Intensional Programming Languages*

Panos Rondogiannis  William W. Wadge

Abstract

This article introduces the basic notions behind the intensional programming paradigm. Intensional Logic—which is the underlying theoretical framework of intensional languages—is described and motivated by examples. One-dimensional and multidimensional intensional languages are discussed and existing intensional systems are surveyed. The article concludes with a presentation of the most successful application areas of intensional programming.

Keywords: Intensional Logic, Intensional Languages, Dataflow, Functional Programming.

1 Introduction

One of the major challenges of computer science is the design of programming language paradigms that would free the programmers from low-level, machine-related tasks. Such paradigms are usually based on sound mathematical foundations and they allow for a cleaner and more declarative way of programming. The intensional programming paradigm is a relatively recent one and has its foundations in the area of intensional logic [30].

Although it is not easy to give a brief and non-technical explanation of what an intensional language really is, we could say that one of the main characteristics of the paradigm is that it deals with infinite entities of ordinary data values. Such entities could be a stream of numbers, a two-dimensional table of characters, a tree of strings, and so on. These entities are treated as first-class objects by intensional languages: two streams of numbers can be added together as if they were ordinary data values, functions can be applied on infinite tables and trees, and so on. Due to the above characteristic, intensional languages are especially appropriate for describing the behavior of systems that change with time or physical phenomena that depend on more than one parameter (such as time, space, temperature, etc). Moreover, as intensional languages are based on solid mathematical foundations (i.e., intensional logic), they promote a purer and more declarative way of programming than traditional imperative programming languages.

In this article we introduce the basic principles underlying intensional languages. §2 presents at an intuitive level the area of mathematical logic known as intensional logic. §3 introduces Lucid, an intensional language which is based on the notion of stream, and GLU, an extension of Lucid which supports multidimensional entities. §4 discusses the main issues in implementing intensional languages. §5 surveys existing intensional programming systems and their applications. The article concludes with a discussion of interesting research problems in the area of intensional programming.

2 Intensional Logic

Intensional logic [30, 8] is a branch of mathematical logic which has been used in order to describe concisely context-dependent entities. The initial motivation for the development of intensional logic was the formal description of the meaning of natural languages. Many sentences of the languages we use in everyday life are often ambiguous, i.e., they can be interpreted in different

ways under different situations or from different people. This has led many scientists to believe that natural languages are not formal from a mathematical point of view, and it is therefore impossible to analyze and study them systematically.

However, Montague (the father of intensional logic) firmly believed that natural languages have a mathematical basis which is analogous to that of artificial languages (e.g., computer languages). Montague’s work is extremely technical and is therefore difficult to present in detail (a very good introduction to the subject can be found in [8]). Intuitively speaking, we can say that Montague developed a formal system for effectively describing entities whose value depends on implicit contexts. The meaning of natural language expressions often depends on such hidden parameters. Consider for example the expression:

*Athens is the capital of Greece*

The truth value of this sentence depends on the time at which it was uttered. Today it is certainly true, but there existed time points in the past when Greece had a different capital. In other words, the truth value of the above sentence is time-dependent; formally, the meaning of the sentence is a function from the set of time-points to the set of Boolean values.

One can easily think of sentences or expressions that depend on more than one hidden context, such as *time*, *space*, *audience*, and so on. For example, the meaning of the expression:

*The value of the temperature*

can be thought of as a two dimensional table of the form:

<table>
<thead>
<tr>
<th></th>
<th>London</th>
<th>Paris</th>
<th>Rome</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/98</td>
<td>-3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2/1/98</td>
<td>-2</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3/1/98</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In other words, the meaning of the above expression is a function in \((T \times S) \rightarrow D\), where \(T\) is the set of time-points, \(S\) is the set of space points and \(D\) is a set of values that the temperature can obtain.

