text{Const}\) is for tuples, which are used to encode the current context.
• **Expr** (∃ E) is for expressions. Here is their syntax:

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

• A **system** \( S = (S_{\text{eqn}}, S_{\text{const}}, S_{\text{op}}, S_{\text{conv}}) \) retains these data:

- \( S_{\text{eqn}} : \text{Str} \to \text{Expr} \) is the set of equations.
- \( S_{\text{const}} : \text{Typ} \to \text{Str} \to \text{Ctx} \to \text{Const} \times \text{Const}^+ \) is the set of constants.
- \( S_{\text{op}} : \text{Str} \to \text{Typ}^+ \to \text{Const}^+ \to \text{Ctx} \to \text{Const} \times \text{Const}^+ \) is the set of data operators.
- \( S_{\text{conv}} : \text{Typ} \to \text{Typ} \to \text{Ctx} \to \text{Const} \times \text{Const}^+ \) is the set of type convertors.

• An **external-internal context pair** \( K = (K^{\text{ce}}, K^{\text{ci}}) \) retains these data:
  - \( K^{\text{ce}} \) is the current **external context**, i.e., the context of the initial demand for \( x \).
  - \( K^{\text{ci}} \) is the current **internal context**, i.e., the context built up from context changes since the last demand for a variable.

• A **warehouse** \( W = (W^{\text{se}}, W^{\text{vc}}) \) retains these data:
  - \( W^{\text{vc}} \), the current **value cache**, which encodes the calculated values and demand queries:
    \[
    W^{\text{vc}} : \text{Str} \times \text{Ctx} \to (\text{Const} \cup \text{Const}^+)?
    \]
  - \( W^{\text{se}} \), the current **external subcontext**, i.e., the part of \( K^{\text{ce}} \) that has been examined since the beginning of the evaluation of \( x \).

• An **extended warehouse** \( X = (X^{\text{se}}, X^{\text{vc}}, X^{\text{si}}) \), retains these data:
  - \( X^{\text{vc}} \) is like \( W^{\text{vc}} \) above.
  - \( X^{\text{se}} \) is like \( W^{\text{se}} \) above.
  - \( X^{\text{si}} \) is the current **internal subcontext**, which plays the same rôle as the external sub-context, but which is only used in the warehouse rules.
3 Operational Semantics

3.1 Summary of Rules

The basic evaluation rule is as follows:

\[ x, S, K, W_0 \vdash E : c, W_1 \]

where

- \( x \) is the string for the current identifier whose defining expression is being evaluated.
- \( c \) is the calculated value.
- \( S \) is the current system.
- \( K \) is an external-internal context pair.
- \( W_0 \) and \( W_1 \) correspond to the before and after states of the warehouse.

The expression evaluation rules call the \( \vdash \) rules whenever an identifier is encountered.

\[ S, K, X \vdash x : c, W \]

- \( x \) is the identifier’s string.
- \( c \) is the calculated value.
- \( S \) is the system.
- \( K \) is an external-internal context pair.
- \( X \) is an extended warehouse.
- \( W \) is a warehouse.

3.2 Identifiers

3.2.1 Interaction with Warehouse

There are four rules for accessing the warehouse. The first rule is the simplest, and corresponds to a situation in which there is already a value in the warehouse. In this case, the value is simply returned, and the visited subcontext and the warehouse do not change.

\[ X \vdash (x, X \searrow \uparrow X \swarrow) = c \]

\[ W = (X \searrow, X \swarrow) \]

\[ S, K, X \vdash x : c, W \]

In the next two rules, the warehouse informs the evaluator that the values corresponding to a certain set of dimensions need to be known for evaluation to continue. This set of dimensions is split into two, since some of these dimensions form part of the internal context, which does not contribute to the warehouse, while the remainder form part of the external context, and must be returned.

\[ X \vdash (x, X \searrow \uparrow X \swarrow) = C? \]

\[ C \not\subseteq \text{dom} \ K^{ne} \cup \text{dom} \ K^{ci} \]

\[ W = (X \searrow, X \swarrow) \]

\[ S, K, X \vdash x : \text{sp}(\text{undefdim}), W \]
\(X_1 = \left( X_0^{\text{vc}}, X_0^{\text{se}} \upharpoonright X_0^{\text{si}} \right) = C?\)

\[ X_1 = \left( X_0^{\text{vc}}, X_0^{\text{se}} \upharpoonright \left( K^{\text{ce}} \mid (C - \text{dom} K^{\text{ci}}) \right), X_0^{\text{si}} \upharpoonright \left( K^{\text{ci}} \mid C \right) \right) \]

\[ S, K, X_0 \vdash x : c, W \]

\[ S, K, X_1 \vdash x : c, W \]

