INTRODUCTION TO INTENSIONAL PROGRAMMING

JOHN PLAICE
Département d'informatique, Université Laval
Ste-Foy (Québec), Canada G1K 7P4
E-mail: John.Plaice@ift.ulaval.ca

and

JOEY PAQUET
E-mail: Joey.Paquet@ift.ulaval.ca

ABSTRACT

Using the World Wide Web as illustration, the basic concepts of intensional programming, in which the meaning of an expression depends on the context in which it is being evaluated, are presented. A historical overview of the Lucid language, past, present and future, is given, thereby allowing an intensional understanding of dataflow programming, multidimensional programming, functional programming and object-oriented programming.

1. Introduction

Change is pervasive in many domains of computer science. In fact, problems pertaining to change are often the most difficult to solve. For example, one of the most pressing problems in software engineering is how to manage evolving requirements, specifications, software and documentation. Similarly, in artificial intelligence, some of the most difficult theoretical problems pertain to modeling changing or ambiguous information, along with belief systems. As for real-time programmers, they must control changing environments, such as physical or chemical processes.

However, there is currently a plethora of formalisms, used in a number of domains, to handle change. This situation can be contrasted with that in more established disciplines, such as physics or chemistry, where a single formalism, namely differential equations, is used to describe all continually changing phenomena.

One of the goals of mathematics is to express, in a finite, static manner, infinite or dynamic concepts. In logic, these concepts are often translated into intensional logic, or to one of its modal or temporal variants. The origins of intensional logic come from research in natural language understanding, in particular to understand such notions as the past, the future and belief. Probably the most important figures in this work are Saul Kripke and Richard Montague.

The essential aspect of intensional logic is possible world semantics. The semantics of a logical sentence depends on the possible world in which one finds oneself. For example, the sentence
It is true that $3 \times 3 = 9$.

is true in any possible world — assuming of course, that mathematics are not being redefined. However, the sentence

It is true that the number of planets is equal to 9.

is certainly not true in all possible worlds. It is quite easy to imagine a world in which the number of planets is 8 or 10. In fact, it is quite possible that the number of planets is greater than 9, but that we are only aware of 9 of them.

For Kripke, a possible world is the real, existing world, apart from the fact that certain aspects have been explicitly changed. It is this aspect that has been retained in intensional programming, which allows computations to be made according to the actual context. This notion is perfectly natural and can be found, as we shall see, in the World Wide Web or in object-oriented programming.

Possible worlds are a semantic tool used for many purposes. For example, O'Hearn and Tennent have used possible worlds to give semantics to memories in imperative programming. But it is once that we can actually change the context that we begin to have intensional programming.

The purpose of programming is to make computers effect tasks that we consider to be useful. Up to now, most programming has been based on the model of computation. The notable exception is databases, which are primarily based on the model of querying. Intensional programming is a generalization of the second model: it is based on two activities — querying and navigation.

There are already several examples of computing that use querying and navigation. For example, Web browsers and object-oriented system browsers allow one to navigate among a set of objects (Web pages in the first case, objects or classes in the second) and to make queries on the set of objects, even to deposit new information.

Intensional programming takes these ideas and goes one step further. A systematic set of operators, called intensional operators, is required to allow navigation in some sort of regular fashion. For example, on Web pages, one can go to any other page, but there are few systematic operators, apart from the BACK and FORWARD buttons for the current session.

Depending on the domain, operators might be of the form UP, DOWN, RIGHT, LEFT, BACK and FORWARD; NORTH, EAST, SOUTH and WEST; or YESTERDAY, TODAY and TOMORROW. For a visual language, navigation might take place through buttons.

An intensional programming language, be it textual or visual, retains therefore two aspects from intensional logic. First, at the syntactic level, are context-switching operators, called intensional operators. Second, at the semantic level,
possible world semantics must be provided for each intensional operator (in conjunction, of course, with other semantic notions, such as least fixpoints).

