Type Systems and Object-Oriented Programming (III)

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

Varieties of OO languages

- class-based languages
  - » behavior of object determined by its class
  - » objects created by instantiating a classes
- object-based
  - » objects defined directly
    - – in total, cloning, extension, override
- multi-methods
  - » code-centric instead of object-centric
  - » run-time overloaded functions

Method invocation

- single dispatch:
  - » receiver.message(object, ..., object)
  - » code depends on receiver only
- multiple dispatch ("multi-methods")
  - » operation(object, ... , object)
  - » code may depend on types of all objects

Comparison

- single dispatch
  - » data hidden in objects
  - » cannot access private data of parameters
- multiple dispatch
  - » better for symmetric binary operations
  - » loss of encapsulation
  - » but see work by Chambers and Leavens
  - » curried multiple dispatch =? single dispatch

These lectures

- Class-based, object-based languages
- Single-dispatch method invocation

- References for other languages
  - » Cecil, CommonLisp are multimethod-based
  - » Foundations by Castagna, et al., others

Intuitive picture of objects

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

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

- Program sends messages to objects

Class-based 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
- Java 1990's

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

Priority Q, similar to Queue

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

Abstract Data Types

- Guarantee invariants of data structure
  - only functions of the data type have access to the internal representation of data
- Limited "reuse"
  - Cannot apply queue code to pqueue, except by explicit parameterization, even though signatures identical
  - Cannot form list of points, colored points

Dynamic Lookup

- receiver <= operation (arguments)
- code depends on receiver and operation
- This is may be achieved in conventional languages using record with function components.

OOP in Conventional Lang.

- Records provide "dynamic lookup"
- Scoping provides another form of encapsulation

Try object-oriented programming in ML

Stacks as closures

```
fun create_stack(x) =
  let val store = ref [x]    in
    {push = fn (y) =>
            store := y::(!store),
     pop  = fn () =>
            case !store of
              nil => raise Empty |
              y::m => (store := m; y)
     } end;
```

Does this work ???

- Depends on what you mean by "work"
- Provides
  - » encapsulation of private data
  - » dynamic lookup
- But
  - » cannot substitute extended stacks for stacks
  - » only weak form of inheritance
    - – can add new operations to stack
    - – not mutually recursive with old operations

Weak Inheritance

```
fun create_stack(x) =
   let val store = ... in {push = ..., pop=...}  end;
fun create_dstack(x) =
  let val stk = create_stack(x)  in
    { push = stk.pusk,  pop= stk.pop,
      dpop = fn () => stk.pop;stk.pop    }  end;
```

But cannot similarly define nstack from dstack with pop redefined, and have dpop refer to new pop.

Weak Inheritance (II)

```
fun create_dstack(x) =
  let val stk = create_stack(x)  in
    { push = stk.push,  pop= stk.pop,
      dpop = fn () => stk.pop;stk.pop  }  end;
fun create_nstack(x) =
  let val stk = create_dstack(x)  in
    { push = stk.push,  pop= new_code,
      dpop = fn () => stk.dpop   }  end;
```

Would like dpop to mean "pop twice".

Weak Inheritance (II)

```
fun create_dstack(x) =
  let val stk = create_stack(x)  in
    { push = stk.push,  pop= stk.pop,
      dpop = fn () => stk.pop;stk.pop  }  end;
fun create_nstack(x) =
  let val stk = create_dstack(x)  in
    { push = stk.push,  pop= new_code,
      dpop = fn () => stk.dpop  }  end;
```

New code does not alter meaning of dpop.

Inheritance with Self (almost)

```
fun create_dstack(x) =
  let val stk = create_stack(x)  in
    { push = stk.push,  pop= stk.pop,
      dpop = fn () => self.pop; self.pop}  end;
fun create_nstack(x) =
  let val stk = create_dstack(x)  in
    { push = stk.push,  pop= new_code,
      dpop = fn () => stk.dpop   }  end;
```

Self interpreted as "current object itself"

Summary

- Have encapsulation, dynamic lookup in traditional languages (e.g., ML)
- Can encode inheritance:
  - » can extend objects with new fields
  - » weak semantics of redefinition
    - – NO *"SELF"*; NO *"OPEN RECURSION"*
- Need subtyping as language feature

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
  subtype polymorphism provides extensibility
- Property of types, not implementations

Object Interfaces

- Type
  Counter =
  ⟨⟨ val : int, inc : int -> Counter ⟩⟩
- Subtyping
  RCounter = ⟨⟨ val : int, inc : int -> RCounter,
  reset : RCounter ⟩⟩
  <: Counter

Facets of subtyping

- Covariance, contravariance
- Width and depth
- For recursive types
- F-bounded and higher-order

Covariance

- Definition
  - A type form $\tau(...)$ is covariant if
    $s <: t$ implies $\tau(s) <: \tau(t)$
- Examples
  - $\tau(x) = int \times x$ (cartesian product)
  - $\tau(x) = int \rightarrow x$ (function type)

Contravariance

- Definition
  - A type form $\tau(...)$ is contravariant if
    $s <: t$ implies $\tau(t) <: \tau(s)$
- Example
  - $\tau(x) = x \rightarrow bool$
    Specifically, if $int <: real$, then
    $real \rightarrow bool <: int \rightarrow bool$
    and *not* conversely

Non-variance

- Some type forms are neither covariant nor contravariant
- Examples
  - $\tau(x) = x \rightarrow x$
  - $\tau(x) =$ Array[1..n] of x

Arrays are covariant for read, contravariant for write, so non-variant if both are allowed.

