Eduction: A general model for computing

John Plaice    Slim Ben Lamine
Département d’informatique
Université Laval
Ste-Foy (Québec), Canada G1K 7P4
E-mail: {John.Plaice,benlamin}@ift.ulaval.ca

1996

Abstract

Eduction is shown to be a very general model of computation. A general form of warehouse is defined, using the notion of \( n \)-dimensional intervals. With such warehouses, it is possible to formally define the educative processes currently being used in Lucid and Lemur, as well as to point to interesting solutions for fault-tolerant and heterogeneous computing.

1 Introduction

Eduction is a lazy demand-driven technique first used for implementing the Lucid language. Under this model, when a particular value in a program is required, a demand is made to the program for that value. If this value is available immediately, it is returned; otherwise the value is computed from the appropriate definition, possibly provoking new demands. Unless an exceptional situation occurs, there will always be a reply to the original demand.

Key to this implementation technique is the warehouse, which allows one to cache previously computed values, ensuring that if recomputations of intermediate values are required in the future, then the values need not be recomputed.

Eduction is a perfectly natural way of computing. A user makes a query for some information, and this query can itself provoke new demands, possibly requiring computation along the way. As the process takes place, some of the newly computed or acquired information can be stored in a cache. This is, for example, the process that takes place on the World Wide Web.

More generally, the user of a computer system spends his or her time requesting information—using input devices such as keyboard, mouse, joystick, etc.—to be placed on some output device (screen, printer, speakers, etc.), or depositing information that can later be used when some other user requests information. It is precisely this notion of demand that provokes other demands that is key to eduction.

Using eduction does not just offer a means for computation. It also has the advantage of allowing computations to be effected in a fault-tolerant, parallel manner, with a variety of physical implementations. A single demand over a large domain can actually provoke many smaller demands in parallel.
The purpose of this paper is to analyze in detail the process of eduction, bringing out some of its hitherto hidden aspects. We do this by examining four kinds of computing: dataflow programming with Lucid, software configuration with Lemur, heterogeneous computing, and $N$-version programming. The heart of the paper is a formal presentation of warehouses and queries to warehouses. We then show how these warehouses can be used for the different problems.

2 Requirements

2.1 Lucid

Our first view of eduction is through the multidimensional programming language Lucid. Suppose that there is a warehouse $W$ associating values $v$ with $(id, T)$ pairs, where $T$ is some kind of tag. The intuition is that the variable $id$ has value $v$ in context $T$, the latter being defined by the values of its components.

For example, in the program

```plaintext
X @.d 3
where
dimension d;
X = 42 fby.d X+1;
end;
```

the computing process is as follows:

\[ \begin{align*}
X_{[d=3]} &\rightarrow X_{[d=2]} + 1 \\
&\rightarrow (X_{[d=1]} + 1) + 1 \\
&\rightarrow ((X_{[d=0]} + 1) + 1) + 1 \\
&\rightarrow ((42 + 1) + 1) + 1 \\
&\rightarrow 45.
\]

A demand is made to a warehouse $W$ for a variable $id$ along with a tag $T = \{D_j : d_j\}$, where all the $d_j$ are integers. If the required value $v$ is stored in the warehouse, then $v$ is returned. Otherwise, a special computing value is placed in the warehouse for that $(id, T)$ combination, the value $v$ is computed from the variable’s definition — possibly provoking more demands for values from the warehouse — and then the computing value is replaced in the warehouse by $v$. The value $v$ is then returned. Should a new demand find the value computing, then this means that there is a dependency cycle in the definition of $id$.

For this implementation to work, a Lucid warehouse must be able to store points defined over the space of dimensions

\[ \{\text{Id}, \text{Val}\} \cup \{D_i\}_{i \in \omega}, \]

where \text{Id} is a dimension for identifiers, \text{Val} is a dimension for values and each $D_i$ is an integer dimension. Since Lucid has no default reasoning mechanism, the orders for all of the dimensions are flat.

