

## NEW FRONTIERS IN INTENSIONAL PROGRAMMING

Bill Wadge  
University of Victoria

### BIRTH OF A PARADIGM

Intensional Programming (IP) was born in 1986 at Arizona State - or at least that's when we came up with the name. We'd actually been doing it for some time already.

Tony Faustini and I had been discussing whether or not Lucid is really a "functional" programming language. This is quite an interesting topic, and I'll have something to say about it a bit later. Suddenly, we asked ourselves why should we be begging to be let on board someone else's bandwagon. Lucid really is different from SASL, Miranda, Haskell et al. They're extensional programming languages, and Lucid is an .. \*intensional\* programming language. Intensional programming ... what a concept!

At first this coinage was almost tongue in cheek. But the more we thought about it, the more sense it made. There were already several people in the so-called "Lucid Community" working systems which could hardly be called mere extensions of Lucid - for example, Chronolog, or Tao's attribute grammar compiler compiler. What they all had in common, though, was that they were inspired by Intensional Logic (IL), the semantics and even the implementations were based on the IL notion of "possible world", and finally intensional operators appeared in object-language expressions.

This is fairly vague, of course, but clear enough to be the definition of a paradigm (after all, what exactly is OOP?). For that matter, we shouldn't expect a paradigm to have too precise a definition - then it's a dead-end as far as research. IP, like IL itself, is open ended.

IP is a relatively new paradigm but already it's produced some promising ideas, some of which are better than half-baked. In this paper we'll try to give a survey of the various directions which are being followed, or should be followed. Some of this are familiar to you, others are more recent brainwaves which no one has had the time (or been crazy enough) to pursue. I don't want to claim credit from all these ideas and in fact the better they are, the more people there are who contributed.

I fear that some of what I say will be a (dare I say) rehash of ideas that appeared in the Lucid book or earlier. But many of these ideas are (like typical hash) a bit lumpy, and could use another trip through the grinder.

### WHAT IS INTENSIONAL PROGRAMMING?

It goes without saying that you can't understand intensional programming without understanding intensional logic. Furthermore, it is not enough to have just a technical understanding of the definitions. You have to have a real feel for it. This is not easy to do. Writing intensional programs is a good way.

The usual definition of IL is that it's the logic of expressions whose meaning depends on a context. This is true enough, but doesn't necessarily convey the spirit. A better definition I think is that it is the logic of referentially opaque operators.

For example, let  $Y$  be the "yesterday's" operator in expressions like "exceeded yesterday's trading volume by three million". This 'function' has interesting formal properties. For example, it's quite plausible that  $y(a+b) = y(a) + y(b)$ ; the meaning of  $y(y(a))$  is clear enough, and if  $m$  is the "tomorrow's" function,  $y(m(a)) = m(y(a))$ .

On the other hand, basic principles such as  $t=g \rightarrow y(t)=y(g)$  fail (imagine that  $t$  is the temperature and  $g$  your golf score). It seems glaringly obvious that  $y$  is not a function: the value of the argument does not determine the value of the result.



How do we make sense of these functions-that-aren't-functions? This is the contradiction at the heart of Intensional Logic. To me, it still has an air of mystery or magic.

Now in fact, there is a simple way to resolve the paradox. We introduce the concept of an \*intension\*, an entity that varies over a universe  $U$  of "contexts" or "possible worlds". An intension is an element of  $(U \rightarrow D)$  and an intensional operator is an a function from intensions to intensions; an element of  $(U \rightarrow D) \rightarrow (U \rightarrow D)$

This formalization makes it all respectable again (this is due to Montague and Scott, and is far more general than Kripke's original accessibility relations approach). It also makes it clear how much potential it has. For one thing  $(U \rightarrow D) \rightarrow (U \rightarrow D)$  is so much bigger than  $D \rightarrow D$ ; and  $U$  itself can be any set.

