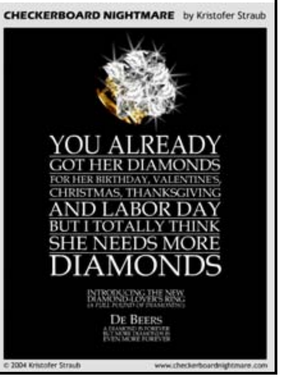
Modeling and Understanding Object-Oriented Programming

Cunning Plan: Focus On Objects

- A Calculus For OO
- Operational Semantics
- Type System
- Expressive Power
- Encoding OO Features



The Need for a Calculus

- There are many OO languages with many combinations of features
- We would like to study these features formally in the context of some primitive language
  - Small, essential, flexible
- We want a "λ-calculus" or "IMP" *for objects*

Why Not Use λ-Calculus for OO?

- We *could* define some aspects of OO languages using λ-calculus
  - e.g., the operational semantics by means of a translation to λ-calculus
- But then the notion of object be secondary
  - Functions would still be first-class citizens
- Some typing considerations of OO languages are hard to express in λ-calculus
  - i.e., object-orientation is not simply "syntactic sugar"

Object Calculi Summary

- As in λ-calculi we have
  - operational semantics
  - denotational semantics
  - type systems
  - type inference algorithms
  - guidance for language design
- We will actually present a family of calculi
  - typed and untyped
  - first-order and higher-order type systems
- We start with an untyped calculus

An Untyped Object Calculus

- An object is a collection of methods
  - Their order does not matter
- Each method has
  - A bound variable for "self" (denoting the host object)
  - A body that produces a result
- The only operations on objects are:
  - Method invocations
  - Method update

Untyped Object Calculus Syntax

- Syntax:

  a, b ::= x — variables
  | $[m_i = \varsigma(x)\ b_i]$ — object constructor
  - ς is a variant of Greek letter σ
  - x is the local name for “self”

  | a.m — method invocation
  - no arguments (just the self)

  | $a.m \leftarrow \varsigma(x)\ b$ — method update
  - this is an expression !
  - the result is a copy of the object with one method changed

  - This is called the untyped ς-calculus (Abadi & Cardelli)

First Examples

- An object o with two methods m_1 and m_2
  - m_1 returns an empty object
  - m_2 invokes m_1 through self

    $o = [m_1 = \varsigma(x)\ [],\ \ m_2 = \varsigma(x)\ x.m_1\ ]$
- A bit cell with three methods: value, set and reset
  - value returns the value of the bit (0 initially)
  - set sets the value to 1, reset sets the value to 0
  - models state without λ/IMP (objects are primary)

  b = [value = $\varsigma(x)$. 0,
  set = $\varsigma(x)$. x.value $\leftarrow \varsigma(y)$. 1,
  reset = $\varsigma(x)$. x.value $\leftarrow \varsigma(y)$. 0]

Operational Semantics

- $a \rightarrow b$ means that a reduces in one step to b
- The rules are: (let o be the object $[m_i = \varsigma(x).\ b_i\ ]$)

$$o.m_i \rightarrow [o/x]\ b_i$$

$$o.m_k \leftarrow \varsigma(y).\ b \rightarrow [m_k = \varsigma(y).\ b,\ \ m_i = \varsigma(x).\ b_i] \quad (i \in \{1,\dots, n\} - \{ k\})$$

- We are dealing with a calculus of objects
- This is a deterministic semantics (has the Church-Rosser or “diamond” property)

Expressiveness

- A calculus based only on methods with “self”
  - How expressive is this language? Let’s see.
  - Can we encode languages with fields? Yes.
  - Can we encode classes and subclassing? Hmm.
  - Can we encode λ-calculus? Hmm.
- Encoding fields
  - Fields are methods that do not use self
  - Field access “o.f” is translated directly
    - to method invocation “o.f”
  - Field update “o.f $\leftarrow$ e” is translated to “o.f $\leftarrow \varsigma(x)$ e”
  - We will drop the $\varsigma(x)$ from field definitions and updates

As Expressive As λ

- Encoding functions
  - A function is an object with two methods
    - arg - the actual value of the argument
    - val - the body of the function
  - A function call updates “arg” and invokes “val”
- A conversion from λ-calculus expressions

  $\underline{x}$ = x.arg *(read the actual argument)*
  $\underline{e_1\ e_2}$ = $(\underline{e_1}.arg \leftarrow \varsigma(y)\ \underline{e_2}).val$
  $\underline{\lambda x.\ e}$ = $[arg = \varsigma(y)\ y.arg,\ val = \ \varsigma(x).\ \underline{e}\ ]$
  - The initial value of the argument is undefined
- From now on we use λ notation in addition to ς

