Intensional objects*

Peter Kropf John Plaice

May 17, 1999

Abstract

We summarize the interaction between the theory behind intensional programming, as seen in Lucid; and intensional versioning, as seen in Lemur, Intensional HTML, ISE, VMAKE, etc. These two concepts can be seen to be duals of each other, and they rely on dual notions of store, the *warehouse* for caching values, and the *catalog* for providing definitions. Catalogs contain *intensional objects*, which are openable boxes labelled by Lucid contexts. Finally, Lucid contexts are now first-class values.

1 Introduction

It has always been understood that there is some real relationship between intensional programming and intensional versioning. Clearly the two rely on possible-world semantics as the underlying basis, but are they the same, or are they different?

At one level, they appear to be the same: each uses a warehouse to undertake its task. In intensional programming, an expression is evaluated in the current context, and the current values of variables are looked up in a warehouse; should a value not be there, then a computation thread is initiated to compute that value, and then to place it in the warehouse before returning it.

In intensional versioning, the task is to build a system from versioned components, which are already sitting in the warehouse. When a variable is encountered, a search is made of the warehouse for those expressions that are relative to the current context. The best fit is chosen and returned.

It should be clear that these two warehouses play quite different roles. In the first case, the warehouse serves as *cache*, and exists primarily to improve performance. Within reason, removing the warehouse does not affect the outcome of a computation, only how long it takes. However, one might argue that the warehouse increases the accuracy of computations, because one source of errors, the computation itself is eliminated when the

*Presented to the *International Symposium on Languages for Intensional Programming*, Demokritos Institute, Athens, Greece, 28–30 June 1999. Current addresses: Peter Kropf, Département d'informatique, Université Laval, Québec (Québec) Canada G1K 7P4. John Plaice, School of Computer Science and Engineering, The University of New South Wales, Sydney 2052 Australia. Email: `kropf@ift.ulaval.ca`, `plaice@cse.unsw.edu.au`.

value already exists in the warehouse, leaving the memory access as the only source of error persisting.

On the other hand, the warehouse in intensional versioning contains the source of everything. If something is removed, then the semantics of the system changes. Bill Wadge has suggested that the word *catalog* may be more appropriate, since if we have an empty warehouse, we do not worry, except in the sense that delivery might take too long, but if we have an empty catalog, then everyone will worry, because nothing can be ordered.

If we retain the word *catalog*, we must ask ourselves: What does a catalog contain? If, upon encountering a variable, a search through the catalog is undertaken by the version system for the component that best matches the current context, then we can see our catalog as a bunch of boxes, each with some contents and a tag.

Where there are tags, there are contexts, and here is the key to the rest of the discussion. The tags on these boxes are contexts that have been stored. In other words, at some point, a current context was held as a value. Contexts must themselves be values.

This approach is consistent with the IHTML work undertaken by Gord Brown [4]. The `vmod` operation allows relative contextual changes, akin to the @operator in Lucid. However, the `version` operation allows one to completely replace the current context by another one.

We call these labeled boxes *intensional objects*. They will have the advantage, unlike typical objects, of being openable, and repackageable, with modified contents or updated labels.

The paper is structured as follows. We begin with yet another flavor of Lucid, which we call Contextual Lucid. Its syntax and semantics are followed by a discussion of the utility of Intensional Objects in a distributed environment. The full paper will contain a much richer discussion, and possibly a better understanding about how the internal states of catalogs and warehouses are changed.

2 Syntax and semantics of Contextual Lucid

Below is an intuitive explanation of the semantics. The formal semantics can be found in Figure 2 (page 2). We would like to point out that we are not currently satisfied with the definitions of our boxes, as we seem to be packaging extensions, rather than intensions. We hope to have these issues resolved for the symposium.