At the same time, however, I think it fails to capture the real essence of IL. Formally, there is no problem. But anyone who thinks of intensions exclusively as maps from  $U$  to  $D$  has missed the point. Some of us remember Calvin Ostrum, the brilliant Waterloo student who as an undergraduate produced the first workable eductive Lucid interpreter. The irony is that he had no real feel for the language; he understood it like his interpreter understood it, from a purely formal viewpoint. We nicknamed this approach "Calvinism" but soon realized that the heresy is widespread. In fact many people who take up Lucid never get beyond a purely formal, Calvinistic understanding. To them it seems that Lucid is just a bizarre language for computing with "infinite sequences".

Even convinced Lucidophiles have trouble with nested iteration. Partly, this is because "is current" is not ISWIM (more about it later). But more likely, their attempts are sabotaged by residual (even unconscious) Calvinism.

The key is to understand IL from both the formal and intuitive point of view. This is the real strength of the paradigm - that there are two radically different but complementary approaches to writing and understanding programs. For that matter, the two viewpoints aid us not just in writing programs, but in designing new forms of intensional programming. I hope that in what follows you can see the interplay between intuitions and formalism: informal intensional ideas suggest new universes  $U$  and new maps in  $(U \rightarrow D) \rightarrow (U \rightarrow D)$ ; and new formal objects suggest in turn new intuitions.

## TIME IS ON OUR SIDE

One promising avenue is considering a universe of timepoints richer than the traditional Lucid natural numbers. A simple example is negative time, using the integers.

At first negative time seems "messy and weird" to cite a famous phrase, but in fact there are excellent reasons. For one thing, many recurrence algorithms need two or more initial values and the negative points are useful for "pre-initialization". Secondly, negative space coordinates are clearly desirable, we often want to convert between time and space. And finally, sometimes we want to define an recurrence which goes in both directions. Back to the future, and forward to the past!

A more problematic area is multidimensional time, for nested iterations. I have always felt unhappy about the classical is-current construct: it is not pure ISWIM. (this probably explain why so few people understand it, or think it is messy and weird). Recently I worked out a pure ISWIM approach which makes current a real operator and, is I hope, both more general and easier to understand than the standard approach. A possible source of controversy: it involves redefining as-soon-as and other operators.

One idea which is very natural is to have continuous time - for doing genuine numerical analysis. Mathematically, there isn't much problem, the trouble is implementation. Data stored in the warehouse will be tagged with a real number, but since the representation is approximate, how can you retrieve it? Dave Ralston suggested an intriguing idea: implement the warehouse with binary trees (data ordered by tag) and interpolate when you don't have an exact match.

Recently Tony Faustini and I worked out the details of branching time: in which each instant has more than one



successor. This could be useful for describing nondeterministic iterative algorithms, say in playing a game, or searching for a solution to a bin-packing problem.

This is also a fairly natural idea but one which, like so many others, is impractical if you formulate it in the obvious way. In this case the simple minded approach is to introduce a family of next operators and a super followed-by which has a number of right-hand arguments. The resulting programs each look like a dog's breakfast.

Faustini and I hit on a more sophisticated approach: a single next operator which returns a vector of values, and a followed-by which takes a vector as its second argument. These vectors vary not in space but in what we call the "choice" dimension. Values which vary in this dimension can be thought of as "nondeterministic" values.

## LUCID GETS REAL

John Plaice and I made some real progress here, and have some ideas for real-time Lucid based on LUSTRE. LUSTRE was based in turn on ordinary Lucid but is not ISWIM. Real-time Lucid is ISWIM (a more accurate title is "reactive system" Lucid).

Oddly enough, the design does not involve enriching the universe of timepoints. Instead, it uses timestamps, and can also be understood in terms of hiatons. Furthermore the Faustini/Lewis proposal can be understood as a refinement.

A hiaton, of course, is a pause object - a sort of null value to indicate that there is nothing to report, or that a signal is not yet available. With hiatonic streams time sensitive operators like the first-come first-served merge are mathematically respectable. But hiatons are not a very practical implementation technique - the warehouse might quickly fill up with zillions of hiatons, all neatly tagged and labelled.