For the above program, the warehouse will be initially empty. The first, unsuccessful demand will be for $(\text{Id} : X, \text{z} : 3)$. So $(\text{Id} : X, \text{Val} : \text{computing}, \text{z} : 3)$
will be added to the warehouse. Successive demands will be made, ultimately creating the warehouse

\[
\begin{align*}
\{ & \text{Id} : X, \text{Val} : \text{computing}, z : 3 \}, \\
\{ & \text{Id} : X, \text{Val} : \text{computing}, z : 2 \}, \\
\{ & \text{Id} : X, \text{Val} : \text{computing}, z : 1 \}, \\
\{ & \text{Id} : X, \text{Val} : \text{computing}, z : 0 \}
\end{align*}
\]

The computations may now begin, and the resulting warehouse, at the end, will be

\[
\begin{align*}
\{ & \text{Id} : X, \text{Val} : 45, z : 3 \}, \\
\{ & \text{Id} : X, \text{Val} : 44, z : 2 \}, \\
\{ & \text{Id} : X, \text{Val} : 43, z : 1 \}, \\
\{ & \text{Id} : X, \text{Val} : 42, z : 0 \}
\end{align*}
\]

Now, in Lucid, it is possible to add extra dimensions. For example, in the program below, \(w\) is an unused dimension: one would not want a query using \(w\) to affect the behavior of the function.

\[
X \oplus . z \ 3
\]

where

\[
\text{dimension } w, z;
\]

\[
X = 42 \ fby. z \ X+1;
\]

end;

The initial demand will be \((\text{Id} : X, w : 0, z : 3)\). However, \(w\) never plays a role in the definition of \(X\). So, the warehouse at the end of computation should still only be:

\[
\begin{align*}
\{ & \text{Id} : X, \text{Val} : 45, z : 3 \}, \\
\{ & \text{Id} : X, \text{Val} : 44, z : 2 \}, \\
\{ & \text{Id} : X, \text{Val} : 43, z : 1 \}, \\
\{ & \text{Id} : X, \text{Val} : 42, z : 0 \}
\end{align*}
\]

### 2.2 Lemur

Lemur is an experimental C programming environment developed by Plaice and Wadge, in which versioning of components and complete systems is seamlessly integrated. There is an order between versions, and when a demand for a component is made, the most relevant version in the warehouse, as defined by the order, is returned. We discuss here the natural extension of Lemur to multidimensional version spaces.

Given the model of a warehouse as presented in the previous section, the universe \(U\) of dimensions is

\[
\{ \text{Id} , \text{Val} \} \cup \{ D_i \}_{i \in \omega},
\]

where \(\text{Id}\) is a dimension for identifiers, \(\text{Val}\) is a dimension for values and each \(D_i\) is an attribute dimension. However, what is new from the Lucid warehouse is that for each attribute dimension \(D_i\), we will need a partial order \(\sqsubseteq_i\), with minimal element \(\bot_i\).

A general partial order can be defined from all the partial orders on individual dimensions. Given two points \(p = \{ D_j : d_j \}_{j} \) and \(p' = \{ D_j : d'_j \}_{j} \), we write \(p \sqsubseteq p'\) if and only if for all \(i = 1..n\), \(d_i \sqsubseteq d'_i\).
Now, we are no longer interested in querying for individual points in the version space. Rather, demands to the warehouse must request all points underneath a given point:

\[
\{ \text{Id} : id, \; D_1 : \leq d_1, \ldots, D_n : \leq d_n \}.
\]

The query’s result should return a set of points, and the variant substructure principle should choose the most relevant of these points. Quite simply, the generalized order is used to select the maximal element, if it exists.

It is not clear what should be done in situations where \((D_1 : d_1)\) is requested and something of the form \(\{D_1 : d_1', D_2 : d_2'\}\) is returned. It is not of much interest in Lemur the way Lemur stands. However, it is certainly of use in heterogeneous systems.

