Type Systems and Object-Oriented Programming (IV)

John C. Mitchell
Stanford University

Outline

- ✓ Foundations; type-theoretic framework
- ✓ Principles of object-oriented programming
- ✓ Decomposition of OOP into parts
- Formal models of objects

Goals

- Understand constituents of object-oriented programming
- Possible research opportunities
  - » language design
  - » formal methods
  - » system development, reliability, security

Object-oriented programming

- Programming methodology
  - » organize concepts into objects and classes
  - » build extensible systems
- Language concepts
  - » encapsulate data and functions into objects
  - » subtyping allows extensions of data types
  - » inheritance allows reuse of implementation

Analysis

- Study properties of object-oriented languages using simplified models

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

"Analogy is extremely unfair to λ-calculus" - D. Scott

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 of object

- symbolic evaluation rules

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

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

Examples

- Syntactic Sugar

  write ⟨x = λself.3, y = λself.5⟩

  for ⟨⟨⟨⟩ ←+ x=λself.3⟩ ←+ y=λself.5⟩

- Selection

  ⟨x = λself.3, y = λself.5⟩ <= x

  = (λself.3) ⟨x = λself.3, y = λself.5⟩

  = 3

Examples

- Use of Self
  - » o = ⟨ x = λself.3, inc =
    λself.self <-- x=λs.((self<=x)+1) ⟩
  - » o <= inc

    = (λself.self <-- x=λs.((self<=x)+1)) o

    = o <-- x=λs.((o<=x) + 1)

    = o <-- x=λs.4

    = ⟨ x = λself.4, inc = λself. self <-- x= ... ⟩

Examples

- Use of Self
  - » o = ⟨ x = λself.3, inc =
    λself.self <-- x=λs.((self<=x)+1) ⟩
  - » o <= inc

    = (λself.self <-- x=λs.((self<=x)+1)) o

    = o <-- x=λs.((o<=x) + 1)

    = o <-- x=λs.4

    = ⟨ x = λself.4, inc = λself. self <-- x= ... ⟩

Examples

- Non-termination
  - » bee = ⟨buzz = λself.(self <= buzz) ⟩
  - » bee <= buzz

  --> (λself. self <= buzz) bee

  --> bee <= buzz

Numerals as Pure Objects

- Represent number n by object

  n = ⟨rep = λself. λx. x<=m<= ... <=m,

  succ = λself. ⟨self ←

  rep = λs. λx.(self<=rep x) <=m ⟩,

  pred = λself. ⟨self ← rep = . . . ⟩ ⟩

- Fixed-point operator also definable using object operations

Choice of Object Primitives

- Create, Select, Extend, Override
- With extension, simplify creation

  build ⟨m_1 = e_1, ..., m_k = e_k⟩

  by ⟨⟨⟨⟩ ←+ m_1 = e_1⟩ ... ←+ m_k = e_k⟩
- Typing complications with extension
  - » Need intermediate objects to make sense
    - – may want three methods, need sensible obj with 2
  - » Permutation rules in op semantics

Alternatives in the literature

- Mitchell 1990, FHM 93, ...
  - » empty object, selection, extension, override
- Abadi, Cardelli
  - » direct construction, selection, overide
- Trade-offs
  - » extensible objects are more complicated, *especially* with method specialization
  - » inheritance modelled directly with extension

Methods vs Functions

- Calculus presented so far:
  - » $\langle$ ... m = λself. e ... $\rangle$
  - » functions used for data fields, methods
- Alternatives [Abadi/Cardelli, Obliq]
  - » method: m = meth(self, args) .... end
  - » data field: v = exp
  - » function: fun (arg) ... end
- Restrictions can be useful in practice

Imperative Objects

- Various studies [Bruce, Abadi/Cardelli, Harper/Mitchell, ...]
- Similar to extending lambda calc with asg [Felleisen, Harper, ...]
- Can be viewed as special case of concurrency [Walker, Pierce, ...]
- *Cloning* becomes interesting operation