- *id* :
  The starting point. The subject of the LOOKUP and EVAL functions.
- `const` c :
  Constant in an algebra.
- `op` f :
  Operation in an algebra.
- `fn` $id_1, \ldots, id_n \Rightarrow E$:
  Definition of a function.

$$
\begin{array}{rcl}
E & ::= & \mathit{id} \\
& | & \texttt{const}\ c \\
& | & \texttt{op}\ f \\
& | & \texttt{fn}\ \mathit{id}_1, \ldots, \mathit{id}_n\ \Rightarrow\ E \\
& | & E(E_1, \ldots, E_n) \\
& | & \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' \\
& | & \texttt{dim}\ E \\
& | & \texttt{undim}\ E \\
& | & \texttt{box}(E, E') \\
& | & \texttt{gettag}\ E \\
& | & \texttt{getval}\ E \\
& | & !! \\
& | & \langle\rangle \\
& | & \#\#(E, E') \\
& | & E\ @E'\ E'' \\
& | & E\ @@\ E'
\end{array}
$$

Figure 1: Abstract syntax for Contextual Lucid

- $E(E_1, \ldots, E_n)$:
  Applying an operator or a function to its operands. Operators are strict in their operands.
- `if` E `then` E' `else` E'' :
  The conditional expression; nothing special.
- `dim` E :
  Definition of a dimension; E must evaluate to a constant.
- `undim` E :
  The value identifying a dimension; E must evaluate to a dimension. Dual to `dim`.
- `box`(E, E') :
  Create a tagged object. Expression E must evaluate to a context.
- `gettag` E :
  Get the tag from a tagged object.
- `getval` E :
  Get the value from a tagged object.
- !! :
  The current context.
- $\langle\rangle$:
  The empty context.

$$\mathbf{E_{id}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash \mathcal{D}(id) : v}{\mathcal{D},\mathcal{P} \vdash id : v}$$

$$\mathbf{E_{op}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{op}\ f \qquad \mathcal{D},\mathcal{P} \vdash E_i : \texttt{const}\ c_i}{\mathcal{D},\mathcal{P} \vdash E(E_1,\ldots,E_n) : \texttt{const}\ f(c_1,\ldots,c_n)}$$

$$\mathbf{E_{fn}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{fn}\ id_1,\ldots,id_n \Rightarrow E' \qquad \mathcal{D},\mathcal{P} \vdash E'[id_i \leftarrow E_i] : v}{\mathcal{D},\mathcal{P} \vdash E(E_1,\ldots,E_n) : v}$$

$$\mathbf{E_{c_T}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{const}\ \mathit{true} \qquad \mathcal{D},\mathcal{P} \vdash E' : v'}{\mathcal{D},\mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v'}$$

$$\mathbf{E_{c_F}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{const}\ \mathit{false} \qquad \mathcal{D},\mathcal{P} \vdash E'' : v''}{\mathcal{D},\mathcal{P} \vdash \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' : v''}$$

$$\mathbf{E_{dim}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{const}\ c}{\mathcal{D},\mathcal{P} \vdash \texttt{dim}\ E : \texttt{dim}\ c}$$

$$\mathbf{E_{undim}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{dim}\ c}{\mathcal{D},\mathcal{P} \vdash \texttt{undim}\ E : \texttt{const}\ c}$$

$$\mathbf{E_{empty}} \;:\; \frac{}{\mathcal{D},\mathcal{P} \vdash \langle\rangle : \langle\rangle}$$

$$\mathbf{E_{current}} \;:\; \frac{}{\mathcal{D},\mathcal{P} \vdash !! : \mathcal{P}}$$

$$\mathbf{E_{box}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : v \qquad \mathcal{D},\mathcal{P} \vdash E' : \mathcal{P}'}{\mathcal{D},\mathcal{P} \vdash \texttt{box}(E,E') : \texttt{box}(v,\mathcal{P}')}$$

$$\mathbf{E_{getval}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{box}(v,\mathcal{P}')}{\mathcal{D},\mathcal{P} \vdash \texttt{getval}(E) : v}$$

$$\mathbf{E_{gettag}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{box}(v,\mathcal{P}')}{\mathcal{D},\mathcal{P} \vdash \texttt{gettag}(E) : \mathcal{P}'}$$

