

Featherweight Java

Lecture 12

CS 565

3/18/08

Objects vs Functions

- Will start exploring an object model that treats objects and classes as primitives
- Avoids some of the complexities faced when encoding objects in the lambda calculus (but introduces others)
- Starting point: Model Java
 - only consider "core oo" features
 - will even ignore assignment!
 - will obviously omit: concurrency, class loading, inner classes, exceptions, iterators, overloading

Featherweight Java (FJ)

- What's left:
 - classes and objects
 - methods and invocation
 - fields and accesses
 - inheritance (open recursion)
 - casts
- Similar goals to λ -calculus in this sense.

Example

```
class A extends Object { A() { super ();} }
```

```
class B extends Object { B() { super();} }
```

```
class Pair extends Object {
```

```
    Object fst;
```

```
    Object snd;
```

```
    Pair(Object fst, Object snd) { super(); this.fst = fst; this.snd = snd; }
```

```
    Pair setFst(Object newFst) { return new Pair(newFst, this.snd); }
```

```
}
```

Another example

```
class Point extends Object {  
    int x; int y;  
    Point(int x, int y){super(); this.x:=x; this.y:=y;}  
    int getX() {return self.x;}  
    int getY() {return self.y;}  
}
```

```
class ColorPoint extends Point {  
    Color c;  
    ColorPoint(int x, int y, color c){ super(x,y); this.c := c;}  
    Color getc() {return self.c;}  
}
```

Conventions

- Always include superclass (even when it's Object)
- Always write out constructor (even when it's trivial)
- Always explicitly name receiver object in method invocation (even when it is this).
- Every method consists of a single return expression
- Constructors always take:
 - same number (and types) of parameters as class fields.
 - Constructor parameters assigned to local fields
 - Super constructor is called to assign remaining fields
 - Have no other computation.

Formalizing FJ

- First, distinguish between two kinds of type systems:
 - Nominal type systems:
 - types are always named.
 - typechecker validates types based on its name, not on its structure
 - subtyping declared explicitly by the programmer
 - Structural type systems:
 - the structure of the object determines its type, not its name
 - Names are merely convenient abbreviations.
- What kind of system does Java employ? ML?
- What are the advantages/disadvantages of these two approaches?
 - type tags for runtime manipulation of types
 - ease of typechecking
 - dealing with recursive types
 - simplicity of presentation; extensibility
 - provability

Object representation

- Key simplification: eliminating assignment
 - objects can differ only via:
 - their classes
 - the parameters passed to the constructor when they were created
 - all necessary information is available in the form: `new C(v)` which is the only value

Syntax

$t ::=$ x variable
 | $t.f$ field access
 | $t.m(t\dots t)$ method invocation
 | $\text{new } C(t\dots t)$ object creation
 | $(C) t$ cast

$v ::=$ $\text{new } C(v\dots v)$ value

Methods and classes

$K ::= C(C\ f) \{ \text{super}(f); \text{this.f} = f \}$
constructor declarations

$M ::= C\ m(C\ x) \{ \text{return } t; \}$
method declarations

$CL ::= \text{class } C \text{ extends } C \{ C\ f; K\ M \}$
class declarations

Auxiliary Operations: Field Lookup

$$\text{fields}(\text{Object}) = \phi$$

$$CT(C) = \text{class } C \text{ extends } D \{ C \ f; K \ M \}$$

$$\text{fields}(D) = D \ g$$

$$\text{fields}(C) = D \ g; C \ f$$

Auxiliary Operations: Method Type Lookup

$CT(C) = \text{class } C \text{ extends } D \{C \text{ f; } K \ M\}$

$B \ m \ (B \ x) \ \{\text{return } t;\} \in M$

$\text{mtype}(m, C) = B \rightarrow B$

$CT(C) = \text{class } C \text{ extends } D \{C \text{ f; } K \ M\}$

$m \notin M$

$\text{mtype}(m, C) = \text{mtype}(m, D)$

Auxiliary Operations:

Method Body Lookup and Override

$$\begin{array}{l} CT(C) = \text{class } C \text{ extends } D \{ \mathbf{C} \text{ f}; \mathbf{K} \mathbf{M} \} \\ \quad \mathbf{B} \text{ m } (\mathbf{B} \text{ x}) \{ \text{return t;} \} \in \mathbf{M} \\ \hline \text{mbody}(m, C) = (\mathbf{x}, \mathbf{t}) \end{array}$$

$$\begin{array}{l} CT(C) = \text{class } C \text{ extends } D \{ \mathbf{C} \text{ f}; \mathbf{K} \mathbf{M} \} \\ \quad \mathbf{m} \notin \mathbf{M} \\ \hline \text{mbody}(m, C) = \text{mbody}(m, D) \end{array}$$

$$\begin{array}{l} \text{mtype}(m, D) = \mathbf{D} \rightarrow \mathbf{D}_0 \Rightarrow \mathbf{C} = \mathbf{D} \text{ and } \mathbf{C}_0 = \mathbf{D}_0 \\ \hline \text{override}(m, D, \mathbf{C} \rightarrow \mathbf{C}_0) \end{array}$$