The alternative is to use timestamps - each daton has a little number stamped on it to indicate the clock time at which it was produced. In terms of education, timestamps are handled very simply: you have two kinds of demands: one for the value of a variable (in a given context), and one for its timestamp. These two aspects are handled separately.

We do however, need a bit of ingenuity to handle, with timestamps, everything that can be done with hiatons. The race merge is a case in point. Naively, we decide which comes first by examining their stamps. But if we are dealing with input which hasn't arrived, we don't know what the stamp is. We solve it by storing partial information about stamps. For example, if the clock says 1527 and the 8th value of X has not arrived, we know nothing about the 8th value. But we do know that the stamp is  $> 1527$ ; so we store as the stamp a special partially undefined integer which we might call "1527+".

The partial information may be enough to make some decisions. If not, we can demand that it be improved, and update the stored information when our demand is answered. Updating the stored value sounds like assignment, but it is respectable: we're just improving the information, not contradicting it.

## SPACE, NOT THE FINAL FRONTIER

This will probably be covered by other speakers. But I will discuss some new developments with the Intensional Spreadsheet and use it to explain the nested and non-nested approaches to the extra space dimensions (the source of so much controversy). I'll argue that it is not an either-or choice.

W. Du and I have also been looking into "branching space". In terms of the spreadsheet, it means that you can "click" on any cell and have it open up to reveal a whole subs spreadsheet; and each of those can in turn be opened up, ad infinitum. This would be very useful for spreadsheets: it means you could have a hierarchically structured spreadsheet. None of the commercial sheets allow this. The same structure also appears in some distributed programming algorithms, where it is called a "pyramid".

I have also discovered a scheme for having finite vectors and matrices, without using special terminating objects. The



idea is, roughly, "spacestamps". Suppose, for simplicity, that there is one space dimension. Then each value of a "vector"  $V$  is stamped with the length of  $V$ : the first out of range space index. These stamps are calculated and stored much like the timestamps mentioned earlier.

The advantage is that it is possible to calculate the length of a vector without necessarily evaluating all the components. This means in turn that we can have finite vectors with holes - elements whose value is bottom. The approach even handles infinite vectors if we use partial spacestamps, as described above.

Incidentally Du and Ashcroft (and others probably) have been investigating infinitely many nonnested space dimensions. They both concluded you need a very general is-current parametrized by the set of dimensions in which freezing does not occur. This is either a great breakthrough, or messy and weird, and I am not sure which.

#### A TRIBUTE TO GRAMMARS

Senhua Tao has made good progress with educative attribute grammars. The basic idea, of course, is that the nodes in the parse tree can be treated as possible worlds or indices (this work has been reported on in earlier ISLIPs).

She will be talking to you herself about the latest results, concerning time varying attributes, incremental evaluation, and, more recently, educative least fixed point evaluation. It is possible that (after spreadsheets) attribute grammars will be the second area in which IP ideas make a recognized contribution.

#### A VERSION THERAPY

This has a lot of potential. The idea is that the different versions of an object (say, a piece of software) can be thought of as the values the object has in different possible worlds. The universe of versions is partially (not necessarily linearly) ordered by the refinement relation:  $V1 \leq V2$  if  $V2$  is further on some refinement path which passes through  $V1$ .

For example, suppose we have a program in portable C which runs on the mac with only a few changes - and on the sun, with other changes. Then in addition to the plain or vanilla version we have the "mac" version of our software, and the "sun" version. Also, suppose that on the mac it is a little slow and that to speed it up we need to make some more extensive changes, including (say) eliminating some run time checks. We therefore have a subversion of the mac version, namely the "mac%fast" version.

Clearly, each of these versions is useful; none can be unconditionally discarded in favour of any of the others. Furthermore, we cannot order them linearly. The mac and sun versions are each refinements of the plain version but are incomparable; and the mac%fast version is a refinement of the mac version (and the plain one) but is also incomparable with the sun version.