Object types

- Type determines set of operations
  - » can choose any combination of
    select, extend, override
- Subtyping depends on set of operations
  - » more operations, weaker subtyping
- Include type recursion or keep separate
  - » obj t. $\langle\langle$ ... m: τ ... $\rangle\rangle$ restricted recursive type
  - » some simplification, some idiosyncracies

Basic Forms of Object Types

$\langle\langle$ m_1 : τ_1, ... , m_k : τ_k $\rangle\rangle$_{ops}

where

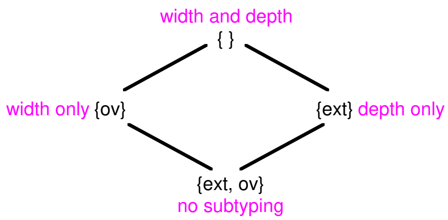
- » m_1, ..., m_k are method (or field) names
- » τ_1, ... , τ_k are types
- » ops is a subset of {sel,ext,ov} , for
  select, extend, override

Basic Forms (cont'd)

- Example
  $\langle\langle$ x : int, y: int $\rangle\rangle$_{sel, ov}
  "points", with selection and override
- Confusing to have so many options
  - » will focus on subset of possibilities
- Also need more (in a few slides ...)
  - » recursive types for common object interfaces
  - » first-class "rows" for certain forms of poly...

Object operations and subtyping

- Selection compatible with width, depth
- For other operations, a small lattice:



Review: forms of subtyping

- Width subtyping

  ⟨⟨m_1 : τ_1, ..., m_k : τ_k , n: σ ⟩⟩
  <: ⟨⟨ m_1 : τ_1, ..., m_k : τ_k ⟩⟩

- Depth subtyping

$$\frac{\sigma\_1 <: \tau\_1, \quad \ldots, \quad \sigma\_k <: \tau\_k}{\langle\langle m\_1 : \sigma\_1, \ldots, m\_k : \sigma\_k \rangle\rangle \;<:\; \langle\langle m\_1 : \tau\_1, \ldots, m\_k : \tau\_k \rangle\rangle}$$

Examples

- Width subtyping

  ⟨⟨ x : int, y : int, c : color ⟩⟩
  <: ⟨⟨ x : int, y : int ⟩⟩

- Depth subtyping

$$\frac{\text{manager} <: \text{employee}}{\langle\langle \text{name : string, sponsor : manager} \rangle\rangle \;<:\; \langle\langle \text{name : string, sponsor : employee} \rangle\rangle}$$

Conflict: Width and Extension

- Suppose

  ob : ⟨⟨ f:int->int, g:int->int⟩⟩_{sel,ext}

- If we asume width subtyping

  ⟨⟨ f:int->int, g:int->int⟩⟩_{sel,ext}
  <: ⟨⟨ f:int->int ⟩⟩_{sel,ext}

- Then we can form extended object

  ⟨ob <--+ g = e : bool ⟩

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

  ob = ⟨ f = λself.(... self <= g ...), g = ...⟩

Conflict: Depth and Override

- Suppose

  ob : ⟨⟨ x : posint, logX : real ⟩⟩_{sel,ov}
  = ⟨x = λself.3, logX = λself. log(self<=x) ⟩

- If we asume depth subtyping

  ⟨⟨x : posint, logX : real ⟩⟩_{sel,ov}
  <: ⟨⟨ x : int, logX : real ⟩⟩_{sel,ov}

- Then we can override x by

  ⟨ob <-- x = λself. -5 ⟩

- Violates assumption made in typing logX

Example taken from [Abadi and Cardelli 94]

Conclusion from this?