The last rule corresponds to the situation where there is no information in the warehouse. It is here that the expression evaluator is initiated. It is in this rule that the identifier in the left-hand side of the expression evaluator is set.

\[ W_0 = \left( X_0^{\text{vc}}, X_0^{\text{se}} \upharpoonright X_0^{\text{si}} \right) \]

\[ K_1 = \left( K_0^{\text{ce}} \upharpoonright K_0^{\text{ci}}, \emptyset \right) \]

\[ E = S_{\text{eqn}}(x) \]

\[ x, S, K_1, W_0 \vdash E : c, W_1 \]

\[ W_2 = \left( W_1^{\text{vc}} \upharpoonright [(x, W_0^{\text{se}}) \mapsto c], X_0^{\text{se}} \upharpoonright (W_0^{\text{se}} - K_0^{\text{ci}}) \right) \]

3.2.2 Computing an Identifier in an Expression

There are two rules for identifiers appearing within expressions. The warehouse is called with no information whatsoever about the current subcontext. It then returns with a needed subcontext. Should this subcontext be included in the original subcontext, then nothing needs to be done.

\[ X = \left( W_0^{\text{vc}}, \emptyset, \emptyset \right) \]

\[ S, K, X \vdash y : c, W_1 \]

\[ W_1^{\text{se}} \subseteq W_0^{\text{se}} \]

\[ W_2 = \left( W_1^{\text{vc}} \upharpoonright [(x, W_0^{\text{se}}) \mapsto c], W_1^{\text{se}} \upharpoonright W_0^{\text{se}} \right) \]

Otherwise a demand needs to be added to the warehouse.

\[ X = \left( W_0^{\text{vc}}, \emptyset, \emptyset \right) \]

\[ S, K, X \vdash y : c, W_1 \]

\[ C = \text{dom} W_1^{\text{se}} - \text{dom} W_0^{\text{se}} \left( \neq \emptyset \right) \]

\[ W_2 = \left( W_1^{\text{vc}} \upharpoonright [(x, W_0^{\text{se}}) \mapsto C?], W_0^{\text{se}} \upharpoonright W_1^{\text{se}} \right) \]

3.3 Constants

Contexts in TransLucid are context-dependent. The string \( s \) is interpreted in the current context to produce a constant value of type \( \tau \). If this cannot be done, the result is a type error.

\[ \tau \notin \text{dom}(S_{\text{const}}) \lor s \notin \text{dom}(S_{\text{const}}(\tau)) \]

\[ x, S, K, W_0 \vdash \text{const}(\tau, s) : \text{sp(type)}, W_0 \]

The constant interpreter returns a set \( C \) of dimensions needed to interpret \( s \), and these must be added to the set of dimensions seen.

\[ (c, C) = S_{\text{const}}(\tau)(s)(K^{\text{ce}} \upharpoonright K^{\text{ci}}) \]

\[ W_1 = \left( W_0^{\text{vc}}, W_0^{\text{se}} \upharpoonright (K^{\text{ce}} \mid C) \right) \]

\[ x, S, K, W_0 \vdash \text{const}(\tau, s) : c, W_1 \]
3.4 Data Operators

The operands of a data operator are evaluated eagerly. Should any errors arise, the worst error is propagated.

\[ x, S, K, W_{i-1} \vdash E_i : \tau_i(v_i), W_i, \quad i = 1..n \]

\[ v = \min\{v_i : \tau_i = sp\} \]

\[ x, S, K, W_0 \vdash \text{op}(s)(E_1, \ldots, E_n) : \text{sp}(v), W_n \]

If no operator of the right type can be found, a type error is returned.

\[ x, S, K, W_{i-1} \vdash E_i : \tau_i(v_i), W_i, \quad i = 1..n \]

\[ s \notin \text{dom}(S_{\text{op}}) \lor (\tau_1, \ldots, \tau_n) \notin \text{dom}(S_{\text{op}}(s)) \]

\[ x, S, K, W_0 \vdash \text{op}(s)(E_1, \ldots, E_n) : \text{sp}(\text{type}), W_n \]

