Intensional Intentional Programming

Bill Wadge

Abstract

We discuss the use of intensional (context-switching) operators—Intensional Programming—in the development of families of versions of programs and other forms of software—Intentional Programming.

1 One Letter Apart

It is hardly unusual for two separate research groups to independently develop similar (but not identical) ideas. It is unusual, however, for the groups to coin similar, but not identical, names for these ideas.

This seems to have happened in the case of “Intensional Programs” and “Intentional Programming”. In this paper we will suggest some very promising ways in which the first can be applied to the second.

We should all be familiar with “Intensional Programming”—briefly, it means programming in a language/system based on intensional (e.g., modal or temporal) logic and implemented with some kind of demand-driven dataflow.

So what is “Intentional Programming”?

This is the name chosen for a project underway at Microsoft’s research division.

Roughly speaking the MS project [1] is based on the fact that the development of any software system or component invariably involves a large number of arbitrary decisions: how big the stack should be, how many levels (if any) of undo are allowed, whether the button bars should be artistically bevelled, whether bignums are supported, and so on.

The idea of Intentional Programming is to allow the developers to postpone these decisions—in fact, to avoid making them. The IP approach is to develop a whole indexed/parameterized family of components, each member of which is determined by the settings of the parameters—that is, by a particular set of design decisions. They call these families “Intentions”.

The Microsoft project is more ambitious than just described, but let us adopt the term “Intentional Programming” to refer to any approach in which programs or other kinds of software are developed as families of versions (variants) rather than as single instances.

Clearly, an MS “Intention” is a special case of what logicians since Carnap have referred to as an “intension”—a function which assigns an extension to each possible world (index, context) in some index space. In the special case, a possible world is a complete set of design decisions—a complete specification of a single version of the software unit. The extension at that world is the resulting concrete system of component with, say bevelled buttons, seven levels of undo, 32-bit integers only, and so on.

2 The Intensional approach to Intentional Programming

Intentional Programming in this sense corresponds very closely to the intensional approach to version control developed by the author and John Plaice [3].

*Presented to the 9th International Symposium on Languages for Intensional Programming, Arizona State University, Tempe, Arizona, USA, 13–15 May 1996.
This approach is based on a partially ordered version space used to index the different versions of software and software components. The ordering represents version refinement: If version \( w \) is a refinement of version \( v \), then version \( v \) of a component can also serve as version \( w \) if no version closer to \( w \) is available. In other words, version \( w \) inherits from version \( v \).

We presented Lemur, a simple module facility/tool for C which uses the ordered-space approach to support the development of multiple versions of modular software with sharing of source and object code between versions. Briefly, Lemur allows the developer to break a project down into a collection of interdependent modules and furthermore allows the components of modules to exist in separate versions. When the developer requests a particular version of the whole system, Lemur assembles it using in each case the most relevant version of the individual components.

Lemur modules have import lists (which can also be versioned). For example, the import list of module \( A \) might consist of \( B, C \) and \( D \). If there is only one such list, it means that any version of \( A \) requires the corresponding versions of \( B, C \) and \( D \). For example, it means that the french version of \( A \) requires the french versions of \( B, C \) and \( D \) (some of which, of course, might be the same as the standard versions).

The semantics of the Lemur import lists corresponds to the Lucid pointwise interpretation of addition, and can be written semi-formally as

\[ A \% v = B \% v + C \% v + D \% v \]

where \% is the Lemur (sub)versioning operator. In other words, the software component assembly operation is interpreted by Lemur as a pointwise operation of the space of Intentions.

### 3 The External Component Problem

So much for Intensional vs Intentional programming—what is Intensional Intensional programming? Briefly, it is IntenTional programming with ItenSional operators.

This idea seems natural enough from the point of view of theory but in fact this (in retrospect) obvious extension to the Lemur approach grew out of an attempt to remedy practical limitations of the approach.

In the Lemur approach, the versions have a uniform meaning—the french version of the dictionary module is meant to work with the french version of the spell-checking module. However, most existing software version systems have separate version spaces for each component. In these systems there is no a priori reason to expect version 5.3 of the windowing system to be compatible with version 5.3 of the storage management module. The configuration tools allow (require) the programmer to explicitly define a choice from each private version space to form a version of an application.

The Lemur user is spared this burden if all the development is done in Lemur—but what if an application requires an ‘external’ component with a necessarily different version space?

The problem was brought to our attention by Bjorn Freeman-Benson in 1995 but the solution is due to Gord Brown, who as a student implemented the Lemur system for intensional version management. (I am very pleased to report that Gord is currently my MSc student here at UVic.)

Gord Brown suggested (back in 1989!) that the modules on the import list be versioned; for example, \( A, B \) and \( D \% \text{graphical} \). We did not pursue it at the time because the external component problem had not arisen. Furthermore we were not clear what this should mean in general. If version \( \gamma \) of \( B \) appears on the import list of \( A \), this means that the standard version of \( A \) requires the \( \gamma \) version of \( B \). But what does the \( \delta \) version of \( A \) require? Still the \( \gamma \) version of \( B \)? Sometimes, if \( B \) is an external module and \( \gamma \) is in a separate version space. But sometimes not—if you want the french version of \( A \), should it not import the french versions of all the modules it needs?
4 Operators on Intensions

The puzzles are easily resolved once we formulate the problem in terms of intensional operators over the space of intentions. In fact, Gord Brown's suggestion has two possible (and useful) interpretations, corresponding to two simple (families of) intensional operators.

