Modeling and Understanding Object-Oriented Programming

Official Survey
- Please fill out the Toolkit course survey
- 40142 CS 655-1
- Apr-21-2006 Midnight → May-04-2006 9am
  - Why not do it this evening?

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:

- Variables
  - $a, b ::= x$

- Object constructor
  - $[m_i = \zeta(x) \ b_i]$ - $\zeta$ is a variant of Greek letter $\sigma$

- Method invocation
  - $a.m$ - no arguments (just the self)

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

- This is called the untyped $\zeta$-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

- 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 $\lambda$/IMP (objects are primary)

  - $b = \{ \text{value} = \zeta(x). 0, \text{set} = \zeta(x). x.\text{value} \leftarrow \zeta(y). 1, \text{reset} = \zeta(x). x.\text{value} \leftarrow \zeta(y). 0 \}$

Operational Semantics

- $a \rightarrow b$ means that $a$ reduces in one step to $b$
- The rules are: (let o be the object $[m_i = \zeta(x). b_i]$ )
  
  - $o.m_i \rightarrow [o/x] b_i$
  - $o.m_k \leftarrow \zeta(y). b \rightarrow [m_k = \zeta(y). b_i, m_i = \zeta(x). b_j]$
    
  - 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 $\lambda$-calculus? Hmm.

  - Encoding fields
    - Fields are methods that do not use self
    - Field access "o.f" is translated directly
    - Field update "o.f \leftarrow e" is translated to "o.f \leftarrow \zeta(x) e"
    - We will drop the $\zeta(x)$ from field definitions and updates

As Expressive As $\lambda$

- 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 $\lambda$-calculus expressions
    
    \[
    x = x.\text{arg} \quad \text{(read the actual argument)}
    e_1, e_2 = (e_1.\text{arg} \leftarrow \zeta(y) \ e_2).\text{val}
    \]

    - The initial value of the argument is undefined
    - From now on we use $\lambda$ notation in addition to $\zeta$
**λ-calculus into ζ-calculus**

- Consider the conversion of $(\lambda x.x) \, 5$
  
  Let $o = [\text{arg} = \zeta(y) \, z, \text{val} = \zeta(x) \, x, \text{arg}]$
  
  $(\lambda x.x) \, 5 = (o.\text{arg} \leftarrow \zeta(y) \, 5).\text{val}$

- Consider now the evaluation of this latter ζ-term

  Let $o' = [\text{arg} = \zeta(y) \, 5, \text{val} = \zeta(x) \, x, \text{arg}]$

  $(o.\text{arg} \leftarrow \zeta(y) \, 5).\text{val} \rightarrow o'.\text{val} = [\text{arg} = \zeta(y) \, 5, \text{val} = \zeta(x) \, x, \text{arg}]$.\text{val} \rightarrow x.\text{arg}[o'/x] = o'.\text{arg} \rightarrow 5[\text{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 $o = [m_1 = \zeta(x) \, b_1]$
    
    $c = [\text{new} = \zeta(z) \,[m_1 = \zeta(x) \, \text{z.m.x}]$, $m_1 = \zeta(\text{self}) \, \lambda x. \, b_1]$

    - 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
  
  $\text{BitClass} = [\text{new} = \zeta(z). \,[\text{val} = \zeta(x) \, 0, \text{set} = \zeta(x) \, \text{z.set.x}, \text{reset} = \zeta(x) \, \text{z.reset.x}]]$
  
  $\text{set} = \zeta(z) \, \lambda x. \, x.\text{val} \leftarrow \zeta(y) \, 1$
  
  $\text{reset} = \zeta(z) \, \lambda x. \, x.\text{val} \leftarrow \zeta(y) \, 0$

- **Example:**
  
  $\text{BitClass}.\text{new} \rightarrow [\text{val} = \zeta(x) \, 0, \text{set} = \zeta(x) \, \text{BitClass.set.x}, \text{reset} = \zeta(x) \, \text{BitClass.reset.x}]$

  - 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

  $\text{FlipBitClass} = [\text{new} = \zeta(z). \,[\text{flip} = \zeta(z) \, \text{z.flip.x}, \text{flip} = \zeta(z) \, \lambda x. \, x.\text{val} \leftarrow \not(x.\text{val})]$

- **Example:**

  $\text{FlipBitClass}.\text{new} \rightarrow [\text{val} = \zeta(x) \, 0, \text{set} = \zeta(x) \, \text{BitClass.set.x}, \text{reset} = \zeta(x) \, \text{BitClass.reset.x}, \text{flip} = \zeta(x) \, \text{FlipBitClass.flip.x}]$

  - We can model method overriding in a similar way

**Object Types**

- The previous calculus was **untyped**

- Can write invocations of nonexistent methods

  $[\text{foo} = \zeta(x) \, \ldots].\text{bogus}$

- 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:

    $A ::= [m_1, m_2, \ldots, m_n]$

    - Not good enough:
      
      If $o : [m, \ldots]$ then we still don’t know if $o.\text{m.m}$ is safe
      
      - We also need the **type of the result of a method**

- Second attempt:

  $A ::= [m_1 : A_1, \ldots, m_n : A_n]$

  - 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**:

    $$A \prec B \quad B \prec C \quad A \prec C$$

    $$n \geq k \quad \left[ m_1 : A_1, \ldots, m_n : A_n \right] \prec \left[ m_1 : A_1, \ldots, m_k : A_k \right]$$

**First-Order Object Types. Subtyping**

- Second attempt:

  $A ::= [m_1 : A_1]$

  - Specify the available methods and their **result types**

    - This can be expressed using **(width) subtyping**:

    $$A \prec B \quad B \prec C \quad A \prec C$$

    $$n \geq k \quad \left[ m_1 : A_1, \ldots, m_n : A_n \right] \prec \left[ m_1 : A_1, \ldots, m_k : A_k \right]$$
Typing Rules

\[ \begin{align*}
  \Gamma, x : A & \vdash b_i : A_i \\
  \Gamma & \vdash b : A \\
  \Gamma = [ m_i = \varsigma(x : A), b_i ] & \vdash A_i \\
  \Gamma & \vdash b.m_i : A_i
\end{align*} \]

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 \( \emptyset \vdash a : A \) and a \( \rightarrow v \) then \( \emptyset \vdash v : A \)
  - Type preservation. Evaluating a well-typed expression yields a value of the same type.

Type Examples

- Consider that old `BitCell` object
  \[
  o = \{ \text{value} = \varsigma(x).0, \text{set} = \varsigma(x).x.value \leftarrow \varsigma(y).1, \text{reset} = \varsigma(x).x.value \leftarrow \varsigma(y).0 \}
  \]
  - An appropriate type for it would be
    \[
    \text{BitType} = \{ \text{value} : \text{int}, \text{set} : \text{BitType}, \text{reset} : \text{BitType} \}
    \]
  - Note that this is a recursive type
  - Consider part of the derivation that \( o : \text{BitType} \) (for set)

\[
\begin{align*}
x : \text{BitType} & \vdash \text{value} : \text{int} \\
x : \text{BitType} & \vdash \text{value} : \text{int} \\
x : \text{BitType} & \vdash \text{value} : \text{int}
\end{align*}
\]

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:
    \[
    q : Q = \{ x = [ m = [ ]], f = \varsigma(s : Q) s.x.m \}
    \]
  - Then q : P (by subsumption with Q < P)
  - Hence q.x \leftarrow [ ] : P
  - This yields the object [ x = [ ], f = \varsigma(s : Q) s.x.m ]
  - Hence (q.x \leftarrow [ ]).f : U yet (q.x \leftarrow [ ]).f fails!

Covariance Would Be Nice Though

- Recall the type of bit cells
  \[
  \text{BitType} = \{ \text{value} : \text{int}, \text{set} : \text{BitType}, \text{reset} : \text{BitType} \}
  \]
- Consider the type of flipable bit cells
  \[
  \text{FlipBitType} = \{ \text{value} : \text{int}, \text{set} : \text{FlipBitType}, \text{reset} : \text{FlipBitType}, \text{flip} : \text{FlipBitType} \}
  \]
- We would expect that \( \text{FlipBitType} < \text{BitType} \)
- *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)
  \[[... m_0^0 : B ...] < [... m_0^0 : B' ...]\] if \(B = B'\)

- Covariant subtyping (Read-only)
  \[[... m^+ : B ...] < [... m^+ : B' ...]\] if \(B < B'\)

- Contravariant subtyping (Write-only)
  \[[... m^- : B ...] < [... m^- : B' ...]\] 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
  \(\text{BitType} = [\text{value}^0 : \text{int}, \text{set}^* : \text{BitType}, \text{reset}^* : \text{BitType}]\)

- Consider the type of flipable bit cells
  \(\text{FlipBitType} = [\text{value}^0 : \text{int}, \text{set}^* : \text{FlipBitType}, \text{reset}^* : \text{FlipBitType}, \text{flip}^* : \text{FlipBitType}]\)

- Now we have \(\text{FlipBitType} < \text{BitType}\)

Classes and Types

- Let \(A = [m : B, i]\) be an object type
- Let Class(A) be the type of classes for objects of type \(A\)
  \(\text{Class}(A) = [\text{new} : A, m_i : A \rightarrow B]\)
  - 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

Classical Subtyping Rule for recursive types
\[
\mu \mu \sigma.
\]

Higher-Order Object Types

- We can define bounded polymorphism
- Example: we want to add a method to BitType that can copy the bit value of self to another object
  \(\text{lendVal} = \varsigma(z) \lambda x : t < \text{BitType}. x.\text{val} \leftarrow z.\text{val}\)
  - Can be applied to a BitType or a subtype
  \(\text{lendVal} : \forall t < \text{BitType}. t \rightarrow t\)
  - Returns something of the same type as the input
  - Can infer that \(z.\text{lendVal} y : \text{FlipBitType}\) if \(y : \text{FlipBitType}\)

- We can add bounded existential types
  - Ex: abstract type with interface "make" and "and"
    \(\text{Bits} = \exists t < \text{BitType}. \{\text{make} : \text{nat} \rightarrow t, \text{and} : t \rightarrow t \rightarrow t\}\)
  - We only know the representation type \(t < \text{BitType}\)

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

Homework

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