For example, in the Web, the possible contexts are the Uniform Resource Locators (URLs). The standard activity is to view a page, and what one views depends on the current context, namely the current URL. To change contexts, most browsers provide BACK, FORWARD and OPEN buttons to change context within the given session. Also, many pages provide links to other pages.

Once a language is designed, including syntax and semantics, it must be implemented. Once again, we look to the Web for illustrations.

The standard operational model for intensional programming is called eduction. It is the mode of operation for tagged demand-driven dataflow. It works as follows: The value of an object in a particular context (here called a tag) is requested. If it is already available, perhaps stored in some data warehouse or cache, then the value can be returned immediately. If it is not immediately available, then the value must be computed through some means. When the computation takes place, the values of other variables with different tags might be needed. These can be requested in parallel.

Currently, the most popular Web browser is Netscape. It shows remarkably well the concept of eduction. Basically, pages are only retrieved if they are requested. Furthermore, when a page is requested, the text for the first page is placed on the screen immediately, even if not all the text has been retrieved. Furthermore, if graphics links must be followed to complete a page, several simultaneous connections can be made to different remote hosts to recuperate the graphics. The whole interface is demand-driven, both for retrieving pages and for placing them on the screen.

To increase performance and to reduce network traffic, there are caching systems that ensure that if a page is requested several times, then it is only retrieved once. These systems can be parameterized to change how caching takes place and how pages are retired.

Eduction itself can be implemented in a number of different ways. However, the ‘natural’ implementation supposes that a distributed set of processors transmit demands one to the other through a packet network. On the Web, the Internet, which connects computers around the world, is used as medium for the various browsers. The Internet is based on several protocols, in which data circulates in packets. These are currently being standardized into the 53-byte (48 header, 5 data) packets used for Asynchronous Transfer Mode (ATM).

So intensional programming, at the implementation level, uses eduction as abstract computation model. The natural real implementation is a set of processors communicating using a packet network.

The Web was used for this initial discussion because most readers should now be familiar with how it works. Nevertheless, the Web was never consciously
designed as an intensional system. However, viewing the Web as an intensional programming system already allows us to ask questions, such as, what kinds of intensional operators could be added to the Web to make it more useful?

The concepts in this paper have arisen out of research in the development of the Lucid programming language. Lucid was invented in 1974 by Bill Wadge and Ed Ashcroft during research in program verification. Very quickly, they realized that it could be used for dataflow programming. It has been extended and modified several times in the last twenty years. It is with these developments that the very concept of intensional programming was created.

2. Lucid

The first version of Lucid was simply Landin's ISWIM (If You See What I Mean) with two new operators: next and fby. For example, the equation

\[ N = 0 \text{ fby } N+1; \]

defines \( N \) to be the infinite stream \((0, 1, 2, \ldots)\). The value of \( N \) is 0 initially, subsequently it is the previous value of \( N \), plus 1. More formally, if

\[ X = (x_0, x_1, x_2, \ldots, x_n, \ldots) \]

and

\[ Y = (y_0, y_1, y_2, \ldots, y_n, \ldots), \]

then

\[ X + Y = (x_0 + y_0, x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n, \ldots), \]

\[ \text{next } X = (x_1, x_2, x_3, \ldots, x_{n+1}, \ldots), \]

\[ X \text{ fby } Y = (x_0, y_0, y_1, \ldots, y_{n-1}, \ldots). \]

The arithmetic and boolean operators are called pointwise operators, since they apply to elements with the same index. So, for example, in the program

\[
\begin{align*}
\text{M} \\
\text{where} \\
M &= N \text{ fby } (M + \text{next } N); \\
N &= 0 \text{ fby } (N + 1); \\
\end{align*}
\]

So \( N \) is the sequence of natural numbers, \((0, 1, 2, 3, \ldots)\), and \( M \) is the running sum of the natural numbers, \((0, 1, 3, 6, \ldots)\). Note that the parentheses in the program were only added for increased readability.