Simula Bug

- Statically-typed program with A <: B

```
proc asg (x : Array[1..n] of B)
   begin;
      x[1]  :=   new B;   /* put new B value in B location */
   end;
y : Array[1..n] of A;
asg( y ):
```

- Places a B value in an A location
- Also in Borning/Ingalls, Eiffel systems

Subtyping for records/objects

- Width subtyping

$\langle\langle m\_1 : \tau\_1, \ldots, m\_k : \tau\_k,\ n: \sigma \rangle\rangle$
<: $\langle\langle m\_1 : \tau\_1, \ldots, m\_k : \tau\_k \rangle\rangle$

- 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

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

Subtyping for recursive types

- Basic rule

If s <: t implies A(s) <: B(t)
Then $\mu t.A(t)$ <: $\mu t.B(t)$

- Example
  - A(t) = ⟨⟨ x : int, y : int, m : int --> t ⟩⟩
  - B(t) = ⟨⟨ x : int, y : int, m : int --> t, c : color ⟩⟩

Subtyping recursive types

- Example
  - Point = ⟨⟨ x : int, y : int, m : int --> Point ⟩⟩
  - Col_Point = ⟨⟨ x : int, y : int,
    m : int --> Col_Point , c : color ⟩⟩
- Explanation
  - If p : Point and expression e(p) is OK, then if q : Col_Point then e(q) must be OK
  - Induction on the # of operations applied to q.

Contravariance Problem

- Example
  - Point = ⟨⟨ x : int, y : int,
    equal : Point --> bool ⟩⟩
  - Col_Point = ⟨⟨ x : int, y : int, c : color,
    equal : Col_Point --> bool ⟩⟩
- Neither is subtype of the other
  - Assume   p: Point, q: Col_Point
  - Then   q <= equal p   may give type error.

Parametric Polymorphism

- General "max" function
  - » max(greater, a,b) = if greater(a, b) then a else b
- How do we assign a type?
  - » assume a:t, b:t for some type t
  - » need greater : $t \times t \to$ bool
- Polymorphic type
  - » max : $\forall$ t. $(t \times t \to \text{bool}) \times t \times t \to t$

Subtyping and Parametric Polymorphism

- Object-oriented "max" function
  - » max(a,b) = if a.greater(b) then a else b
- How do we assign a type?
  - » assume a:t, b:t for some type t
  - » need $t <: \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle$
- F-bounded polymorphism
  - » max : $\forall t <: \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle .\ t \times t \to t$

Why is type quantifier useful?

- Recall conditions of problem
  - » conditional requires a:t, b:t for some type t
  - » need $t <: \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle$
- "Simpler" solution
  - » use type $\mu t. \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle$
  - » max : $\mu t.\langle\langle .. \rangle\rangle \times \mu t.\langle\langle .. \rangle\rangle \to \mu t.\langle\langle .. \rangle\rangle$
- However ...
  - » not supertype due to contravariance
  - » return type has only greater method

Alternative

- F-bounded polymorphism
  - » max : $\forall t <: \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle .\ t \times t \to t$
- Higher-order bounded polymorphism
  - » max : $\forall F <: \lambda t. \langle\langle \text{greater} : t \to \text{bool} \rangle\rangle .\ \mu F \times \mu F \to \mu F$
- Similar in spirit but technical differences
  - » Transitive relation
  - » "Standard" bounded quantificaion

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 ...?

Method Specialization

- Change type of method to more specialized type
- May not yield subtype due to contravariance problem
- Illustrates difference between inheritance and subtyping
- Also called "mytype specialization" [Bruce]; Eiffel "like current"

Covariant Method Specialization

- Assume we have implemenation for

  Point = ⟨⟨ x : int, y : int,

  move : int × int --> Point⟩⟩
- Extension with color *could* give us an object of type

  Col_Point = ⟨⟨ x : int, y : int, c : color

  move : int × int --> Col_Point ⟩⟩
- Inheritance results in a subtype

Contravariant Specialization

- Assume we have implemenation for

  Point = ⟨⟨ x : int, y : int, eq : Point --> bool ⟩⟩
- Extension with color and redefinition of equal could give us an object of type

  Col_Point = ⟨⟨ x : int, y : int, c : color

  eq : Col_Point --> bool ⟩⟩
- Inheritance does not result in a subtype

Summary of Part III

- Language support divided into four parts
  - encapsulation, dynamic lookup, subtyping, inheritancs
- Subtyping and inheritance require extension to conventional languages
- Subtyping and inheritance are distinct concepts that are sometimes correlated