Type Systems and Object-Oriented Programming

John C. Mitchell
Stanford University

Plan for these lectures

- 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
- Insight may be useful in software design
- Trade-offs in program structure
- Possible research opportunities
  - » language design
  - » formal methods
  - » system development, reliability, security

Applications of type systems

- Methodology
  - » Design expressed through types relationships.
- Security
  - » Prevent “message not understood.''
- Efficiency
  - » Eliminate run-time tests.
  - » Optimize method lookup.
- Analysis
  - » Debugger, tree-shaking, etc.

Research in type systems

- Repair insecurities and deficiencies in existing typed languages.
- Find better type systems
  - » Flexible OOP without type case and casts
- Basis for formal methods
  - » Formulas-as-types analogy
  - » No Hoare logic for sequential objects

Specific Opportunities

- Typed, sequential OOP

Conventional	Object-oriented
Lisp	Smalltalk
ML	??
C	C ++

- Improvements in C++, Java
- Concurrency, distributed systems

Type Systems and Object-Oriented Programming

PART I

John C. Mitchell
Stanford University

Foundations for Programming

- Computability theory
- Lambda Calculus
- Denotational Semantics
- Logics of Programming

Computability Theory

- A function f : N -> N is *computable* if
  - » there is a program that computes it
  - » there is an idealized machine computing it
- Reductions: compute one function using another as subroutine
- Compare degrees of computability
- Some functions cannot be computed

Inductive def'n of computable

- Successor function, constant function, projection f(x,y,z) = x are computable
- Composition of computable functions
- Primitive recursion

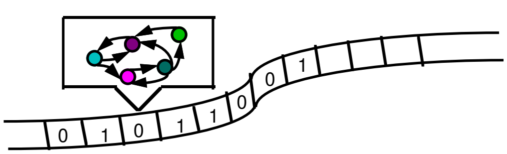
  f(0,x) = g(x)

  f(n+1, x) = h(n, x, f(n,x))
- Minimalization

  f(x) = the least y such that g(x,y) = 0

Turing machine

- infinite tape with 0, 1, blank
- read/write tape head
- finite-state control



Strengths of Computability

- Robust theory
  - » equiv language and machine definitions
- Definition of "universal"
  - » Confidence that all computable functions are definable in a programming language
- Useful measures of time, space

Weaknesses

- Formulated for numeric functions only
- Need "equivalence" for program parts
  - » optimization, transformation, modification
- Limited use in programming pragmatics
  - » Are Lisp, Pascal and C equally useful?
  - » Turing Tarpit

Lambda Calculus

- early language model for computability
- syntax
  - » function expressions
  - » free and bound variables; scope
- evaluation of expressions
- equational logic

Untyped Lambda Calculus

- Write $\lambda x.\ e$ for "the function f with f(x) = e"
- Example

  $\lambda f\ .\ f(f(a))$ apply function argument twice to a
- Symbolic evaluation by "reduction"

  $(\ \lambda f\ .\ f(f(a))\ )(\ \lambda x\ .\ b)$

  => $(\ \lambda x\ .\ b)\ (\ (\ \lambda x\ .\ b)\ \ a\ )$

  => $(\ \lambda x\ .\ b)\ (\ b\ )$

  => b

Syntactic concepts

- Variable x is *free* in f(g(x))
- Variable y is *bound* in $(\lambda y.\ y(x))(\lambda x.\ x)$
- The *scope* of binding λy is y(x)
- α–conversion

  $(\lambda x.\ \ldots\ x\ \ldots\ x \ldots)\ =\ (\lambda y.\ \ldots\ y\ \ldots\ y \ldots)$
- Declaration

  let x = e_1 in e_2 ::= $(\ \lambda x.\ e\_2)\ e\_1$

The syntax behind the syntax

```
function f(x);
  begin;
    return ((x+5)*(x+3));
  end;
f(4+2);
```

is just another way of writing

let f = $(\ \lambda x.\ ((x+5)*(x+3))\ )$ in f(4+2)

Equational Proof System

(α)

$(\lambda x.\ \ldots\ x\ \ldots\ x \ldots) = (\lambda y.\ \ldots\ y\ \ldots\ y \ldots)$

(β)

$(\lambda x.\ e\_1)\ e\_2\ =\ [e\_2/x]\ e\_1$

rename bound var in e_1 to avoid capture

(η)

$\lambda x.\ e\ x\ =\ e$ x not free in e

Rationale for (η)

- Axiom: $\lambda x.\ e\ x\ =\ e$ x not free in e
- Suppose e is function expression $\lambda y.\ e'$
- Then by (β) and (α) we have

  $\lambda x.\ (\lambda y.\ e')\ x\ =\ \lambda x.\ ([x/y]\ e')\ =\ \lambda y.\ e'$
- But (η) not needed in computation

Why the Greek letter λ ?