Generalizing the above discussion, we can say that in many cases, the meaning of a natural language expression is a function from contexts (also called possible worlds) to values. Such a function is called the *intension* of the expression. The value of the intension at a particular context is called the *extension* of the expression at that particular context. So, in the temperature example, the intension of the expression is the whole table; the extension of the expression in the time point 1/1/98 and at the space coordinate of *Rome* is 7.

This context-dependent character of natural languages is possibly due to practical reasons: when we talk to each other, it would be completely impractical if we had to explain explicitly the context under which our sentences are true. So for example when we say “the prime minister”, we most probably mean the prime minister at the time we talk and in the place where we talk (although we do not usually have to specify explicitly these two parameters).

Another characteristic of natural languages is that they use *context-manipulation operators* in order to alter the present context. For example, we say “Yesterday’s temperature” in order to refer to the value of the temperature the previous day. Words such as *north*, *tomorrow* and *next* are often used to change the context. Again, using these operators we avoid referring to the new context explicitly, but we specify it as a function of the present context.

Concluding this brief and unavoidably informal introduction to the intricacies of natural language, the reader should keep in mind that most of the expressions used in everyday talk are context-dependent. Intensional logic is a branch of mathematical logic which captures the context-dependent characteristics of natural languages. But if intensional logic is so close to real languages, why not use it in order to design new, more powerful and expressive programming languages? This topic is further discussed in the coming sections of this article.
3 Intensional Languages

Most existing programming languages are machine-oriented: the programmer is often obliged to know many aspects of the architecture of the underlying machine in order to be able to write even the simplest programs. Consider for example the notions of variable and assignment in an imperative language. Programmers usually think of variables as memory locations and assignments as commands that alter the contents of such locations. Another example is arrays: programmers usually think of them as consecutive memory locations which they can access and alter at will.

Therefore, many aspects of traditional programming languages, although widely used, are usually understood operationally. In other words, traditional programming languages have the architecture of the underlying machine very explicit in their design. We are interested in a programming paradigm that would hide unnecessary operational ideas from the programmer, in the same way that a natural language helps people hide many unnecessary context details during everyday talk. One idea that comes to mind is to develop programming languages based on intensional logic. Before describing in detail the philosophy behind such a language, we present a simple intensional program that computes the infinite sequence of all natural numbers:

\[
\text{result} = \text{nat}; \\
\text{nat} = 0 \text{ fby (nat+1)};
\]

Notice the use of the operator \texttt{fby} (read followed by) in the above program. The semantics of the operator will be soon defined precisely. For the time being, we provide the intuitive reading of the program:

The result of the program is the sequence of natural numbers. The first value of the natural number sequence is 0. The next value in the sequence can be produced by adding 1 to the previous value of the sequence.

Notice that the usual mathematical definition of the sequence of natural numbers involves the use of a time index:

\[
\text{nat}_0 = 0 \\
\text{nat}_{t+1} = \text{nat}_t + 1
\]

The intensional program avoids the explicit use of the time index. The sequence of natural numbers is defined using the temporal operator \texttt{fby} rather than using subscripts (i.e., indices).

The example given above is actually a program of the intensional language Lucid [33] (which was probably the first such language). The value of a Lucid expression depends on a hidden time parameter. In other words, the meanings of entities in Lucid programs are not ordinary data values but infinite sequences of ordinary values.

The statements of a Lucid program are equations defining individual and function variables, required to be true at every context (time point). Ordinary data operations (such as +, *, and if-then-else) are referentially transparent. This means, for example, that the value of \texttt{x + y} at time point \(t\) is the sum of the values of \texttt{x} and \texttt{y} at the same time point \(t\).

The basic Lucid context-switching operators are \texttt{first}, \texttt{next}, and \texttt{fby}. The operator \texttt{first} switches us to time point 0, \texttt{next} takes us from \(t\) to \(t+1\). The operator \texttt{fby} takes us back from \(t+1\) to \(t\) (giving us the value of its second operand at that point) or from 0 to 0, giving us the value of its first operand.