Not all streams are infinite. In Lucid, to code finite streams, we use a special value called eod ('end of data'), which is placed at the end of the part of the stream of interest. For example, the stream \((0, 1, 2)\) would be written

\[ 0 \text{ fby } 1 \text{ fby } 2 \text{ fby } \text{eod} \]
Functions can be defined in the normal manner. For example, a running sum function is defined as follows:

```plaintext
runningSum(N) = M
where
    M = N fby M + next N;
end;
```

Other commonly used operators can be derived from fby and next.

The first element in a stream can be singled out (ad infinitum) using the first operator, defined as follows:

```plaintext
first X = X fby first X;
```

The result is

```plaintext
first X = (x_0, x_0, x_0, ..., x_0, ...).
```

The boolean filter wvr ('whenever') selects those elements in its first argument that correspond to true elements in its second argument:

```plaintext
X wvr Y = if first Y
    then X fby next X wvr next Y
    else next X wvr next Y
fi;
```

So, for example, if

```plaintext
Y = (true, false, false, true, false, true, ...),
```

then

```plaintext
X wvr Y = (x_0, x_3, x_5, ...).
```

The boolean filter asa ('as soon as') selects the element in its first argument that corresponds to the first true element in its second argument:

```plaintext
X asa Y = first (X wvr Y);
```

So, for example, if

```plaintext
Y = (false, false, true, false, true, ...),
```

then

```plaintext
X asa Y = (x_2, x_2, x_2, ...).
```

The sequential approach is not the only one possible. In fact, from an implementation point of view, a random access approach is more appropriate. For this reason, the basic operators are no longer fby and next, rather they are # and @.

The current index # allows one to determine the current context.

```plaintext
# = 0 fby # + 1;
```

The random access operator @ ('at') uses the second argument as index into the first:
\( X \odot Y = \text{if } Y \equiv 0 \text{ then first } X \)
\( \text{else } (\text{next } X) \odot (Y-1) \)
\( \text{fi; } \)

As a result, if \( Y \) is an integer stream

\[ (y_0, y_1, y_2, \ldots, y_n, \ldots), \]

then

\[ X \odot Y = (x_{y_0}, x_{y_1}, x_{y_2}, \ldots, x_{y_n}, \ldots). \]

We can now redefine the \texttt{fby} and \texttt{next} operators:

\[
\begin{align*}
X \texttt{ fby } Y &= \text{if } # \equiv 0 \text{ then } X \text{ else } Y \odot (#-1) \text{ fi; } \\
\text{next } X &= X \odot (#+1); 
\end{align*}
\]

The \# and \odot operators are in fact the two fundamental operators of intensional programming. The first is used for \textit{querying} and the second for \textit{navigating}, as outlined in the introduction.

It is now possible to show how eduction takes place in Lucid programs. Consider the following program:

\[
X \odot 3 \\
\text{where} \\
X = 42 \texttt{ fby } X+1; \\
\text{end;}
\]

In this example, variable \( X \) is the stream

\[ (42, 43, 44, 45, \ldots). \]

Therefore, evaluating this program should give the value 45. To see how the program is evaluated, we rewrite it so that the \# and \odot operators are visible:

\[
\begin{align*}
X \odot 3 \\
\text{where} \\
X &= \text{if } # \equiv 0 \text{ then } 42 \\
&\quad \text{else } (X+1) \odot (#-1) \\
&\quad \text{fi; } \\
\text{end; }
\end{align*}
\]

Evaluation takes place by generating successive demands for the appropriate
values of $x$, until the final computation can be effected:

$$
x\times 3\{0\} = x\{3\} = (x+1)\{2\} = x\{2\} + 1 = (x+1)\{1\} + 1 = x\{1\} + 2 = (x+1)\{0\} + 2 = x\{0\} + 3 = 42 + 3 = 45
$$

3. Indexical Lucid

In the original Lucid, contexts were simply integers. The currently available version of Lucid, called Indexical Lucid, is *multi-dimensional*. Contexts are no longer single integers, but tuples of identifier–integer pairs. The identifiers are called *dimensions*, and each intensional operator can be parameterized so that it can change the context in any given dimension. For example, the above running sum program can be modified as follows:

```
runningSum.d(N) = M
where
    M = N fby.d M + next.d N;
end;
```

The objects are no longer streams, but multidimensional streams, which can vary in an arbitrary number of dimensions.