- Want to support subtyping, inheritance
- Subtyping:
  - must drop extension and/or override
- Inheritance:
  - must have some extensible data structure
  - extensible objects, records, or modules
- Typed OOL's must have two distinct ways of aggregating methods and fields

Models of Inheritance

- Extensible objects [M, FHM, FM, ...]
  - » separate extensible prototypes from non-extensible proper objects (without ext, ov)
- Inheritance based on records or modules
  - » Recursively-defined records as objects, combine generators for inheritance [Cook,...]
  - » Record of traits [Abadi, Cardelli]
  - » Other related approaches

Roadmap

- Few more typing concepts
  - » Recursive types
  - » Rows
  - » Typing rules for extensible objects
- Use class construct to compare object calculi
  - » inheritance by extensible objects
  - » inheritance by record of constructor, traits

Recursive Types

- Occur frequently
  - » Point = $\langle\langle$ x : int, y : int, move : int $\times$ int -> Point $\rangle\rangle$
- Standard approach
  - » Point = μ t. $\langle\langle$ x:int, y:int, move: int x int -> t $\rangle\rangle$
- Two variants
  - » $\mu t.\ A(t) = A(\mu t.\ A(t))$ with other equations
  - » $\mu t.\ A(t) \cong A(\mu t.\ A(t))$ by fold and unfold

Subtyping Recursive Types

- Basic Rule

$$\frac{s <: t \ \vdash\ A(s) <: B(t)}{\mu s.\ A(s) <: \mu t.\ B(t)}$$

- Useful in many contexts
- Introduces complications
  - » algorithmics of type checking
  - » type equality

Specialized Alternative

- Use isomorphism $\mu t.\ A(t) \cong A(\mu t.\ A(t))$
- Special syntax obj t. A(t)
  - » combine fold with object formation
  - » combine unfold with method invocation
- Simplified subtyping (A, B covariant)

$$\frac{t : Type \ \vdash\ A(t) <: B(t)}{\text{obj } t.\ A(t) <: \text{obj } t.\ B(t)}$$

- Allows special treatment of meth spec

Technical Device: Rows

- A row is a set of name:type pairs
- Operator from Rows to Types
  - » obj :: (Type => Row) => Type
  - » write obj t.R for obj (λt.R)
- Use only "subtyping" on Rows
  - » width
  - » depth

Specific Type System

- Extensible objects
- Row expressions
  - » $R ::= r \mid \langle\langle\rangle\rangle \mid \langle\langle R \mid m:\tau \rangle\rangle \mid \lambda t.R \mid R\tau$
  - » relations <: in width, depth
- Types contain rows
  - » $\tau ::= t \mid \tau \to \tau \mid$ pro R | obj R
  - » write pro t.R for pro (λt.R)

Prototypes vs Objects

- Prototypes `pro v.⟨⟨ m:... ⟩⟩`
  - » Extension and overriding
  - » No subtyping between `pro` types
- Objects `obj v.⟨⟨ m:... ⟩⟩`
  - » Selection only (no extension, override)
  - » Subtyping between `obj` types
  - » Method body may override a method
- Conversion: `pro R <: obj R` (R covar)

Typing Rule for Messages

$$\frac{e : \texttt{obj } t.R \qquad R <: \langle\langle m:\tau \rangle\rangle}{e \Leftarrow m \ : \ [\texttt{obj } t.R/t]\ \tau}$$

- Combination of selection and application
- Includes unfolding recursive type

Example

p = ⟨ x = λself. 3,
inc = λself. self <-- x=λs.((self<=x)+1) ⟩
: obj t. ⟨⟨ x : int, inc : t ⟩⟩
p <= inc
: obj t. [⟨⟨ x : int, inc : t ⟩⟩ /t] t
: obj t. ⟨⟨ x : int, inc : t ⟩⟩

Similar to Eiffel *like current* (but sound)

Typing Rule for Extension

$$\frac{\begin{array}{c} e : \texttt{pro } u.R \\ u : T \vdash R :: \{n\} \\ r <:_w \lambda u.\langle\langle R \mid n:\tau \rangle\rangle \ \vdash e' \ : \quad . \\ \quad [(\texttt{pro } u.ru)/u]\,(u \to \tau) \end{array}}{\langle e \ \texttt{<-+} \ n=e' \rangle : \texttt{pro } u.\langle\langle R \mid n:\tau \rangle\rangle}$$