Let $\gamma$ be fixed. We can define an operator $G$ on the space of module intensions as follows: version $\delta$ of $G(C)$ is version $\gamma$ of $C$. This corresponds to the first operator of Lucid. Semi-formally,

$$G(C)\%\delta = C\%\gamma$$

However, we can also use $\gamma$ to define another operator $R$, which uses subversioning (refinement): so that version $\delta$ of $R(C)$ is subversion $\gamma$ of version $\delta$. The equation is

$$R(C)\%\delta = C\%\gamma\%\delta$$

$R$ corresponds more to the next operator of Lucid.

Both kinds of what we call trans-version linking could be useful. For example, module $A$ might require a graphical version of the interface module $B$, with captions in the appropriate language. So that, for example, the french version of $A$ uses the french version of the graphical version of $B$ ($B\%$graphical$french$).

On the other hand, $A$ may require a particular version of the third-party editor module $C$. All versions of $A$ would use the same version of $C$.

Our semantic analysis therefore suggests two kinds of versioned imports, and no reason not to allow the user to mix them. In fact, they correspond closely to the relative and absolute links allowed in spreadsheet formulas. On the practical level, we need only devise some syntactic convention for the import lists to indicate relative versus absolute subversioning of imports. For example, lines with absolutely versioned imports could be terminated with a period. One possible import list might then be

```
stack
interface%textual
word%5.1.
```

5 Mixing and Matching Versions

Once we have absolute import versioning, and the ability (already provided by Lemur) to create different versions of the import list, we can imitate the existing practice of configuring versions of an application by combining (in a potentially arbitrary fashion) various versions of various components.

Suppose, for example, that we want version 12 of $A$ to use versions 6, 8 and 2.3 of modules $B$, $C$ and $D$, respectively. We create version 12 of its import list and enter the lines

$$A\%6.$$

$$B\%8.$$

$$C\%2.3.$$ 

If we do this for every version of $A$, we have essentially disabled Lemur's intensional approach and converted it to a conventional tool. This is exactly analogous to writing Prolog using only facts and never rules, or writing Lucid equations like

```plaintext
first X = Y atime 2
first next X = Y atime 8
first next next X = Y atime 1
```
We clearly do not recommend this style. Instead, we urge software designers to use common version spaces for the components and write generic import lists. However, it may be useful to be able to mix and match in special cases, for example when using families of components developed under the old methodology with private version spaces. The added flexibility should ease the transition from the older versioning approach to the intensional approach.

6 Versioning the Web

Intensional version operators have proved useful (in fact, indispensable) in another context, namely the multiversion Web site project currently pursued by the author and Taner Yıldırım, an MSc student at Victoria.

The idea is to adapt the intensional approach to versioning to the production of Web sites, to allow sites with many different versions but with sharing of source.

For example, we want Websters to be able to create sites with versions in several different languages. But we do not want to force them to make clones of all the pages in the site. For example, if a page has no text at all, we would like to have only one copy of the source. This is not possible if the site has links to pages with text; then we need versions of the page which look the same on the screen but differ in that they link to different versions of the following text pages.

The idea is to allow authors to use a high-level intensional HTML in which the links are generic—much as Lemur’s imports are generic. In other words, if a page (say MAP) has a link to another page (say DIRECTIONS) then the understanding is that any particular version (say french) of the MAP page is linked to the corresponding version of the DIRECTIONS page.

This allow versions to share: if there is only one source for the MAP page, labeled as the vanilla version, then it serves (by version inheritance) as the source for all the other versions of the MAP page. However, authors will provide several versions of the DIRECTIONS page, written in and labeled by the appropriate language.

There is one obvious limitation to this scheme: how do viewers of a site choose a version, or move from one to the other? Without cumbersome forms, pages or changes to the HTTP protocol to allow state?

The answer is again trans-version links, based on intensional operators over the space of page intensions.

These links require a slightly more elaborate version space/algebra. Following the suggestion of Plaice and Ben Lamine [2], we make dimensions explicit. Thus a typical version might be

language: english + display: text + background: color: pink

Once we have explicit dimensions, we extend our intensional HTML with links in which changes are specified to particular dimensions of the current version. For example, authors can write in the MAP page

\[ \text{<a href=DIRECTIONS language=english>...} \]

This means that version \( v \) of this page links to a version \( v' \) of the DIRECTIONS page, where \( v' \) differs from \( v \) at most in that the language component is \textit{english}. In other words, it links a MAP page to the corresponding version of \( E(\text{DIRECTIONS}) \), where the intensional operator \( E \) is defined by

\[ E(P)\%v = P\%v\text{[language \text{english}\rightarrow english]} \]

The net result is that authors can create, using a relatively small number of files, a site which to the viewer seems to exist in many (perhaps hundreds) of different versions which coexist as kinds of parallel universes. Furthermore, the author can provide at strategic points links between the different versions of the site—links which can have a dramatic effect on the appearance of the pages but are nevertheless standard point-and-click hyperlinks.

We will report in a future work on the many possibilities offered by trans-version linking, and on the techniques needed to make the sites accessible to existing browsers. In the meantime, we invite you to visit Taner Yıldırım’s prototype multiversion home page [4].
Acknowledgements

We would like to thank Taner Yildirim, John Plaice, Gord Brown, Monica Schraefel and Bjorn Freeman-Benson for many discussions/objections/suggestions. The work was supported by a grant from the Canadian Natural Sciences and Engineering Research Council (NSERC).

References