The operator interpreter returns a set \( C \) of dimensions needed to interpret \( s \), and these must be added to the set of dimensions seen.

\[ x, S, K, W_{n+1} = (W_n^c, W_n^s \uparrow (K^c \mid C)) \]

\[ x, S, K, W_0 \vdash \text{op}(s)(E_1, \ldots, E_n) : c, W_{n+1} \]

3.5 Type Convertors

The operand of a type convertor is evaluated eagerly. Should an error arise, it is propagated.

\[ x, S, K, W_0 \vdash E : \text{sp}(v), W_1 \]

\[ x, S, K, W_0 \vdash \text{convert}(\tau_o) E : \text{sp}(v), W_1 \]

If no convertor of the right type can be found, a type error is returned.

\[ x, S, K, W_0 \vdash E : \tau(v), W_1 \]

\[ \tau \notin \text{dom}(S_{\text{conv}}) \lor \tau_o \notin \text{dom}(S_{\text{conv}}(\tau)) \]

\[ x, S, K, W_0 \vdash \text{convert}(\tau_o) E : \text{sp}(\text{type}), W_1 \]

The operator interpreter returns a set \( C \) of dimensions needed to interpret \( s \), and these must be added to the set of dimensions seen.

\[ x, S, K, W_2 = (W_1^c, W_1^s \uparrow (K^c \mid C)) \]

\[ x, S, K, W_0 \vdash \text{convert}(\tau_o) E : c, W_2 \]

3.6 Tests for Types

The istype command is not strict with respect to special values. It always returns a Boolean value.

\[ x, S, K, W_0 \vdash E : \tau(v), W_1 \]

\[ x, S, K, W_0 \vdash \text{istype}(\tau) E : \text{bool}(\text{true}), W_1 \]

\[ x, S, K, W_0 \vdash E : c, W_1 \]

\[ x, S, K, W_0 \vdash \text{istype}(\tau) E : \text{bool}(\text{false}), W_1 \]
3.7 Tests for Special Values

The `isspecial` command is not strict with respect to special values. It always returns a Boolean value.

\[
\begin{align*}
x, S, K, W_0 & \vdash E : \text{sp}(v), W_1 \\
x, S, K, W_0 & \vdash \text{isspecial}(v) E : \text{bool}(true), W_1 \\
x, S, K, W_0 & \vdash E : c, W_1 \\
x, S, K, W_0 & \vdash \text{isspecial}(v) E : \text{bool}(false), W_1
\end{align*}
\]

3.8 Absolute Tuple Creation

The operands of a tuple are evaluated eagerly. Should any errors arise, the worst error is propagated.

\[
\begin{align*}
x, S, K, W_{i-1} & \vdash E_i : \tau_i(v_i), W_i, \ i = 1..2n \\
v & = \min \{ v_i \mid \tau_i = \text{sp} \} \\
x, S, K, W_0 & \vdash [E_1 \gets E_2, \ldots, E_{2n-1} \gets E_{2n}] : \text{sp}(v), W_{2n}
\end{align*}
\]

The tuple is created directly.

\[
\begin{align*}
x, S, K, W_{i-1} & \vdash E_i : c_i, W_i, \ i = 1..2n \\
{c} & = \text{tuple}(c_1 \mapsto c_2, \ldots, c_{2n-1} \mapsto c_{2n}) \\
x, S, K, W_0 & \vdash [E_1 \gets E_2, \ldots, E_{2n-1} \gets E_{2n}] : c, W_{2n}
\end{align*}
\]

3.9 Relative Tuple Creation

The operands of a tuple are evaluated eagerly. Should any errors arise, the worst error is propagated.

\[
\begin{align*}
x, S, K, W_{i-1} & \vdash E_i : \tau_i(v_i), W_i, \ i = 1..2n \\
v & = \min \{ v_i \mid \tau_i = \text{sp} \} \\
x, S, K, W_0 & \vdash [E_1 \gets E_2, \ldots, E_{2n-1} \gets E_{2n}] : \text{sp}(v), W_{2n}
\end{align*}
\]

The computed tuple perturbs the current context. The entire external context is now considered to be seen.