Let \(x\) and \(y\) be sequences. Then, the above ideas are formalized by the following semantic equations:

\[
(x + y)_t = x_t + y_t \\
\text{first}(x)_t = x_0 \\
\text{next}(x)_t = x_{t+1} \\
(x \text{ fby } y)_t = \begin{cases} 
    x_0 & \text{if } t = 0 \\
    y_{t-1} & \text{if } t > 0 
\end{cases}
\]
These equations constitute what we call the *indexical semantics* of the operations; they define the extensions of the result of an operation in terms of the extensions of the operands.

A Lucid intension \( x \) can be thought of as a value which is varying over time, for example, a loop variable in an iterative computation. Thus \( x_0 \) is the initial value of \( x \), \( x_1 \) is the value after the first iteration step, and \( x_2, x_3, x_4 \), etc., are the values after subsequent steps.

The \( \text{fby} \) operator allows us to express many iterative algorithms concisely; the first operand of the \( \text{fby} \) gives the initial value, and the second operand specifies the way in which each succeeding value is determined by the current value. For example, the following program computes the stream \((1, 1, 2, 3, 5, \ldots)\) of all Fibonacci numbers:

\[
\text{result} = \text{fib} \\
\text{fib} = 1 \text{ fby (fib+g)} \\
\text{g} = 0 \text{ fby fib}
\]

The Lucid language also supports user-defined functions that operate on sequences and return sequences as results. For example, consider the following program:

\[
\text{result} = \text{fact(nat)}; \\
\text{nat} = 0 \text{ fby (nat+1)}; \\
\text{fact}(x) = \text{if } (x<2) \text{ then } 1 \text{ else } x \text{ fact}(x-1) \text{ fi};
\]

The output of the program is the infinite stream of the factorials of the natural numbers, i.e., the sequence \((0!, 1!, 2!, \ldots)\).

The language Lucid presented so far obviously differs in many aspects from the usual imperative programming languages. For example, the notion of variables that change values using assignment commands in imperative programming, is modeled in Lucid by sequences that possess different values at different time-points. Similarly, the notion of infinite sequence that Lucid supports can be used to model one-dimensional arrays of traditional programming languages. But how about two, or three or generally \( n \)-dimensional arrays? Lucid as described above is a one-dimensional intensional language. For many applications such a language might be enough. However, there exist many natural problems which can be better thought of and solved if viewed from a multidimensional perspective.

Lucid has recently been extended to support more than one-dimensional entities [3, 4]. This new extended language was named GLU and its first implementation was developed in SRI. GLU allows the user to declare new dimensions and to define multidimensional entities that vary across these dimensions. So, a two-dimensional entity can be thought as an infinite table, a three-dimensional one as a cube extending infinitely across the three dimensions, and so on. One can perform various operations with arguments of such higher-dimensional entities or even define functions that take them as parameters and return new entities as results. Moreover, the new language supports intensional operators that work along each different dimension.

As an illustration of the expressive power of GLU, we consider a simple program which first appeared in [11] and which models the problem of heat transfer in a solid. Suppose that we have a long thin metal rod which initially has temperature 0 and whose left-hand end touches a heat source with temperature 100. As the heat is transferred, the temperature at the various points of the rod changes. In other words, the temperature can be thought of as a two-dimensional entity because it depends on the spatial position on the rod that we are interested in as well as on the time point (the temperature increases as time passes). It can be shown that the temperature of the rod as a function of time and space, is given by the following recurrence relations (where \( k \) is a small constant related to the physical properties of the rod):

\[
\begin{align*}
\text{Temp}_{t+1,s+1} &= k \times \text{Temp}_{t,s} - (1 - 2 \times k) \times \text{Temp}_{t,s+1} + k \times \text{Temp}_{t,s+2} \\
\text{Temp}_{t,0} &= 100 \\
\text{Temp}_{0,s+1} &= 0
\end{align*}
\]
The GLU program that models the above equations is the following:\textsuperscript{1}

\begin{verbatim}

dimension time, space;
result = temp;
temp = 100.0 fby.space (0.0 fby.time k*temp - (1.0-2.0*k)*(next.space temp) + k*(next.space next.space temp)));
k = 0:3;
\end{verbatim}