$$\mathbf{E_{index}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E : \texttt{dim}\ c \qquad \mathcal{D},\mathcal{P} \vdash E' : \mathcal{P}'}{\mathcal{D},\mathcal{P} \vdash \#\#(E,E') : \mathcal{P}'(c)}$$

$$\mathbf{E_{rel}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E' : \texttt{dim}\ c' \qquad \mathcal{D},\mathcal{P} \vdash E'' : \texttt{const}\ c'' \qquad \mathcal{D},\mathcal{P}\dagger[c' \mapsto c''] \vdash E : v}{\mathcal{D},\mathcal{P} \vdash E\ @E'\ E'' : v}$$

$$\mathbf{E_{abs}} \;:\; \frac{\mathcal{D},\mathcal{P} \vdash E' : \mathcal{P}' \qquad \mathcal{D},\mathcal{P}' \vdash E : v}{\mathcal{D},\mathcal{P} \vdash E\ @@\ E' : v}$$

Figure 2: Semantic rules for newest Lucid

- $E\#\#E'$:
  Querying a context. Expression E must evaluate to a dimension, while expression E' must evaluate to a tag.

- $E\ @E'\ E''$:
  Lucid's change of context, Gord Brown's VMOD operation.

- $E\ @@\ E'$:
  Gord Brown's VERSION operation. Expression E' must evaluate to a tag.

Our first-class values are therefore constants, operations, functions, dimensions, tags, and objects.

3 Object-oriented and networked programming

Current object-oriented programming only allows communication to take place between pairs of objects directly, or indirectly, through a broker, such as in Sun's recently announced Jini technology [7]. In the words of John Gage, Chief Scientific Officer at Sun, Jini should allow the creation of communities of components. Jini defines communities as *federations* which are sets of nodes that may interact and collaborate. These nodes offer different services, each relying on its own specific service protocol. However, communities are much more than the side-by-side placement of isolated components sitting in empty space. Communities are dynamically and autonomously created by defining and redefining the relationships between their components as well as their environment and context. in which they emerge and exist. The Web Operating System (WOS)[1, 6] approach for global computing is an example relying on this novel concept of dynamically defined communities or *versions* [2, 9]. The associated warehouses or catalogs typically contain objects providing information about a node's resources. A community of nodes acting as a parallel computer may now be defined by searching the node's warehouses for the resources necessary to define the virtual parallel computer. This will thus define a new context of computation.

In the domain of electronic commerce one would want to dynamically create and manage open marketplaces. A marketplace involving a large number different participants with diverse interests and needs must constantly adapt to the evolving market and to the behavior of the participants. New participants might join the marketplace as providers of goods (with their catalogs) or as consumers (with their specific conditions and needs) and participate in all kinds of business processes (bids, sell and buy, etc.). For such a system to be functional, it is necessary to manage dynamic changes rather than just offering a fixed, by definition restricted set of functions. Again, the participants of such an open marketplace will dynamically form communities and new contexts.

Intensional objects

In general, the **object model** includes the following principles: abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. Since none of those principles

are new, the strength of the object model lies within the synergetic effects when they are brought together. The object model is commonly applied at all three levels in IT: analysis, design, and programming. Object-oriented programming may be defined as follows [3]:

> Object oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

An object is an entity which exists in time and space whereas a class represents a set of objects that share a common structure and a common behaviour. A class is thus an abstraction of the object. An object has a *state*, *behavior*, and *identity*. Objects alone are uninteresting. Therefore, relationships between objects are defined. The objects contribute thus to the system's behavior by collaboration. Given these main features of object-oriented systems, the following key principle of the object-orientation can be deduced: In order that a system based on the above properties to be functional in a well defined manner, the internal behavior of objects must be hidden from the invoker of an object. This means that only the interfaces to access the functionality of an object will be known outside of the object. The implementations themselves are thus always hidden as much as possible.

