XII

Formal Aspects of Object-Oriented Systems

M. Wolczko

Semantics of Object-Oriented Languages

Mario Wolczko

Dept. of Computer Science
Manchester University

mario@uk.ac.man.cs.ux

What is object-oriented programming?

Is it a language feature or a methodology?

What are the key concepts? Inheritance (what sort)? Dynamic binding? Persistence?

How does it differ from programming with ADTs?

XII.1

Aims of Investigation

- Comparative study
- Clarification of concepts
- Classification of essential / important / other features.

Hoped-for results

- Deeper understanding of object-oriented programming
- Useful suggestions for language designers

Approach

Denotational definition(s) of Smalltalk-like languages, using VDM

Excluded: "unusual" systems (Actors), concurrency

What's an Object?

More usefully, What properties do objects have?

- Semantic, rather than syntactic concept
- Each object has its own identity, distinct from its contents
- Has a private "inside" — only the inside of one object is in scope at any place in the program

Principal Semantic Domain

Store or object memory:

$\mathit{Object\_memory} = \textbf{map}\ \mathit{Oop}\ \textbf{to}\ \mathit{Object}$

Objects have internal structure, e.g.:

$\mathit{Object} = \textbf{map}\ \mathit{InstVarID}\ \textbf{to}\ \mathit{Oop}$

Contrast with conventional store:

$\mathit{Env} = \textbf{map}\ \mathit{Id}\ \textbf{to}\ \mathit{Loc}$

$\mathit{Store} = \textbf{map}\ \mathit{Loc}\ \textbf{to}\ \mathit{Value}$

Is this difference significant?

Language Development (1)

Simplest store

$Store = \textbf{map}\ Id\ \textbf{to}\ Val$

$MStat = Stat \rightarrow (Store \rightarrow Store)$

All identifiers are global — no abstraction mechanism.

Language Development (2)
Nesting (via blocks), and procedures

```
var x, y, ...

begin
  var x, ...
end
```

No change to underlying store model required, even if we add call-by-value procedures, but need an *environment* to describe which identifiers are in scope.

Only way to create multiple "instances" of a block is to "name" them (i.e., make them procedures) and use recursion — lifetimes are LIFO.

Language Development (3)

Aliasing, Shared variables, call-by-reference

```
var x,...

proc p(var y)   ...x...y

p(x);
```

Need to introduce *locations.*

$\mathit{Store} = \textbf{map } \mathit{Loc} \textbf{ to } \mathit{Val}$

$\mathit{Env} = \textbf{map } \mathit{Id} \textbf{ to } \mathit{Loc}$

$\mathit{MStat}\colon \mathit{Stat} \to (\mathit{Env} \to (\mathit{Store} \to \mathit{Store}))$

Language Development (4)

Heaps, Pointers, Modules, ...

Still have global variables and LIFO allocation, therefore introduce a *heap* and *pointers.*

$\mathit{Val} = \mathit{Loc} \cup \ldots$

Still not *modular*, so add modules and packages (at the syntactic level).

Is there a simpler way?

The Object-Oriented Way

Objects are similar to the simple store:

Object = **map** *Id* **to** *Oop*

No shared variables and aliasing allowed.

Make each object an independent entity (created on demand, destroyed when inaccessible).

Each object has an identity (*Oop*) that distinguishes it from others.

All objects are in a one-level store:

Object_memory = **map** *Oop* **to** *Object*

Objects as Environments

Each *method* is a store-transformer:

MMethod: *Args* × *Object_memory* →
Result × *Object_memory*

Within a method only the inside of one object is in scope:

MStat: *Oop* × *Object_memory* →
× *Object_memory*

MStat(*rcvr*, *mem*) can only alter *mem*(*rcvr*) (without sending a message).

Benefits

- Modularity
- Separation of control and data
- Data abstraction
- Security
- Simplicity

Conjecture

Objects are the *only* essential feature of an object-oriented language

Any object-oriented language will have an underlying structure like *Object_memory*.

Pure object-oriented languages have no other storage structure.

By simulating the object memory we can practice OOP in almost any language.

Principle of Object Identity

Every object has a unique identity, which cannot change without the object's cooperation

Principle of Object Encapsulation

The internal state of an object can only be accessed or modified by the execution of a method associated with the object, in response to a message sent to that object.

Important Features

Dynamic binding

If you can't see the internal state of other objects, why not let different sorts of objects be used for similar purposes?

Object = **map** *Id* **to** *Oop*

```
    var x;
    x.print();
```

Leads to a form of polymorphism / overloading / generic functions.

Persistence and Incrementality

Once we have dynamic binding, long-lived (persistent) data makes sense — its behaviour can alter as required. Need not anticipate all operations on an object when defining it.

Classes

An implementation of a behaviour defines a *class* of objects.

Inheritance

One class can inherit part of another's implementation. Dynamic binding (via *self*) allows incomplete or *abstract* classes.

Classes can define syntactic modules.

Delegation

Dynamic inheritance