Eduction: A general model for computing

John Plaice        Slim Ben Lamine†

May 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 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.

It turns out that 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 some sort of cache. This is, for example, the process that takes place on the World Wide Web.

Eduction is not just a natural form of computation. It also has the advantage of allowing computations to be effected in a fault-tolerant, parallel manner, with a variety of physical implementations.

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

*Presented to ISLIP’96 (International Symposium on Languages for Intensional Programming), held in Tempe, Arizona, USA, on 13–15 May 1996.
†Département d’informatique, Université Laval, Ste-Foy (Québec), Canada G1K 7P4.
email: (John.Plaice,benjamin)@ift.ulaval.ca.
with Lemur, heterogeneous computing, and $N$-version programming. The heart of the paper is a formal presentation of warehouses and queries to warehouses.

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

```
X 0 d 3
where
  dimension d;
  X = 42 fby.d X+1;
end;
```

the computing process is as follows:

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

A demand is made to a warehouse $W$ for a variable $id$ and a tag $T = \{D_j : d_j\}_{j}$, 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 demand will be for $(\text{Id} : x, z : 3)$, and the demand will fail. So $(\text{Id} : x, \text{Val} : 45)$
computing, \(z : 3\) 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.

\[
X \ 0, z 3 \\
\text{where} \\
\text{dimension } w, z; \\
X = 42 \ fby. z \ X + 1; \\
\text{end;}
\]

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

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

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 $\sqsubseteq$, 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 $i = 1..n$, $d_i \leq d'_i$.

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:

$$\{Id : id, D_1 : \leq d_1, \ldots, D_n : \leq d_n\}.$$  

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

Ideally, if one is working in a heterogeneous computing environment, one would just request that a task be effected without worrying about the actual computing resources that are required. However, this can be a difficult task, as not all software will run equivalently on all machines, since version upgrades may not be undertaken consistently in all computing environments. So, what should be done?

Given our eductive approach, we can assume the existence of two warehouses, one which is similar to that used for Lemur, the other defining the available computer resources. Requests to effect a task 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 that particular version of the software. 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

A special kind of heterogeneous computing 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.

3 Dimensions, intervals and warehouses

We are now ready to formalize the interactions with a warehouse. The fundamental concept here is that of canonical interval.

**Definition 1** A dimension \( D \) is a triple \((D_D, \mathcal{D}_D, \sqsubseteq_D)\), where \( D_D \) is the name of the dimension, \( \mathcal{D}_D \) is the domain of values associated with the dimension and \( \sqsubseteq_D \) is a partial order over \( D_D \).

**Definition 2** Let \( D = (D_D, \mathcal{D}_D, \sqsubseteq_D) \) be a dimension. A point \( p \) in \( D \) is simply \((D_D : d)\), where \( d \in \mathcal{D}_D \).

**Definition 3** Let \( D = (D_D, \mathcal{D}_D, \sqsubseteq_D) \) be a dimension. An interval \( I \) in \( D \) is one of the following possibilities:
- \((D_D : \mathcal{D}_D) = \{(D_D : d)\} ; \)
- \((D_D : \geq \text{d}_{\text{min}}) = \{(D_D : d) \mid d \geq \text{d}_{\text{min}}\} ; \)
- \((D_D : \leq \text{d}_{\text{max}}) = \{(D_D : d) \mid d \leq \text{d}_{\text{max}}\} ; \)
- \((D_D : [\text{d}_{\text{min}}, \text{d}_{\text{max}}]) = \{(D_D : d) \mid d \geq \text{d}_{\text{min}} \wedge d \leq \text{d}_{\text{max}}\} . \)

An interval \( I \) will often be written \((D_D : I)\).

We will assume that each dimension has a unique name, so below, we can use \( D \) to uniquely designate a dimension. The meanings of \( D_D \) and \( \sqsubseteq_D \) should be clear.

**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.

**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 (parallelipiped) \( I \) in \( \{D_j: I_j\}_j \), 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 \( R(I) = \{D_j\}_j \).

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

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

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

Proof Trivial.

Definition 10 Let \( I \) and \( I' \) be two arbitrary intervals. Then the intersection of \( I \) and \( I' \) is given by
\[
I \cap I' = I \cap_{R(I') - R(I)} \cap I' \cap_{R(I) - R(I')}.
\]

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

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 \( \{I \in W \mid I_0 \cap I \neq \emptyset \} \).

4 Discussion

This abstract has only briefly examined four different aspects of computing. However, it should already be clear that this approach can be used in many other areas, and that deduction is in fact a very general model for computation.

This should not really come as a surprise. In fact, the user of a computer in fact spends all of his or her time requesting information (using keyboard, mouse, joystick, etc.) to be placed on some output device (screen, printer, speakers, etc.), or depositing information that can later be used when someone else requests information. The warehouse model seems to be adequate to describe what users of computers actually do. In fact, it seems to be usable at many (all) levels of computing, which gives it great promise.
The formalization in the previous section gives definitions for what a warehouse must do. However, what should the 'programming language' for dealing with a warehouse look like? This question remains unanswered, leaving us with a serious problem to be resolved.