\[
\begin{align*}
x, S, K, W_{2n+1} & = (W_{2n}^{\text{se}} \upharpoonright K^{\text{ce}} : W_{2n}^{\text{yc}}) \\
c & = K^{\text{ce}} \upharpoonright K^{\text{ci}} \upharpoonright \text{tuple}(c_1 \mapsto c_2, \ldots, c_{2n-1} \mapsto c_{2n}) \\
x, S, K, W_0 & \vdash [E_1 \gets E_2, \ldots, E_{2n-1} \gets E_{2n}] : c, W_{2n+1}
\end{align*}
\]

3.10 Tuple Perturbation

Both operands are evaluated eagerly. Should any errors arise, the worst error is propagated.

\[
\begin{align*}
x, S, K, W_{i-1} & \vdash E_i : \tau_i(v_i), W_i, \ i = 1..2n \\
v & = \min \{ v_i \mid \tau_i = \text{sp} \} \\
x, S, K, W_0 & \vdash E_2 \cdot E_1 : \text{sp}(v), W_2
\end{align*}
\]
Both operands must produce tuples.

\[ x, S, K, W_{i-1} \vdash E_i : \tau_i(v_i), W_i, \quad i = 1, 2 \]

\[ \tau_1 \neq \text{tuple} \lor \tau_2 \neq \text{tuple} \]

\[ x, S, K, W_0 \vdash E_2 \cdot E_1 : \text{sp(type)}, W_2 \]

The result is the perturbation of the second by the first.

\[ x, S, K, W_{i-1} \vdash E_i : \kappa_i, W_i, \quad i = 1, 2 \]

\[ x, S, K, W_0 \vdash E_2 \cdot E_1 : \kappa_2 \uparrow \kappa_1, W_2 \]

### 3.11 Tuple Restriction

There are two restriction operators, \( \setminus^+ \) and \( \setminus^- \). We combine their descriptions into \( \setminus^\pm \).

Both operands are evaluated eagerly. Should any errors arise, the worst error is propagated.

\[ x, S, K, W_{i-1} \vdash E_i : \tau_i(v_i), W_i \]

\[ v = \min\{v_i \mid \tau_i = \text{sp}\} \]

\[ x, S, K, W_0 \vdash E \setminus^\pm \{E_1, \ldots, E_n\} : \text{sp(v)}, W_1 \]

The main argument must be a tuple.

\[ x, S, K, W_0 \vdash E : \tau(v), W_1 \]

\[ \tau \neq \text{tuple} \]

\[ x, S, K, W_0 \vdash E \setminus^\pm \{E_1, \ldots, E_n\} : \text{sp(type)}, W_1 \]

The result is to limit the domain of the tuple. For \( \setminus^+ \), only the named dimensions are kept. For \( \setminus^- \), all of the named dimensions are thrown out.

\[ x, S, K, W_{n-1} \vdash E_i : c_i, W_i \]

\[ x, S, K, W_n \vdash E : \kappa, W_{n+1} \]

\[ \kappa' = \kappa \setminus^\pm \{c_1, \ldots, c_n\} \]

\[ x, S, W, K_0 \vdash E \setminus^\pm \{E_1, \ldots, E_n\} : \kappa', W_{n+1} \]

### 3.12 Conditional Expressions

The conditional expression begins by evaluating the condition. If there is an error, it is propagated.

\[ x, S, K, W_0 \vdash E_1 : \text{sp(v)}_1, W_1 \]

\[ x, S, K, W_0 \vdash \text{if} (E_1, E_2, E_3) : \text{sp(v)}_1, W_1 \]

If the condition is not Boolean, there is a type error.

\[ x, S, K, W_0 \vdash E_1 : \tau_1(v_1), W_1 \]

\[ \tau_1 \neq \text{bool} \]

\[ x, S, K, W_0 \vdash \text{if} (E_1, E_2, E_3) : \text{sp(type)}, W_1 \]

Expression \( E_2 \) is evaluated if the condition is true.

\[ x, S, K, W_0 \vdash E_1 : \text{bool(true)}, W_1 \]

\[ x, S, K, W_1 \vdash E_2 : c, W_2 \]

\[ x, S, K, W_0 \vdash \text{if} (E_1, E_2, E_3) : c, W_2 \]

Otherwise, expression \( E_3 \) is evaluated.

