Classes
=
Objects + Data Abstraction

John Mitchell
Stanford University

Joint work with K. Fisher

Outline

- Object-oriented programming
- History of object calculus
- Classes and data abstraction

Object-Oriented Programming

- An object consists of
  - » hidden data
  - » public operations

Hidden data	
msg 1	method 1
...	...
msg n	method n

- Program sends messages to objects

What's interesting about this?

- Universal *encapsulation* construct
  - » Data structure
  - » File system
  - » Database
  - » Window
  - » Integer
- Metaphor usefully ambiguous
  - » sequential or concurrent computation
  - » distributed, sync. or async. communication

Languages

- Simula 1960's
  - » Object concept used in simulation
  - » Activation record; no encapsulation
- Smalltalk 1970's
  - » Improved metaphor; wholly object-oriented
- C++ 1980's
  - » Adapted Simula ideas to C
  - » Many special-purpose considerations

Language concepts

- encapsulation
- "dynamic lookup"
  - » different code for different object
  - » integer "+" different from real "+"
- subtyping
- inheritance

Abstract data types

```
Abstype q
   with mk_Queue : unit -> q
        is_empty : q -> bool
        insert   : q * elem -> q
        remove   : q -> elem
     is {q = elem list, ( ... tuple of functions ... ) }
     in
         ...  program  ...
    end
```

Block-structured simplification of modular organization

Abstract data types

```
Abstype q
   with mk_Queue : unit -> q
        is_empty : q -> bool
        insert   : q * elem -> q
        remove   : q -> elem
     is {q = elem list, ( ... tuple of functions ... ) }
     in
         ...  program  ...
    end
```

q's are abstract

q's treated as lists of elems

Subtyping

- *A* is a subtype of *B* if
  any expression of type *A* is allowed in every context requiring an expression of type *B*
- Substitution principle
- Provides extensibility
- Property of types, not implementations

Object Interfaces

- Type

```
Counter  =
      ⟨⟨ val:int, inc:int -> Counter ⟩⟩
```

- Subtyping

```
RCounter  =  ⟨⟨ val:int, reset:RCounter,
                 inc:int -> RCounter ⟩⟩

            <:  Counter
```

Inheritance

- Mechanism for reusing implementation
  - » `RCounter` from `Counter` by extension
  - » `Counter` from `RCounter` by hiding
- In principle, not linked to subtyping
- *Puzzle:*
  Why are subtyping and inheritance combined in C++, Eiffel, Trellis/Owl ...?

Analysis

- Study properties of object-oriented languages using simplified models

$$\frac{\text{Lambda calculus}}{\text{Programming languages}} = \frac{\text{Turing machines}}{\text{Real hardware}}$$

Object Calculus

- extend untyped lambda calculus

`⟨⟩`	empty object
`e <= m`	send message to object
`⟨e <-+ m=e′⟩`	add method to object
`⟨e <- m=e′⟩`	replace method to object

- symbolic evaluation rules

```
⟨e <- m=e′⟩<= m    -->     e′⟨e <- m=e′⟩
```

- static type system
  - » prevent *"message not understood"*

Examples

- Non-termination

```
⟨m = λself.self <= m⟩ <= m
: pro u. ⟨⟨m:u⟩⟩
```

- Counter

```
⟨val = λself.3,
 inc = λself.self <- val=λs.(self<=val +1)⟩
: pro u. ⟨⟨val:int, inc:u⟩⟩
```

Numerals as Pure Objects

- Represent number n by object

```
n = ⟨rep = λself. λx. x<=m<= ... <=m,
     suc = λself.⟨self <-
           rep = λslf.λx.(self<=rep x)<=m ⟩,
     pre = λself.⟨self <- rep = ... ⟩ ⟩
   : pro u.⟨⟨suc:u, pre:u, rep: ..-> .. ⟩⟩
```

- Fixed-point operator also definable, typed

Typing Rule for Messages

```
          e : pro v.R
     v:{v+} |-  R <: ⟨⟨m:τ⟩⟩
  ------------------------------
     e <= m  :  [pro v.R/v] τ
```

Typing Rule for Extension

```
                    e : pro u.R
             u:{u+} |-  R :: ⟨{n},V⟩
  r <: λu.⟨⟨R|n:τ⟩⟩ |- e′ :               .
                       [(pro u.ru)/u](u->τ)
  ----------------------------------------------
        ⟨e <-+ n=e′⟩ : pro u.⟨⟨R|n:τ⟩⟩
```

Problem with Subtyping

- Assume

```
ob : pro v.⟨⟨ f:int->int, g:int->int ⟩⟩
```

- Suppose we have subtyping

```
pro v.⟨⟨ f:.., g:..⟩⟩  <: pro v.⟨⟨ f:.. ⟩⟩
```

- The we can form extended object

```
⟨ob <-+ g : bool ⟩
```

- But if `f` refers to `g`, this causes an error.