λ-calculus into ς-calculus

- Consider the conversion of $(\lambda x.x)\ 5$

  Let $o = [\ arg = \varsigma(z)\ z.arg,\ val = \varsigma(x)\ x.arg]$
  $\underline{(\lambda x.x)\ 5}$ = $(o.arg \leftarrow \varsigma(y)\ 5).val$
- Consider now the evaluation of this latter ς-term
- Let $o' = [\ arg = \varsigma(y)\ 5,\ val = \varsigma(x)\ x.arg]$

  $(o.arg \leftarrow \varsigma(y)\ 5).val \rightarrow$
  $o'.val = [arg = \varsigma(y)\ 5,\ val = \varsigma(x)\ x.arg].val \rightarrow$
  $x.arg[o'/x] = o'.arg \rightarrow$
  $5[o'/y] = 5$

Encoding Classes

- A class is just an object with a "new" method, for generating new objects
  - A repository of code for the methods of the generated objects (so that generated objects do not carry the methods with them)
- Example: for generating o = [m_i = ς(x) b_i]

  c = [new = ς(z) [m_i = ς(x) z.m_i x],

  m_i = ς(self) λx. b_i]
  - The object can also carry "updateable" methods
  - Note that the m_i in c are fields (don't use self)

Class Encoding Example

- A class of bit cells

  BitClass = [new = ς(z). [val = ς(x) 0,
  set = ς(x) z.set x,
  reset = ς(x) z.reset x],
  set = ς(z) λx. x.val ⇐ ς(y) 1,
  reset = ς(z) λx. x.val ⇐ ς(y) 0]
- Example:

  BitClass.new → [val = ς(x) 0,
  set = ς(x) BitClass.set x,
  reset = ς(x) BitClass.reset x]
  - The new object carries with it its identity
  - The indirection through BitClass expresses the dynamic dispatch through the BitClass method table

Inheritance and Subclassing

- Inheritance involves re-using method bodies

  FlipBitClass =
  [new = ς(z) (BitClass.new).flip ⇐ ς(x) z.flip x,
  flip = ς(z) λx. x.val ⇐ not (x.val)]
- Example:

  FlipBitClass.new → [val = ς(x) 0,
  set = ς(x) BitClass.set x,
  reset = ς(x) BitClass.reset x,
  flip = ς(x) FlipBitClass.flip x]
  - We can model method overriding in a similar way

Object Types

- The previous calculus was *untyped*
- Can write invocations of nonexistent methods

  [foo = ς(x) ...].bogus
- We want a type system that guarantees that well-typed expressions only invoke existing methods
- First attempt:
  - An object's type *specifies the methods* it has available:

    A ::= [m_1, m_2, ..., m_n]
  - Not good enough:

    If o : [m, ...] then we still don't know if o.m.m is safe
  - We also need the type of the result of a method

First-Order Object Types. Subtyping

- Second attempt:

  A ::= [m_i : A_i]
  - Specify the available methods *and their result types*
- Wherever an object is usable another with more methods should also be usable
  - This can be expressed using (width) subtyping:

$$\frac{}{A < A} \qquad \frac{A < B \quad B < C}{A < C}$$

$$\frac{n \geq k}{[m_1 : A_1, \ldots, m_n : A_n] < [m_1 : A_1, \ldots, m_k : A_k]}$$

Typing Rules

making an object

$$\frac{\Gamma, x : A \vdash b_i : A_i}{\Gamma \vdash [m_i = \varsigma(x : A).\ b_i] : A}$$

invoking a method

$$\frac{\Gamma \vdash b : A \quad m_i : A_i \in A}{\Gamma \vdash b.m_i : A_i}$$

$$\frac{\Gamma \vdash b : A \quad m_i : A_i \in A \quad \Gamma, x : A \vdash b' : A_i}{\Gamma \vdash b.m_i \Leftarrow \varsigma(x) b' : A}$$

updating a method

Type System Results

- Theorem (Minimum types)
  - If $\Gamma \vdash$ a : A then there exists B such that for any A' such that $\Gamma \vdash$ a : A' we have B < A'
  - If an expression has a type A then it has a minimum (most precise) type B
- Theorem (Subject reduction)
  - If $\varnothing \vdash$ a : A and a $\to$ v then $\varnothing \vdash$ v : A
  - Type preservation. Evaluating a well-typed expression yields a value of the same type.