### 2.3 Heterogeneous computing

The typical computing environment is heterogeneous, in the sense that not all computers offer exactly the same environment, be it hardware or software. Too often, users must spend time trying to figure out what machine a program should run on, as it may not give correct—if at all—results on all machines. Hardware may be different, operating systems may be different, upgrades may be effected inconsistently, a machine might be overloaded, etc.

Ideally, one would just request that a task be effected, without worrying about the actual computing resources that are required. This approach can in fact be implemented using the eductive approach: two warehouses can be assumed, one which is similar to that used for Lemur, the other defining the available computer resources. Requests for a particular version of software are made to the first warehouse. Unlike in Lemur, it will be normal for there to be extra dimensions in the returned values. These extra dimensions will define the characteristics of the required computing environment for the versions of software that are returned after the first query. This information can then be used to create queries to the computing resources warehouse, to see if any of the versions of software that are currently available are actually compatible with the available computing resources.

### 2.4 Fault-tolerant computing

One of the ways to increase fault-tolerance is to use redundancy. A standard method for doing this is called \(N\)-version programming, where \(N\) different copies of a piece of software all run the same thing, and in which a voting system is used to increase the likelihood of correct behavior despite the potential failure of a single component.

Supposing that we have a computation method based on eduction, it can naturally be generalized to a fault-tolerant computation method, simply by adding a Machine dimension. Simultaneous demands are made to \(n\) machines, and then a request for a value will return a number of values (perhaps less than \(n\) if one of the machines is not properly functioning). A voting algorithm can then be effected on the set of returned values to determine the correct one.
3 Dimensions, intervals and warehouses

We are now ready to formalize the interactions with a warehouse. We begin by defining the notions of dimension, point and interval.

Definition 1. A dimension $\mathcal{D}$ is a triple $\langle D_\mathcal{D}, D_\mathcal{D}, \sqsubseteq_\mathcal{D} \rangle$, where $D_\mathcal{D}$ is the name of the dimension, $D_\mathcal{D}$ is the domain of values associated with the dimension and $\sqsubseteq_\mathcal{D}$ is a partial order over $D_\mathcal{D}$.

Definition 2. Let $\mathcal{D} = \langle D_\mathcal{D}, D_\mathcal{D}, \sqsubseteq_\mathcal{D} \rangle$ be a dimension. A point $p$ in $\mathcal{D}$ is simply $(D_\mathcal{D} : d)$, where $d \in D_\mathcal{D}$.

Definition 3. Let $\mathcal{D} = \langle D_\mathcal{D}, D_\mathcal{D}, \sqsubseteq_\mathcal{D} \rangle$ be a dimension. An interval $I$ in $\mathcal{D}$ is one of the following possibilities:

- $(D_\mathcal{D} : D_\mathcal{D}) = \{(D_\mathcal{D} : d)\}$;
- $(D_\mathcal{D} : d \geq d^{\min}) = \{(D_\mathcal{D} : d) \mid d \geq d^{\min}\}$;
- $(D_\mathcal{D} : d \leq d^{\max}) = \{(D_\mathcal{D} : d) \mid d \leq d^{\max}\}$;
- $(D_\mathcal{D} : [d^{\min}, d^{\max}]) = \{(D_\mathcal{D} : d) \mid d \geq d^{\min} \land d \leq d^{\max}\}$.

An interval $I$ will often be written $(D_\mathcal{D} : I)$.

We will assume that each dimension has a unique name, so below, we can use $D$ to uniquely designate a dimension. The meanings of $D_\mathcal{D}$ and $\sqsubseteq_\mathcal{D}$ should be clear.

Now that we have defined intervals, we can define their intersection, which should also be an interval.

Definition 4. Let $I = (D : I)$ and $I' = (D : I')$ be two intervals in $D$. The intersection of $I$ and $I'$ is $I \cap I' = (D : I \cap I')$.