The first line in the above example declares that the program uses two dimensions, namely \texttt{time} and \texttt{space}. Notice the new operators that appear in the definition of \texttt{temp: fby.space, fby.time} and \texttt{next.space}. The semantics of these operators are straightforward generalizations of the semantics of the corresponding Lucid operators.

It should be noted here that the temperature example described above is expressed very compactly and naturally in the GLU formalism. The solution of such a problem in a traditional imperative language would most probably require the use of a two-dimensional array together with the use of for loops in order to fill the entries of the array.

With the above example we conclude our brief introduction to the syntax and the (informal) semantics of intensional languages. Clearly, one can easily propose alternative intensional languages in which the underlying context space is more complicated and which can be used in different application domains. One such possibility is \textit{branching time}, in which intensions are tree-like. Branching-time intensional languages have been proposed in the literature and have been shown to have interesting applications [34, 28, 31, 27, 26]. However, a further discussion of branching-time is outside the scope of this article.

4 Implementation of Intensional Languages

The infinite nature of intensional programming languages suggests that their implementation might be problematic. For example, we know that the output of a Lucid program is an infinite sequence of ordinary data values. How can such a sequence be computed and delivered to the user? This is obviously not possible as it would require an infinite amount of time. However we can always expect to be able to compute larger and larger parts of the desired output of the program. An implementation can start by computing the first value of the output sequence, then the second, and so on.

The traditional implementation of Lucid programs is based on a computational model known as \textit{eduction} [33]. We illustrate the main idea of eduction using an example. Suppose we want to calculate the first three Fibonacci numbers. Moreover, assume that we have implemented a simple interpreter \texttt{EVAL}, which computes the output of Lucid programs at successive time-points. The interpreter uses the definitions in a program as well as the semantics of the Lucid operators in order to calculate the output sequence. So, the output of the Fibonacci program of the previous section at time 0 can be calculated as follows:

\begin{verbatim}
EVAL(fib,0)
= EVAL((1 fby (fib+g)),0)
= 1
\end{verbatim}

\textsuperscript{1}The syntax we use is slightly different from the actual GLU syntax.
The time 1 output of the program is calculated as follows:

\[
\text{EVAL}(\text{fib}, 1) \\
= \text{EVAL}((1 \text{ fby } (\text{fib+g})), 1) \\
= \text{EVAL}(\text{fib} + \text{g}, 0) \\
= \text{EVAL}(\text{fib}, 0) + \text{EVAL}(\text{g}, 0) \\
= \text{EVAL}((1 \text{ fby } (\text{fib+g})), 0) + \text{EVAL}(0 \text{ fby } \text{fib}, 0) \\
= 1 + 0 \\
= 1
\]

A careful examination of the above steps reveals that some computations can take place in more than one occasion. This suggests that an efficient implementation of an intensional language should store values of variables that have been computed under specific contexts, so that these results will be available if demanded later on during evaluation. The process of storing intermediate results is known as warehousing and the data structure that is used for this purpose (usually a hash-table) is known as the warehouse. Maintaining the warehouse during execution is not always an easy task: garbage collection is often required as the table tends to get full with old entries (which may be useless for future calculations). Many techniques have been devised for managing the warehouse component of the implementation and the interested reader is referred to [33, 6].

When one considers the multidimensional version of Lucid (i.e., GLU) things become more complicated in terms of implementation. The evaluator has to compute the values of variables under more than one dimension. Moreover, the warehouse has to be more complicated as it now stores the values of variables under more complicated contexts. A new problem that appears now is the so-called dimensionality of variables, i.e., knowing in advance on exactly which dimensions a variable depends on. Knowing the exact dimensionality of program variables is important because it helps in reducing the cost associated with the warehouse operations. A promising dimensionality algorithm is reported in [7].