To see how this program works, consider the following two-dimensional piece of data $A$, where dimension $da$ goes horizontally rightwards and dimension $db$ goes vertically downwards:

```
1 3 4 2 7 ...
3 1 6 7 9 ...
1 9 2 1 4 ...
: : : : : ...
```

Now, `runningSum.da(A)`, yields the following result:

```
1 4 8 10 17 ...
3 4 10 17 26 ...
1 10 12 13 17 ...
: : : : : ...
```
Since db is nowhere mentioned in runningSum.da(A), each separate value associated with dimension db can be considered to be independent. Therefore the running sum is applied independently to each row.

As for runningSum.db(A), it yields the following result:

\[
\begin{array}{ccccccc}
1 & 3 & 4 & 2 & 7 & \cdots \\
4 & 4 & 10 & 9 & 16 & \cdots \\
5 & 13 & 12 & 10 & 20 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
\]

Once again the running sum is applied independently to each column.

The two can be combined, as runningSum.da(runningSum.db(A)):

\[
\begin{array}{ccccccc}
1 & 4 & 8 & 10 & 17 & \cdots \\
4 & 8 & 18 & 27 & 43 & \cdots \\
5 & 18 & 30 & 40 & 60 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
\]

It turns out that this is the same as runningSum.da(runningSum.db(A)).

Like for the original Lucid, the operators fby.d and next.d are actually defined in terms of @.d and #.d:

\[
X \text{ fby.d } Y = \text{ if } #.d \text{ eq 0 then } X \text{ else } Y \text{ @.d } (#.d-1);
\]

\[
\text{next.d } X = X \text{ @.d } (#.d+1);
\]

The querying operator is no longer used to determine the entire context but, rather, part of the context. Only information that is needed is asked for. As for the navigation operator, it only changes part of the context.

Dimensions can also be introduced for local computations. This possibility is presented using a matrix multiplication example.

To multiply two n \times n matrices A and B we have to multiply, pointwise, the rows of A and the columns of B and add together the values produced. More precisely, the (i, j)-th element of the product is the sum of the n values

\[A_{i,k} \times B_{k,j}, \quad k \in 1..n.\]

The required program is simply

\[
\text{runningSum.k(product) @.k (n-1)}
\]

\[
\text{where dimension k};
\]

\[
\text{product} = \text{realign.j,k(A)} \times \text{realign.i,k(B)};
\]

\[
\text{realign.u,v(C) = C @.u #.v};
\]

\[
\text{runningSum.d(N) = M}
\]

\[
\text{where M = N fby.d M + next.d N};
\]

end;
In this program, matrix $A$ is turned so that its variation in dimension $y$ is instead changed into variation in dimension $z$. Similarly, $B$ is turned so that its variation in dimension $x$ is changed into variation in dimension $z$. As a result,

$$\text{product}_{i,j,k} = A_{i,j,k}^r \times B_{i,j,k}^r$$
$$= A_{i,k,k} \times B_{k,j,k}$$
$$= A_{i,k} \times B_{k,j}$$

The last equality holds since both $A$ and $B$ are supposed to be constant in dimension $k$.

The 3-dimensional product stream is collapsed into two dimensions by running a sum in the $k$ dimension, exactly as in the problem statement.

This program can be optimized so that the sums are computed in parallel; see reference 1 for more details.

4. Functions as Values

Most languages based on Landin's ISWIM are functional languages, and allow functions to be passed as normal values, either as parameters to other functions or as variables. For example, one can write a expression such as:

```plaintext
apply(sq,2)
where
    apply(f,x) = f(x);
    sq(y) = y*y;
end;
```

In this expression, the apply function is called a higher-order function, since its argument $f$ is itself a function. A function is called first-order if none of its arguments are functions. It is said to be of order $n$ if it has as functional argument of order $n - 1$. Here, apply is second-order.