We want them all around simultaneously but we don't want to make three copies of every line of source and every .o file. So we proceed as follows: when configuring a version  $v$  of the program, we look for version  $v$  of each of its parts. But if a part does not exist in  $v$ , we take the most relevant version of the part available. In the case of mac%fast, we look for a mac%fast version; if there isn't one, we settle for a mac version; otherwise we take a plain one.

The version approximation relation defines an inheritance hierarchy, which allows the code and object sharing. John Plaice and I designed an extension of my sloth environment which automatically handles versions along these lines. The crucial point is that this is an indexical solution to version management, one which could be used in conjunction with any of the other IP ideas described in this paper. Versions are not quite just another field in the tag, because of the partial order, but implementing them involves fairly modest changes to the basic education idea. Pieces of different versions of the same entity can coexist in the warehouse, all neatly labelled.



## INTENSIONAL LOGIC PROGRAMMING

This is one area of IP where we are not alone - although we are pioneers (the first Chronolog report was being circulated in 1985). InTense (an impure Chronolog-like language) is out, Mehmet Orgun has sewed up the semantic problems and Dave Ralston is making headway towards a purely eductive implementation.

It seems that ILP permits a very natural notion of module with internal memory - one which is semantically the analog of the where clause. Also, with this module we may have a declaratively respectable approach to Object-Oriented logic programming.

Another idea which shows promise is Intensional Constraint Logic programming. This would allow terms as well as predicates to be intensions, and makes sense provided the universe of possible worlds is finite. This may often be the case, eg a finite set of spreadsheet cells, or the nodes of a parse tree.

Bjorn Freeman-Benson is currently working on a pure constraint version of the idea. It means, for example, one could write temporal constraints which guide the way in which a ThingLab-like system changes with time.

## HIGHER ORDER FUNCTIONS

Our nemesis. Many people unfairly dismiss our languages because they are first order. I recall one friend (at MCC at the time) who tried to interest a colleague in Lucid. The colleague returned the Lucid paper unread when he learned it lacked higher order functions.

I don't have any breakthroughs, but at least I will try to explain the problems.

On the semantic level, it is not at all clear what higher order intensional ISWIM should be. Even in the typed case, there are problems. The naive approach is to have two formation rules: if  $V$  and  $W$  are types, then so is  $V \rightarrow W$ ; and if  $V$  is a type so is  $\text{stream}(V)$  (assuming the universe  $U$  of possible worlds is the natural numbers).

I argue that this is too rich. A simple example: we shouldn't be forced to have streams of streams since this corresponds to having a richer universe, which we may or may not want. Why should having higher order functions around force us to deal with the subtleties of nested iteration?

More problematic, however, is the fact that  $\text{stream}(D \rightarrow D)$  is really a sub-type of  $\text{stream}(D) \rightarrow \text{stream}(D)$  ( $D$  an extensional or scalar type). The elements of the former correspond exactly to the \*synchronic\* elements of the latter. We don't want to duplicate them. It is for similar reasons that ordinary Lucid does not have a scalar type: elements of  $D$  are already available as the constant streams.

Furthermore, I will argue that  $\text{stream}(\text{stream}(D) \rightarrow \text{stream}(D))$  is not needed. These things are streams of filters. You want to apply them to streams of scalars, this must be done pointwise, with sampling. But then there are examples of different filter streams which always give the same result when used on the same input scalar stream - not a desirable situation. At any rate, since  $\text{stream}(D) \rightarrow \text{stream}(D)$  is "really" a supertype of  $\text{stream}(D \rightarrow D)$  (see above), and the latter is a contains  $\text{stream}(D)$ , if we allowed  $\text{stream}(\text{stream}(D) \rightarrow \text{stream}(D))$  we'd have  $\text{stream}(\text{stream}(D))$  sneaking in the back door.

I am sure there is a reasonable notion of higher order intensional logic which avoids the mess and weirdness just described. But it is important to realize that there is more to the problem than representing streams as lists and using Haskell. Higher order intensional logic cannot be trivially reduced to higher order extensional logic.