Type Examples

- Consider that old BitCell object

  o = [value = ς(x). 0,
  set = ς(x). x.value ← ς(y). 1,
  reset = ς(x). x.value ← ς(y). 0]

- An appropriate type for it would be

  BitType = [value : int, set : BitType, reset : BitType]
  - Note that this is a recursive type
  - Consider part of the derivation that o : BitType (for set)

$$\frac{x : \mathtt{BitType} \quad \mathtt{value} : \mathtt{int} \in \mathtt{BitType} \quad x : \mathtt{BitType}, y : \mathtt{BitType} \vdash 1 : \mathtt{int}}{x : \mathtt{BitType} \vdash x.\mathtt{value} \leftarrow \varsigma(y)1 : \mathtt{BitType}}$$

Unsoundness of Covariance

- Object types are invariant (not co/contravariant)
- Example of covariance being unsafe:
  - Let U = [] and L = [m : U]
  - By our rules L < U
  - Let P = [x : U, f : U] and Q = [x : L, f : U]
  - Assume we (mistakenly) say that Q < P (hoping for covariance in the type of x)
  - Consider the expression:
    q : Q = [x = [m = []], f = ς(s:Q) s.x.m]
  - Then q : P (by subsumption with Q < P)
  - Hence q.x ← [] : P
  - This yields the object [x = [], f = ς(s:Q) s.x.m]
  - Hence (q.x ← []).f : U yet (q.x ← []).f fails!

Covariance Would Be Nice Though

- Recall the type of bit cells

  BitType = [value : int, set : BitType, reset : BitType]
- Consider the type of flipable bit cells

  FlipBitType = [value : int, set : FlipBitType, reset : FlipBitType, flip : FlipBitType]
- We would expect that FlipBitType < BitType
- Does *not work* because object types are invariant
- We need covariance + subtyping of recursive types
  - Several ways to fix this

Variance Annotations

- Covariance fails if the method can be *updated*
  - If we never update set, reset or flip we could allow covariance
- We annotate each method in an object type with a variance:

  \+ means read-only. Method invocation but not update

  \- means write-only. Method update but not invocation

  0 means read-write. Allows both update and invocation
- We must change the typing rules to check annotations
- And we can relax the subtyping rules

Subtyping with Variance Annotations

- Invariant subtyping (Read-Write)

  [… m_i^0 : B …] < [… m_i^0 : B' …] if B = B'

- Covariant subtyping (Read-only)

  [… m_i^+ : B …] < [… m_i^+ : B' …] if B < B'

- Contravariant subtyping (Write-only)

  [… m_i^- : B …] < [… m_i^- : B' …] if B' < B

- In some languages these annotations are implicit
  - e.g., only fields can be updated

Classes, Types and Variance

- Recall the type of bit cells

  BitType = [value^0 : int,
  set$^+$: BitType, reset$^+$: BitType]

- Consider the type of flipable bit cells

  FlipBitType = [value^0 : int, set$^+$: FlipBitType,
  reset$^+$: FlipBitType, flip$^+$: FlipBitType]

- Now we have FlipBitType < BitType
  - Recall the subtyping rule for recursive types

$$\frac{\begin{array}{c}\texttt{FlipBitType} < \texttt{BitType}\\ \vdots \\ \tau < \sigma\end{array}}{\mu\,\texttt{FlipBitType}.\tau < \mu\,\texttt{BitType}.\sigma}$$

#25

Classes and Types

- Let A = [m_i : B_i] be an object type
- Let Class(A) be the type of classes for objects of type A

  Class(A) = [new : A, m_i : A $\rightarrow$ B_i]
  - A class has a generator and the body for the methods
- Types are distinct from classes
  - A class is a "stamp" for creating objects
  - Many classes can create objects of the same type
  - Some languages take the view that two objects have the same type only if they are created from the same class
    - With this restriction, types are classes
  - In Java both classes and interfaces act as types

#26

Higher-Order Object Types

- We can define bounded polymorphism
- Exmaple: we want to add a method to BitType that can copy the bit value of self to another object

  lendVal = ς(z) λx:t<BitType. x.val $\leftarrow$ z.val
  - Can be applied to a BitType or a subtype

    lendVal : $\forall$t < BitType. t $\rightarrow$ t
  - Returns something of the same type as the input
  - Can infer that "z.lendVal y : FlipBitType" if "y : FlipBitType"
- We can add bounded existential types
  - Ex: abstract type with interface "make" and "and"

    Bits = $\exists$t < BitType. {make : nat $\rightarrow$ t, and : t $\rightarrow$ t $\rightarrow$ t}
  - We only know the representation type t < BitType

#27

Conclusions

- Object calculi are both simple and expressive
- Simple: just method update and method invocation
- Functions vs. objects
  - Functions can be translated into objects
  - Objects can also be translated into functions
    - But we need sophisticated type systems
    - A complicated translation
- Classes vs. objects
  - Class-based features can be encoded with objects: subclassing, inheritance, overriding

#28

Homework

- Good luck with your project presentations!
- Have a lovely summer.