- Combines extension, folding of rec type
- Method specialization useful in incremental construction of objects

Example

$$\frac{\begin{array}{c} \langle\rangle \ : \ \text{pro } t.\ \langle\langle\rangle\rangle \\ t : \text{Type} \vdash \ \langle\langle\rangle\rangle :: \{x\} \\ r <: \lambda t.\ \langle\langle x{:}\text{int} \rangle\rangle \vdash \ \lambda\text{self}.3 : (\text{pro } t.\ rt) \to \text{int} \end{array}}{\langle x = \lambda\text{self}.3 \rangle \ : \ \text{obj } t.\ \langle\langle x : \text{int} \rangle\rangle}$$

$$\frac{\begin{array}{c} t : \text{Type} \vdash \langle\langle x{:}\text{int} \rangle\rangle :: \{\text{inc}\} \\ r <: \lambda t.\ \langle\langle x{:}\text{int, inc}{:}t \rangle\rangle \vdash \lambda\text{self.self} \ \texttt{<--} \ x = \lambda s.((\text{self}{<=}x)+1) \\ :\ (\text{pro } t.\ rt) \to (\text{pro } t.\ rt) \end{array}}{\langle x = \lambda\text{self}.3,\ \text{inc} = \lambda\text{self}.\ \ldots \rangle \ : \ \text{obj } t.\ \langle\langle x : \text{int, inc} : t \rangle\rangle}$$

inc returns object *like current* when this object extended

Typing Rule for Override

$$\frac{\begin{array}{c} e : \texttt{pro}\ u.R \\ u{:}Type \vdash R <:_w \langle\!\langle m{:}\tau \rangle\!\rangle \\ r <:_w \lambda u.R \vdash e' : [(\texttt{pro}\ u.ru)/u](u \to \tau) \end{array}}{\langle e \leftarrow m{=}e' \rangle : \texttt{pro}\ u.R}$$

Combines redefinition, folding of recursive type.

Example

s : (pro t. r't)

self	: pro t. ⟨⟨ x : int, inc : t ⟩⟩
(self <= x)	: int
(self <= x)+1	: int

r <: λt. ⟨⟨ x:int, inc:t ⟩⟩

self : (pro t. rt)

$$\frac{\begin{array}{l} \text{self} : (\text{pro } t.\ rt) \\ t{:}Type \vdash r\,t <: \langle\!\langle x{:}int \rangle\!\rangle \\ r' <: r \vdash \lambda s.(\text{self} <= x)+1 : (\text{pro } t.\ r't) \to int \end{array}}{\vdash \text{self} <\!\!-\!\!-\ x = \lambda s.(\text{self} <= x)+1 : (\text{pro } t\ .rt)}$$

⊢ λself.self <-- x =λs.((self<=x)+1) : (pro t. rt) → (pro t. rt)

Review

o = ⟨x = λself.3, inc = λself.self<-- x = λs.((self<=x)+1) ⟩

: obj t. ⟨⟨ x : int, inc : t ⟩⟩

o <= inc

----> (λself.self<-- x = λs.((self<=x)+1)) o

----> o <-- x = λs.((o<=x)+1)

----> ⟨x = λself.4, inc = λself.self <-- x = λs.((self<=x)+1) ⟩

Technical Results [Fisher,M]

Subject Reduction Theorem

If $\Gamma \vdash e : \tau$ is derivable and e ---> e
then $\Gamma \vdash e' : \tau$ is derivable.

Type Soundness Theorem

If $\varnothing \vdash e : \tau$ is derivable
then eval(e)≠error by specific strategy.