Delegation and Subtyping

- Prototypes `pro v.⟨⟨ m:... ⟩⟩`
  - » Can add, replace methods
  - » No subtyping between `pro` types
- Objects `obj v.⟨⟨ m:... ⟩⟩`
  - » Cannot add, replace methods
  - » One method may replace another
  - » Subtyping between `obj` types
- Same representation, different types

Interfaces vs Implementation

- Interface type `obj v.⟨⟨ m:... ⟩⟩`
  - » all objects with these methods, regardless of implementation
  - » useful for library functions, etc.
  - » cannot determine offset statically
  - » cannot implement binary operations

  `set = ⟨⟨mem:el->bool, union:set->set⟩⟩`
- C++ types fix part of implementation

Classes

```
class Point
   private       x : int
   protected setX : int
   public    getX : int
               mv : int -> point
            Point : ...      (constructor)

class ColorPoint : Point
   private       c : color
   protected setC : color
   public    getC : color
       ColorPoint : ...      (constructor)
```

Interfaces

Point_pub = λt. ⟨⟨ `getX:int, mv:int -> t` ⟩⟩

Point_prot = λt. ⟨⟨ `setX:int -> t, getX:..` ⟩⟩

Point_priv = λt. ⟨⟨ `x:.. setX:..getX:.. mv:..` ⟩⟩

CP_pub = λt. ⟨⟨ `pt | getC:color` ⟩⟩

CP_prot = λt. ⟨⟨ `pt | setC:color -> t, getC ..` ⟩⟩

CP_priv = λt. ⟨⟨ `pt | c:.., setC:.., getC:..` ⟩⟩

Point Implementation

```
λiX.⟨    x = λself. iX,
      setX = λself. λnewX.
             ⟨ self <- x = λself. newX ⟩,
      getX = λself. self <= x,
        mv = λself. λdX.
             ⟨self <= setX(dX + self<=getX) ⟩
    ⟩
```

ColorPoint Implementation

```
λiX. λiC. ⟨newPoint(iX)
   <-+    c = λself. iC,
   <-+ setC = λself. λnewC.
              ⟨ self <- c = λself. newC ⟩,
   <-+ getC = λself. self <= c
   ⟩
```

Program Structure

```
Abstype p <: P_prot :: T -> ({c,setC, getC},ø)
   with newPoint : int -> pro v. p(v)
     is {p <: P_prot = P_priv, Impl }
     in
        ...  declaration of cp  ...
      Abstype p <: P_pub :: T -> (ø,ø)
         with newPoint : int -> obj v. p(v)
           is {p <: P_pub = p, newPoint }
           in
              ...  program  ...
```

Properties

- Restricted access to objects
  - » derived class
    - – see protected interface
    - – extend objects, redefine methods
  - » program
    - – see public interface
    - – send messages
    - – subtyping
- Standard properties of data abstraction

Program Structure

```
Abstype p <: P_prot :: T -> ({c,setC,getC},ø)
   with newPoint : int -> pro v. p(v)
     is {p <: P_prot = P_priv, Impl }
     in
        ...  declaration of cp  ...
      Abstype p <: P_pub :: T -> (ø,ø)
         with newPoint : int -> obj v. p(v)
           is {p <: P_pub = p, newPoint }
           in
              ...  program  ...
```

Technical results

- Extend object calculus with
  - » Row variables, variance annotations, negative information about methods, abstract types
- Prove type soundness using
  - » operational semantics
  - » analysis of typing rules
- Use previous work on data abstraction

Insight

- Understand link between inheritance and subtyping
  - » subtypes of abstract type by extension
- C++ `private virtual` problem
- ML: modules, object primitives => OOP

References

- POPL 90: Present basic calculus
- LICS 93: Simplify, type soundness
- TACS/TAPOS: Survey paper comparing
  - » record model
  - » existential types model
- FCT 95: Delegation and subtyping
- Related work by Abadi and Cardelli
- Current paper on `WWW;` K. Fisher thesis