\[ x, S, K, W_0 \vdash E_1 : \text{bool(false)}, W_1 \]

\[ x, S, K, W_1 \vdash E_3 : c, W_2 \]

\[ x, S, K, W_0 \vdash \text{if} (E_1, E_2, E_3) : c, W_2 \]
3.13 Context Query

The context query begins by computing the dimension. If there is an error, it is propagated.

\[ x, S, K, W_0 \vdash E : \text{sp}(v), W_1 \]
\[ x, S, K, W_0 \vdash \# E : \text{sp}(v), W_1 \]

If the dimension is in \( K^c_1 \), its value is looked up there.

\[ x, S, K, W_0 \vdash E : c, W_1 \]
\[ c \in \text{dom}(K^c_1) \]
\[ x, S, K, W_0 \vdash \# E : K^c_1(c), W_1 \]

If the dimension is in \( W^se \), its value is looked up there.

\[ x, S, K, W_0 \vdash E : c, W_1 \]
\[ c \in \text{dom}(W^se) \]
\[ x, S, K, W_0 \vdash \# E : W^se_1(c), W_1 \]

If the dimension is in \( X^ce \), its value is looked up there, and a demand query for that dimension is added to the warehouse.

\[ x, S, K, W_0 \vdash E : c, W_1 \]
\[ c \in \text{dom}(X^ce) \]
\[ W_2 = (W^c_1 \uplus [(x, W^se_1) \mapsto \{c\}], W^se_1 \uplus [c \mapsto X^ce(c)]) \]
\[ x, S, K, W_0 \vdash \# E : K^ce(c), W_2 \]

Otherwise no-one knows about this dimension.

\[ x, S, K, W_0 \vdash E : c, W_1 \]
\[ x, S, K, W_0 \vdash \# E : \text{sp}(\text{undefdim}), W_1 \]

3.14 Context Change

The context change begins by computing the new context. If there is an error, it is propagated.

\[ x, S, K, W_0 \vdash E_1 : \text{sp}(v_1), W_1 \]
\[ x, S, K, W_0 \vdash E_2 \circ E_1 : \text{sp}(v_1), W_1 \]

The new context must be a tuple.

\[ x, S, K, W_0 \vdash E_1 : \tau_1(v_1), W_1 \]
\[ \tau_1 \neq \text{tuple} \]
\[ x, S, K, W_0 \vdash E_2 \circ E_1 : \text{sp}(\text{type}), W_1 \]

When there is a context change, we must compute a new external-internal context pair \( K_1 \), splitting the new context into parts coming from the external context and the rest. The expression \( E_2 \) is then computed, and the external subcontexts computed must be merged.

\[ x, S, W_0, K_0 \vdash E_1 : \kappa_1, W_1 \]
\[ W_2 = (W^c_1 \cap W^se_1, \kappa_1 \cap W^se_1) \]
\[ K_1 = (\kappa_1 \cap W^se_1, \kappa_1 - W^se_1) \]
\[ x, S, W_2, K_1 \vdash E_2 : c, W_3 \]
\[ W_4 = (W^c_3, W^se_1 \uplus W^se_3) \]
\[ x, S, W_0, K_0 \vdash E_2 \circ E_1 : c, W_4 \]
3.15 Sequential Evaluator

The sequential evaluator simply evaluates the second operand after the first, passing on the result of the second.

\[
\begin{align*}
  x, S, K, W_{i-1} & \vdash E_i : c_i, W_i, \quad i = 1..2 \\
  x, S, K, W_0 & \vdash E_1 ; E_2 : c_2, W_2
\end{align*}
\]

4 Conclusions

The eductive algorithm presented in §3 has made it possible to manipulate data and programs of high dimensionality in a demand-driven manner in such a way as to always manipulate the minimal set of apparently relevant dimensions.

The eductive algorithm gives a general demand-driven solution, which yields quite efficient solutions whenever the control flow is difficult to predict. For regular programs in which high performance is necessary, it is quite possible to modify the algorithm so that in specific situations, a single demand can provoke a highly efficient data-driven implementation for an (identifier, region) pair, where a region is a set of contexts.

The eductive algorithm also assumes that the warehouse is always added to. It is possible to add various forms of retirement schemes to an eductive interpreter. The original Lucid interpreters had such schemes, which were quite effective, and the same techniques may be applied here.

References