Type system prevents *message not understood.*

Final Topic: Classes from Objects

- Class provides
  - type definition
  - implementation of objects
  - control over initialization
  - basis for inheritance
    - distinguish public, private, protected
    - control implementation in derived classes
- Goal: Represent classes using extensible objects and abstract data types

Interfaces vs "Class types"

- 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 : elem → bool, union : set → set ⟩⟩
- C++ types fix part of implementation

Sample Class Construct

class class_name { : superclass }
  constructor
      name : type → class_name
          = implementation
  private { name : type = implementation }*
  public { name : type = implementation }*
end

Example (implementation omitted)

class Point
  constructor newP : int → Point
  private x : int
  public setX : int → Point
      getX : int
end
class ColorPoint : Point
  constructor newCP : color → int → ColorPoint
  private c :color
  public setC : color → ColorPoint
      getC : color
end

Interfaces for Point Classes

- Point Interfaces
  - within implementation — pro P_priv
  - to derived class — pro P_pub
  - to client program — obj P_pub
- ColorPoint Interfaces
  - within implementation — pro CP_priv
  - to derived class — pro CP_pub
  - to client program — obj CP_pub

For "protected," use P_priv <: P_prot <: P_pub

Interface Types

P_pub = $\lambda t.\ \langle\langle$ setX : int → t, getX : int $\rangle\rangle$
P_priv = $\lambda t.\ \langle\langle$ x : int, setX : int → t, getX : int $\rangle\rangle$

CP_pub = $\lambda t.\ \langle\langle$ p t | setC : color→ t, getC : color $\rangle\rangle$
CP_priv = $\lambda t.\ \langle\langle$ p t | c:color, setC:color→ t, getC:color $\rangle\rangle$

The CP interfaces are written using a row variable p that will be bound to existentially bound "sub-rows" of P_pub or P_priv in the program.

Point Implementation

```
λiX.⟨   x = λself. iX,
      setX = λself. λnewX.
          ⟨ self <- x = λself. newX ⟩,
      getX = λself.  self <= x
     ⟩
```

Other operations in same style, e.g.,

```
   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_pub :: T -> ({c,setC, getC},ø)
     with newPoint : int -> pro v. p v
       is {p <: P_pub = P_priv, P_impl }
       in
   Abstype cp <: CP_pub :: T -> (ø,ø)
        with newCPoint : int -> col ->prov v. cp v
            is {cp <: P_pub = CP_priv, CP_impl }
            in
              let newPoint   : int -> obj v. p v            = newPoint
                  newCPoint : int -> col -> obj v. cp v = newCPoint
              in       ...   program   ...
```

Properties

- Restricted access to objects
  - derived class
    - see public (or protected) methods
    - extend objects, override methods
  - program
    - see public methods
    - send messages but cannot extend or override
    - subtyping
- Standard properties of data abstraction

Program Structure with `protected`

```
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 [Fisher,M]

- Object calculus with
  - Extensible objects, row variable polymorphism, variance annotations, negative information about methods, abstract types
- Prove type soundness using
  - operational semantics
  - analysis of typing rules
- Appeal to prior work on data abstraction

Insight

- Link between inheritance and subtyping
  - subtypes of abstract types can *only* be produced by extension
- C++ `private virtual` problem
- Tentative design for "ML 2000"
  - modules + object primitives => OOP
  - but need extensible objects, subtyping, ...
  - type system seems very complicated

Assorted References

- [Abadi/Cardelli]
  - object calculi without extension; imperative
- [Castagna, et al.]
  - multimethods
- [Pierce/Turner]
  - exis types model, friend functions, etc.
- [Fisher, Di Blasio]
  - concurrent extension of calculus here

Web pages

- My home page for slides, other papers
  http://theory.stanford.edu/people/jcm
- Cardelli always has good stuff
  http://www.research.digital.com/SRC/personal/Luca_Cardelli/home.html
- Page of object/types people
  http://cuiwww.unige.ch/OSG/Hop/types.html

Don't write any of this down. Use a search engine.

Summary

- ✓ Foundations; type-theoretic framework
- ✓ Concepts in object-oriented programming
- ✓ Decomposition of OOP into parts
- ✓ Formal models of objects

➪ Questions ???

Remember in 5 years

- programming languages are not just a matter of taste, but subject to scientific study
- types determine the building blocks oflanguages
- mathematical semantics is essential for the design of sound proof methods
- proof methods are essential if you really want to be sure about some aspect of a program