Proposition 1. Let $I$ and $I'$ be two intervals in $D$. The intersection $I \cap I'$ is also an interval in $D$.

Proof Trivial.

Now the definitions can be extended to an arbitrary number of dimensions.

Definition 5. Let $\{D_j\}_{j \in J}$ be a finite set of dimensions. A generalized point in $\{D_j\}_{j \in J}$ is simply $\{(D_j : d_j)\}_{j \in J}$, where each $(D_j : d_j)$ is a point in $D_j$.

To simplify the presentation below, we will write $\{D_j\}_j$ for $\{D_j\}_{j \in J}$ and $\{D_j : d_j\}_j$ for $\{(D_j : d_j)\}_{j \in J}$.

Definition 6. Let $\{D_j\}_j$ be a finite set of dimensions. A generalized interval (parallellepiped) $I$ in $\{D_j\}_j$ is $\{D_j : I_j\}_j$, where each $(D_j : I_j)$ is an interval in $D_j$. The interval $I$ contains all the generalized points of the form $\{D_j : d_j\}_j$, where the $d_j \in I_j$.

Definition 7. Let $I = \{D_j : I_j\}_j$ be an interval in $\{D_j\}_j$. The range of $I$ is $R(I) = \{D_j\}_j$. 

5
We can always introduce new dimensions in an expression, it should make no difference to the meaning. However, it will be necessary when defining the intersection of two regions defined over arbitrary dimensions.

**Definition 8.** Let \( \mathcal{I} = \{ D_j : I_j \} \) be an interval in \( \{ D_j \} \) and let \( \{ D'_k \} \) be another finite set of dimensions, where the \( D_j \) and \( D'_k \) are distinct. The natural extension of \( \mathcal{I} \) to \( \{ D_j \} \cup \{ D'_k \} \) is an interval in \( \{ D_j \} \cup \{ D'_k \} \) defined by: \( \{ D_j : I_j \} \cup \{ D'_k : D'_k \} \).

**Definition 9.** Let \( \mathcal{I} = \{ D_j : I_j \} \) and \( \mathcal{I}' = \{ D'_j : I'_j \} \) be two intervals in \( \{ D_j \} \). The intersection of \( \mathcal{I} \) and \( \mathcal{I}' \) is \( \mathcal{I} \cap \mathcal{I}' = \{ D_j : I_j \cap I'_j \} \).

**Proposition 2.** Let \( \mathcal{I} \) and \( \mathcal{I}' \) be two intervals in \( \{ D_j \} \). The intersection \( \mathcal{I} \cap \mathcal{I}' \) is also an interval in \( \{ D_j \} \).

**Proof** Trivial.

**Definition 10.** Let \( \mathcal{I} \) and \( \mathcal{I}' \) be two arbitrary intervals. Then the intersection of \( \mathcal{I} \) and \( \mathcal{I}' \) is given by

\[
\mathcal{I} \cap \mathcal{I}' = \mathcal{I} \cap R(\mathcal{I}) \cap \mathcal{I}' \cap R(\mathcal{I}').
\]

**Definition 11.** Let \( \mathcal{I} = \{ D_j : I_j \} \) be an interval in \( \{ D_j \} \). The canonical interval for \( \mathcal{I} \), written \( C(\mathcal{I}) \), consists of those entries of \( \mathcal{I} \) for which \( I_i \neq D_i \).

Now we can define a warehouse and a query.

**Definition 12.** Let \( U \) be a set of dimensions. A warehouse \( W \) over \( U \) is a set of canonical intervals such that the range of each interval is included in \( U \).

**Definition 13.** Let \( W \) be a warehouse over \( U \) and let \( I_0 \) be an interval such that \( R(I) \subseteq U \). The query \( W?I \) is the subset \( \{ I \in W \mid I_0 \cap I \neq \emptyset \} \).