Widely used concepts for distributed object-oriented programming such as CORBA [8] and Sun's RMI (Remote Method Invocation) allow one to get an object and dynamically look at its properties. However, they do not allow them to be adapted to the new environment into which they have been included. Java Beans allow some sort of adaptation in that they provide object templates, whose specific implementation can be locally defined.

From this discussion it follows that object-orientation assumes a fixed, known environment in which the objects are used. This becomes especially undesirable in the context of distributed objects, i.e. in networked environments where dynamic changes frequently occur. In this case, one would want to have the possibility to dynamically adapt the behavior of an object to fit the new environment without a completely new object being redefined.

It is precisely this capability of creating new contexts and dynamically adapting objects to environments which *intensional objects* address. This means that the principle of hiding every implementation is weakened: instead of only providing interfaces and completely hiding implementations, intensional objects allow one to inspect or even alter implementations. The interfaces therefore become transparent. Instead of hiding implementations, intensional objects rather manage implementations in a well defined manner. The Globus [5] approach for large-scale scientific metacomputing proposes *translucent* interfaces for objects. These interfaces allow one to inspect properties of an object through interface parameters and subsequently select the desired internal properties and behavior of an object. These translucent objects are used in Globus to account for specific properties of the underlying properties of a parallel machine and environment. Intensional objects go a step further than this approach. They allow, depending on the changing context, for the inspection and adaptation of the internal behavior of any object, based on the syntax and semantics defined in the previous section. With intensional objects, because of the ability to open the boxes, and to repackage them at will, we should be able to facilitate networked computing in a way that was never possible with traditional object-oriented techniques. Moreover,

intensional objects still fit into the general definition of object-oriented programming as given above, because the well-defined behavior of intensional objects as defined in the previous section guarantees that a system based on the properties of intensional objects will be functional in a well-defined manner.

4 Conclusion

As presented above, object-oriented programming consists basically in a collection of different known concepts and principles which are combined to create and offer synergetic effects. With the introduction of intensional objects, we complete the object-oriented model of programming for a consistent application in dynamic network environments. Each instance of the environment is thus a version or one specific possible-world semantics.

We would like to point out that we are participating in a debate that goes back all the way to ancient Greece, to Demokritos's atoms and Aristotle's aether. In intensional programming, we say that the aether exists, and that the atoms flow through and are submerged in the aether, which flows through to the finest subunits, whatever the level that they might occur. In traditional intensional programming, we could put our fingers up in the air to get the temperature, now we can also put out fingers on the atoms flowing around us in the aether, and check their temperature.

When many atoms flow around, their combined interactions can change the current context, and even create new contexts. That is the purpose of the Intensional Objects.

References

[1] S. Ben Lamine, P.G. Kropf, and J. Plaice. Problems of Computing on the Web. In A. Tentner, editor, *High Performance Computing Symposium 97*, pages 296 – 301, Atlanta, GA, April 1997. The Society of Computer Simulation International.

[2] S. Ben Lamine and J. Plaice. Simultaneous multiple versions — the key to the WOS. In *Distributed Computing on the Web (DCW'98)*, pages 122–128, Rostock, Germany, June 1998.

[3] G. Booch. *Object-oriented Analysis and Design.* Addison-Wesley, Menlo Park, CA, 1994.

[4] G. Brown. Intensional HTML 2: A practical approach. Master's thesis, University of Victoria, B.C., Canada, 1998.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications,* 1997.

[6] Peter Kropf. Overview of the WOS project. In *1999 Advanced Simulation Technologies Conference (ASTC 1999)*, San Diego, CA, USA, April 1999.

[7] Sun Microsystems. Jini. http://java.sun.com/products/jini/whitepapers/.

[8] T.J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons, New York, NY, USA, 1995.

[9] J. Plaice and P. Kropf. Wos communities – interactions and relations between entities in distributed systems. In *Distributed Computing on the Web (DCW'99)*, Rostock, Germany, June 1999. to appear.