The real difficulty, however, is implementation: we still don't have a place coding scheme that handles higher order programs. Not even finitely typed higher order functions. Not even second order programs (though I think this case at least is not hard). This is one of our greatest challenges.



## WORLDS WITHOUT END

A famous apocryphal philosophy exam question goes as follows: "describe the universe and give two examples". We may be close to answering the question!

As we explained, the various different ideas for IP are based on different universes of possible worlds. Up to now, each of these proposals has to be handmade, so to speak. Our denotational semantics is generic but our implementations have to be constructed one by one.

Furthermore, even the existing techniques run into trouble as the number of dimensions increase. For one thing, the tags get very large and warehouse access time suffers. Also, many variables are in fact independent of some of the dimensions available. If no account is taken of this, the warehouse fills with duplicated data - the same data object stored under many different tags.

These considerations strongly suggest some kind of compile time dimensional analysis. In principle, this is not a problem - it is a simple example of "abstract interpretation". The practicality is another matter. When there is only one dimension, it amounts to checking for constancy, which corresponds closely to strictness analysis (already a hard problem). It gets drastically worse if there are four or five dimensions. And what of universes with infinitely many dimensions? And dimension shifting operations that can't be analysed statically?

One scheme is to compute dimensionality at run time: it is easy, during education, to record the dimensions whose values were actually needed, so that when we put a value in the warehouse we know exactly those tag fields that were really needed to determine that particular values. But the scheme breaks down when we have to find a value stored in the warehouse: we have no idea which fields were used in the tag.

Recently Faustini and I have devised a solution to the problem, involving a multi-step warehouse access algorithm. It's based on using applying education to the contexts themselves, so that the parameters of the various dimensions are accessed on a demand-driven basis.

Suppose then that we want the value of  $V$  in the current context. The first step is to assume that this value is independent of the context ( $V$  is a constant). We look up  $V$  with an empty tag.

Possibly, we will find a value (extension) associated with  $V$  with and the empty tag. If so, we are done. If not, what we will find instead is an indication of which dimension must be examined.

For example, it might indicate that we need to know the time parameter. We demand and get the time parameter, (say it is 18). We then look  $V$  up again with a tag with 18 in its only field. If we find a value, we are done - in the given context  $V$  depends only on time. And if we don't find a value, we are told which dimension to ask about next. So perhaps we find out that in addition to the time we also need the  $y$  space coordinate. We request this value and again look up  $V$ , but this time with a two-field tag. Again, we get either a value or a dimension we have to ask about.

I was assuming that the dimensional information is already present in the warehouse. At first, it won't be, but it can be computed from the definitions. For example, if  $V$ 's defining expression is of the form  $A$  fby  $B$ , we can begin by asking about the time dimension.

Obviously a lot of details need to be filled in; but it seems to work (the idea is new enough we haven't finished working it out let alone testing it). Some of the advantages should be clear. In particular, there is no overhead on having lots of dimensions, you pay as you go for only those dimensions you actually use.

What we have is a uniform implementation of eduction when the worlds are determined by a coordinate system: and indexed family of parameters (space, time, place, choice etc). In other words, IP in which the intensions are elements of  $((F \rightarrow Z) \rightarrow D)$  with  $F$  the set of coordinate indices,  $Z$  the integers (though the coordinates don't have to be integers) and  $D$  the domain of extensions. This covers almost every IP proposal suggested so far (including hyper time, nested space, branching time, and the choice dimensions). It clearly needs work, but perhaps by ISLIP 90 we will all be able to repair to our own universes!

#### BY POPULAR DEMAND

I am sure some readers will find this paper somewhat frustrating. I have presented a number of interesting, but always with not quite enough with not quite enough details to able to decide if they really work. Space is part of the problem: each idea could expand to a full paper if it were treated in depth. Perhaps I could give details on demand, during the presentation. But you have to be there.

Furthermore, these details aren't always available. They represent work in progress - and probably we will be further ahead after ISLIP90 than before. But I want to give you all an idea of the real potential of our paradigm, which is far richer than Lucid or any single language. Let's get to work!