In the next two sections, we will show how the warehouse is used to model Lucid computations and Lemur version control.

### 4 Lucid computation

Rather than to simply explain how computation takes place in Lucid, we will extend this process so that the definitions themselves are found inside the warehouse. This is the process that is used for spreadsheets, as implemented by Du and Wadge. We will suppose that the definition of a variable \( X \) will be written \( \lceil X \rceil \).

We will also suppose, for simplicity, that the syntax of a Lucid expression is as follows:

\[
E ::= k \\
| X \\
| \#d \\
| E_0 @d E_1 \\
| E_1 E_0
\]

To evaluate an expression \( E \), we need to evaluate it for a given tag \( T \) using a given warehouse \( W \). Here is the definition of \( \text{eval}(E, T, W) \):
eval(E, T, W) =
case E of
  \_k, empty, W) -> lookup(X, T, W)
  \_d T, d: d(T), W)
E_0 \oplus d E_1 let (v_0, T_0, W_0) = eval(E_0, T, W)
  in let (v_1, T_1, W_1) = eval(E_1, T[d \leftarrow v_0], W_0)
  in (v_1, T_0 \cup T_1, W_1)
end end
E_1 E_0 let (v_0, T_0, W_0) = eval(E_0, T, W)
  in let (v_1, T_1, W_1) = eval(E_1, T, W_0),
  in ((v_1, v_0), T_0 \cup T_1, W_1)
end end

The evaluation function uses another function, called lookup, which interacts
with the warehouse. Note that interactions with the warehouse can themselves
change the warehouse. If a value for a variable is not available, then the defini-
tion for that variable is sought. Once the definition is found, the comp value is
placed in the warehouse to state that the value is being computed, the expres-
sion is evaluated and the resulting value is stored in the warehouse. Should the
warehouse be queried for the value while computing, then the result is an error.

lookup(X, T, W) =
  let entry = W ? (Id::X, T)
  in if empty(entry)
    then let (defn, T', W') = lookup([X], T, W)
        in let (v, T'', W'') = eval(defn, T, W' + (Id::X, T, val: comp))
            in (v, T'', W'' - (Id::X, T, val: comp)
                + (Id::X, T'', val : v))
    end
  else let v = val(entry)
        T' = tag(entry)
        in if v = comp
            then raise error
            else (v, T', W)
        end
  end
5 Lemur version control

We suppose here for simplicity that we are versioning expressions of the form:

\[ E ::= X \]
\[ | E_1 E_0 \]

Once again for simplicity’s sake, we will suppose that all queries will use only one dimension \( d \). We compute the least tag for an expression, based on the tags of the selected components.

\[
eval(E, T, W) = 
\begin{cases} 
\text{lookup}(X, T, W) & \text{if } E = X \\
\text{let } (v_0, T_0) = \eval(E_0, T, W) \\
(v_1, T_1) = \eval(E_1, T, W) \\
in ((v_1, v_0), \max(T_0, T_1)) 
\end{cases}
\]

The \( \text{lookup} \) function collects all of the versions that are less than the requested tag, then takes the version corresponding to the maximum tag, returning that tag along with the version.

\[
\text{lookup}(X, T, W) = 
\begin{cases} 
\text{let } entry = W ? (id: X, d \leftarrow T) \\
in \text{let } mtag = \max(\text{get tags}(entry)) \\
in \text{if } mtag = \top \\
then \text{raise error} \\
else \text{let } E = \text{select}(entry, mtag) \\
in (\text{val } E, mtag) 
\end{cases}
\end{cases}
\]

6 Discussion

The general model of warehouse that we have presented shows that many different aspects of software engineering and of computation can be understood using a single framework, namely eduction. However, the implied generality leaves us with some unanswered questions. To what extent can a programming language manipulating such warehouses be used for general-purpose computation, and how could it be integrated into network-based software, in which, for example, warehouses might correspond to centralized databases? Clearly, much work remains to be done in this area.