There is no semantic problem in including functions as values in Lucid. Nevertheless, they were never included in any distributed versions of Lucid because the standard implementation technique for higher-order functions involves closures, which are tricky to implement.

It turns out that function applications can be implemented using eduction. For this technique to work, evaluation contexts are assumed to include lists of integers. We begin the presentation with a simple case, namely translating a well-known first-order function.

Consider the expression:

```plaintext
fib(3)
where
    fib(n) = if n<2 then 1 else fib(n-1) + fib(n-2);
end;
```
It can be compiled into an expression in which fib becomes a variable, rather than a function, which evaluates in the context of its calling tree.

For this to take place, the following two intensional operators are needed:

\[
\text{call.c}(i,E) = E \circ c \ (i :: \#.c);
\]

\[
\text{actuals.c}(E) = \text{nth}(E, \text{hd} \ \#.c) \circ \#.c \ (\text{tl} \ \#.c);
\]

All calls to function fib, including recursive calls in its definition, are numbered, using the call operator. For the formal parameter \( n \), all the effective parameters are gathered into a list, using the actuals operator.

The function becomes:

\[
\text{call.f}(0, \text{fib})
\]

where

\[
\text{fib} = \text{if } n < 2 \text{ then } 1 \text{ else call.f}(1, \text{fib}) + \text{call.f}(2, \text{fib});
\]

\[
n = \text{actuals.f}(3, n-1, n-2);
\]

end;

The initial value associated with dimension \( f \) is assumed to be the empty list \([\ ]\). Here is how the program is executed:

\[
\text{call.f}(0, \text{fib})\{f \mapsto []\}
\]

\[
= \text{fib}\{f \mapsto [0]\}
\]

\[
= (\text{call.f}(1, \text{fib}) + \text{call.f}(2, \text{fib}))\{f \mapsto [0]\}
\]

\[
= \text{fib}\{f \mapsto [1, 0]\} + \text{fib}\{f \mapsto [2, 0]\}
\]

\[
= (\text{call.f}(1, \text{fib}) + \text{call.f}(2, \text{fib}))\{f \mapsto [1, 0]\} + 1
\]

\[
= \text{fib}\{f \mapsto [1, 1, 0]\} + \text{fib}\{f \mapsto [2, 1, 0]\} + 1
\]

\[
= 1 + 1 + 1
\]

\[
= 3
\]

To understand what is really going on, one needs to understand how the values of \( n \) are computed:

\[
n\{f \mapsto [0]\} = 3
\]

\[
n\{f \mapsto [1, 0]\} = n\{f \mapsto [0]\} - 1 = 2
\]

\[
n\{f \mapsto [2, 0]\} = n\{f \mapsto [0]\} - 2 = 1
\]

\[
n\{f \mapsto [1, 1, 0]\} = n\{f \mapsto [1, 0]\} - 1 = 1
\]

\[
n\{f \mapsto [2, 1, 0]\} = n\{f \mapsto [1, 0]\} - 2 = 0
\]

A similar translation can be effected for higher-order programs. A program of order \( n \) can be translated, in \( n \) steps and using \( n \) dimensions, to a program
with no function calls. For example, the above program with `apply` is translated as follows:

```plaintext
call.s(0, apply(2))
where
   apply(x) = f(x);
   sq(y)   = y*y;
   f(z)    = actuals.s(sq(call.s(0,z)));
end;
```
and then as follows:

```plaintext
call.f(0, call.s(0, apply))
where
   apply = call.f(0,f);
   sq    = y*y;
   f     = actuals.s(call.f(0,sq));
   z     = actuals.f(x);
   y     = actuals.f(call.s(0,z));
   x     = actuals.f(2);
end;
```
Each step transforms a program of order \( n \) to a program of order \( n - 1 \).

5. Dimensions as Values

In Indexical Lucid, dimensions can be passed as parameters to functions, but they are not real values: for example, one cannot create a stream of dimensions. However, this possibility is necessary whenever one wishes to build functions that work over a variable number of dimensions. In particular, the authors' other articles in this volume, on particle simulation \(^6\) and databases \(^5\) found in this volume, require dimensions as values.