Notice that the main characteristic of eduction is that it computes the value of expressions with respect to contexts. There exists a class of hardware architectures (namely the dataflow one [13, 2]), that efficiently supports such execution with respect to context. In other words, dataflow machines are ideal candidates on which eduction can be implemented.

Concluding this section we should mention that although dataflow architectures are probably the most appropriate hardware platforms on which an intensional language can be executed, it is easy to implement eduction efficiently on traditional architectures (see, for example, [27, 24]).

5 Existing Intensional Systems and their Applications

As we have already seen in previous sections of this article, Lucid [33, 5] was the first (to our knowledge) intensional language developed. Lucid is actually a functional-intensional language, in the sense that it supports (first-order) user-defined functions which operate on streams. The most comprehensive description of the language, its semantics, its applications and its potential extensions is [33] (the Lucid book). From the programs in the Lucid book it becomes obvious that the language is especially appropriate for dataflow-like computations. Moreover, due to its temporal nature, the language appears to be appropriate for developing interactive software and potentially real-time applications. The appendix of the Lucid book describes the implementation of an interactive vi-like screen editor in Lucid (notice that at the time the book was written, most existing functional programming languages were not capable of efficiently executing applications of this type).

Since its inception, Lucid has been extended in several ways. Its variants have been used to specify 3D spreadsheets [9], attribute grammars [29], real-time systems [10, 14], database systems [21], and so on. The most vital extension to the language was the addition of multiple dimensions, which resulted in the language GLU (Granular Lucid) [3, 4]. A GLU program consists of a declarative part (which is in fact a program in multidimensional Lucid) and an imperative
part. The declarative part “glues” together sequential functions that are specified in an imperative language. GLU has been used for the development of real-world applications [1, 23], [4, Chapter 7].

Logic programming is another programming paradigm which has benefited from its interaction with intensional logic. Many intensional logic programming languages have been proposed [16]. A temporal logic programming language that was influenced by the style of Lucid is Chronolog [32]. The theory behind Chronolog is very well understood and developed [14, 18]. Many extensions of basic Chronolog have been proposed (to handle integer time [20], to provide multiple dimensions in the style of GLU [17], to use choice predicates that support a dataflow style of computation [19], to provide branching-time [25], to allow uncertainty to be expressed using rules with disjunction in the heads [12], and so on). However, despite its theoretical advances, Chronolog has not been evaluated in practical applications of significant size, that would help reveal its potential as well as its deficiencies.

Another area in which intensional programming appears to offer significant benefits is the area of version control [22]. The intensional versioning approach described in [22] has recently found applications in the evolving area of Internet computing. One example application in this domain is the development of the language IHTML (Intensional HTML) [35], a high-level Web authoring language. The advantage of IHTML over conventional HTML is that it allows practical specification of Web pages that can exist in many different versions. Web sites created by IHTML are easier to maintain and require significantly less space when compared to the sites created by cloning conventional HTML files.

6 Conclusions

In this article we have presented the basic principles of intensional programming languages. The distinguishing characteristic of intensional languages is that the basic entities they manipulate are intensions; that is functions from a context space to a set of data values. Depending on the context space, one can obtain different languages that can serve for different types of applications. In this article two such examples are given: the language Lucid, whose context space is a set of time-points, and the language GLU, which allows for dimensions other than time.

The area of intensional programming is continuously evolving and new interesting problems are posed. Some of the most important ones are: techniques for efficient implementations of intensional languages, relationship to Web programming, application domains that would benefit from the interaction with intensional programming, and relationships with the more traditional programming paradigms. Answers to the above problems would result in a better understanding of the field and would reveal its real potential.

References