- Dana Scott told me this once:
  - Dana asked Addison, at Berkeley
  - Addison is Church's son-in-law
  - Addison had asked Church
  - Church said, "eeny, meeny, miny, mo"

Symbolic Evaluation (reduction)

(α) (λx. ... x ... x ...) = (λy. ... y ... y ...)

(β) (λx. e_1) e_2 => [e_2/x] e_1

rename bound var in e_1 to avoid capture

(η) λx. e x => e   x not free in e

but this is *not* needed for closed "programs"

Lambda Calculus Hacking (I)

- Numerals

  n = λf. λx. f (f ... f(x) ...)   with n f's
- Successor

  Succ n = λf. λx. f (n (f) (x))
- Addition

  Add n m = n Succ m

Lambda Calculus Hacking (II)

- Nontermination

  (λx. x (x)) (λy. y (y))

  => [(λx. x (x)) / y] y (y)

  = (λy. y (y)) (λy. y (y))
- Fixed-point operator Y f = f (Y f)

  Y = λf. (λx. f (x (x))) (λx. f (x (x)))

  Y f => (λx. ...)(λx. ...) => f ((λx. ...)(λx. ...))

Lambda Calculus Hacking (III)

Write factorial function

f(x) = if x=0 then 1 else x*f(x-1)

As

Y (factbody)

where

factbody =

λf. λx.Cond (Zero? x)(1)(Mult(x)(f (Pred x)))

Lambda Calculus Hacking (IV)

Calculate by reduction

Y (factbody) (5)

=>

λf. λx.Cond (Zero? x)(1)(Mult(x)(f (Pred x)))
(Y (factbody)) (5)

=>

Cond (Zero? 5) (1)
(Mult(5) ((Y (factbody)) (Pred 5))

Insight

- A recursive function is a fixed point
  fun f(x) = if x=0 then 1 else x*f(x-1)
  means
  let f be the fixed point of the functional
  λf. λx.Cond (Zero? x)(1)(Mult(x) (f(Pred x)))
- Not just "*mathematically*" but also "*computationally*"

Extensions of Lambda Calculus

- Untyped lambda calculus is unstructured theory of functions
- Add types
  - separate functions from non-functions
  - add other kinds of data
    - integers, booleans, strings, stacks, trees, ...
  - provide program-structuring facilities
    - modules, abstract data types, ...

Pros and Cons of Lambda Calculus

- Includes syntactic structure of programs
- Equational logic of programs
- Symbolic computation by reduction
- Mathematical structure (with types) provided by categorical concepts
- Still, largely *intensional* theory with few mathematical methods for reasoning

Denotational Semantics

- Can be viewed as model theory of typed lambda calculus
- Interpret each function as continuous map on appropriate domains of values
- Satisfy provable equations, and more
- Additional reasoning principles (induction)

Type-theoretic framework

- Typed extensions of lambda calculus
- Programming lang. features *are* types
  - Recursive types for recursion
  - Exception types for exceptions
  - Module types for modules
  - Object types for objects
- Operational and denotational models
- Equational, other logics (Curry-Howard)

Imperative programs

- Traditional denotational semantics
  - » translate imperative programs to functional programs that explicitly manipulate a *store*
- Lambda calculus with assignment
  - » give direct operational semantics using location names
  - » (compositional denotational semantics only by method above, as far as I know)
- Similar issues for concurrency

Plan for these lectures

- ✓ Foundational framework

  type theory, operational & denotational sem.
- Principles of object-oriented programming
- Decomposition of OOP into parts
- Type-theoretic basis for OOP

Q: What is a type?

- Some traditional answers
  - » a set of values (object in a category)
  - » a set together with specified operations
- Bishop's constructive set
  - » membership predicate, equivalence relation
- Syntactic answer
  - » type expresion (or form)
  - » introduction and elimination rules
  - » equation relating introduction and elimination

Example: Cartesian Product

- » Type expression: $A \times B$
- » Introduction rule: $\dfrac{x : A \quad y : B}{\langle x, y\rangle : A \times B}$
- » Elimination rule: $\dfrac{p : A \times B}{\mathrm{first}(p) : A \quad \mathrm{second}(p) : B}$
- » Equations: intro ° elim = identity

$$\mathrm{first}\,\langle x, y\rangle = x \qquad \mathrm{second}\,\langle x, y\rangle = y$$

$$\langle \mathrm{first}(p), \mathrm{second}(p)\rangle = p$$

Adjoint Situation

- Natural Iso $\mathrm{Maps}(FA, B) \cong \mathrm{Maps}(A, GB)$
- Cartesian Product on category C
  - » Category $C \times C$ with $\langle f, g\rangle : \langle a, b\rangle \to \langle c,d\rangle$
  - » Functor $F : C \to C \times C$ with $F(a) = \langle a, a\rangle$
  - » Cartesian product is right adjoint of F
  - » $\mathrm{Maps}(\langle a, a\rangle, \langle b, c\rangle) \cong \mathrm{Maps}(a, b \times c)$

$$\frac{\langle a, a\rangle \longrightarrow \langle b, c\rangle}{a \longrightarrow b \times c}$$