In the section of Indexical Lucid, we gave an example of computing the running sum of an integer stream \( A \) in two dimensions, \( da \) and \( db \); the expression became

```plaintext
runningSum.da(runningSum.db(A))
```
Suppose that integer stream \( B \) varied in three dimensions, \( da, db \) and \( dc \); to compute the running sum in three dimensions, one would have to write:

```plaintext
runningSum.da(runningSum.db(runningSum.dc(B)))
```
But without dimensions as values, it is impossible to compute the general running sum for an \( n \)-dimensional stream, where \( n \) is arbitrary.

A dimension stream is simply a stream whose values are dimensions. We define the general running sum program:
Fig. 1. Behavior of equal function

\[
\text{RunningSum}.d(D,N) = M \text{ asa}.d \text{ iseod}(D)
\]
where
\[
M = N \text{ fby}.d \text{ runningSum}.D(M);
\]
end;

where it is supposed that D is a stream of dimensions varying in dimension d.
RunningSum calls runningSum, in turn, for each dimension in which N varies.

For the example with stream A, the expression becomes:

\[
\text{RunningSum}.dA(da \text{ fby}.dA db fby.dA eod, A)
\]

For stream B, the expression becomes:

\[
\text{RunningSum}.dA(da fby.dA db fby.dA dc fby.dA eod, B)
\]

In each case, dimension da is a local dimension.

A second example is for testing the equality of \(n\)-dimensional finite rectangular arrays:

\[
\text{equal}.d,D(E0,E1) = s \text{ asa}.d \text{ iseod}(D)
\]
where
\[
s = E0 \text{ eq } E1 \text{ fby}.d \text{ collection}.D s;
\]
\[
\text{collection}.cE = (\text{iseod}(E) \text{ or } E) \text{ asa}.c
\]
\[
\quad (\text{iseod}(E) \text{ or } \text{not } E);
\]
end;

The behavior of equal is illustrated by Figure 1.

6. Definitions as Values

One of the most important developments in software engineering in the last ten years has been the widespread introduction of object-oriented languages, tools and methodologies.
The basic idea in object-oriented programming is that a message is sent to an object, and the object must determine what to do with the message that it has received. From an intensional point of view, the definition of a message depends on the context, here the object which has received the message.

It is no longer simply the case that there is one definition whose meaning varies with the context: the definition itself varies. This idea is found not only in object-oriented programming, but also in spreadsheets and in multiple-version software.

Given that the research in this area is not really mature, we give a brief outline of the issues, encouraging readers to develop these ideas. See also the article on object-oriented programming in this volume.

We first introduce a new kind of value: the definition. If $E$ is an expression, $E$ is also an expression, which evaluates to $E$. For example, if the value of $x$ is 3 and the value of $y$ is 5, then the value of $x+y$ is 8, while the value of $E(x+y)$ is $x+y$.

This concept is not new. The & operator corresponds quite closely to QUOTE in Lisp. More commonly, an object file in any computer system can be seen either as a stream of characters or as a function that can be applied to its arguments.

For evaluation to take place with multiple definitions, eduction must take place in two stages. First, a variable's definition must be deduced, then that definition must be evaluated in the appropriate context, possibly provoking other eductive steps.

7. Conclusion

The concept of intensional programming was introduced with a presentation of the World Wide Web. Querying and navigation, using intensional operators with possible-world semantics, form the basis for intensional programming. These operators are implemented using a form of demand-driven computation, called eduction, ideally through some sort of packet-driven network.

The most developed intensional programming language, Lucid and its extensions, was presented in a historical setting to show the development of intensional concepts. In so doing, dataflow programming, multidimensional programming, functional programming and object-oriented programming were all discussed from an intensional point of view.

It should be clear that intensional programming is a concept whose time has come. Throughout computing, people are using intensional concepts in an unconscious manner. Understanding these concepts on a wide scale should help
better understand notions of change in computer science.

8. References