Subtyping

- As in Java, subtyping in FJ is declared.

$$C \ll C$$
$$\frac{C \ll D \quad D \ll E}{C \ll E}$$
$$\frac{CT(C) = \text{class extends } D \{ \dots \}}{C \ll D}$$

The class table is assumed to be a global (fixed) table that maps class names to definitions.

Term Typing

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}$$

$$\frac{\begin{array}{l} \Gamma \vdash t_0 : C_0 \\ \text{fields}(C_0) = C \mathbf{f} \end{array}}{\Gamma \vdash t_0.f_i : C_i}$$

Term Typing

$$\Gamma \vdash t_0 : C_0$$
$$\text{mtype}(m, C_0) = D \rightarrow C$$
$$\Gamma \vdash t : C \quad C \leqslant D \text{ (built-in subsumption)}$$

$$\Gamma \vdash t_0.m(t) : C$$
$$\text{fields}(C) = D f$$
$$\Gamma \vdash t : C \quad C \leqslant D$$

$$\Gamma \vdash \text{new } C(t) : C$$

Casts

$$\frac{\Gamma \vdash t_0 : D \quad D \leqslant C}{\Gamma \vdash (C)t_0 : C} \quad \text{(upcast)}$$

$$\frac{\Gamma \vdash t_0 : D \quad C \leqslant D \quad C \neq D}{\Gamma \vdash (C)t_0 : C} \quad \text{(downcast)}$$

$$\frac{\Gamma \vdash t_0 : D \quad C \not\leqslant D \quad D \not\leqslant C}{\Gamma \vdash (C)t_0 : C} \quad \text{warning}$$

consider $(A) \text{ (Object) new } B() \rightarrow (A) \text{ new } B()$

Method and Class Typing

$x: C, \text{this}: C \vdash t_0 : E_0 \quad E_0 \leq C_0$

$CT(C) = \text{class } C \text{ extends } D \{ \dots \}$

$\text{override}(m, D, C \vdash C_0)$

$C_0 \ m \ (C \ x) \{ \text{return } t_0; \} \text{ OK in } C$

$K = C(D \ g, C \ f)$

$\{ \text{super}(g; \text{this}.f = f; \}$

$\text{fields}(D) = D \ g \quad M \text{ OK in } C$

$\text{class } C \text{ extends } D \{ C \ f; K \ M \} \text{ OK}$

Typing: class declaration

When is a class declaration well formed?

Provided the constructor, K , has the form above:
the fields are split as $D g$ (i.e., fields of super classes right upto `Object`) and $C f$ (i.e., fields declared in the current class). Next, the constructor body begins by initializations of the super class fields. Finally, the fields declared in the current class (C) are initialized.

Provided method declarations, M , in class C are well-formed.

Typing: class table and program

$\forall C \in \text{dom}(CT), CT(C) \text{ ok}$

$CT \text{ ok}$

When is a class table CT well-formed?

Provided every class declaration, $CT(C)$, in CT is well-formed.

A program (CT, t) is well-formed iff CT is well-formed and $\vdash t : C$.

Evaluation Rules

$$\frac{\text{fields}(C) = \mathbf{C} \mathbf{f}}{(\text{new } C(\mathbf{v})).f_i \rightarrow v_i}$$

$$\frac{\text{mbody}(m, C) = (\mathbf{x}, t_0)}{(\text{new } C(\mathbf{v})).m(\mathbf{u}) \rightarrow [\mathbf{x} \mapsto \mathbf{u}, \text{this} \mapsto \text{new } C(\mathbf{v})]t_0}$$

$$\frac{C \ll: D}{(D) (\text{new } C(\mathbf{v})) \rightarrow \text{new } C(\mathbf{v})}$$

Congruence Rules

$$\frac{t_0 \rightarrow t'_0}{t_0.f \rightarrow t'_0.f}$$

$$\frac{t_0 \rightarrow t'_0}{t_0.m(\mathbf{t}) \rightarrow t'_0.m(\mathbf{t})}$$

$$\frac{t_i \rightarrow t'_i}{v_0.m(\mathbf{v}, t_i, \mathbf{t}) \rightarrow v_0.m(\mathbf{v}, t'_i, \mathbf{t})}$$

$$\frac{t_i \rightarrow t'_i}{\text{new } C(\mathbf{v}, t_i, \mathbf{t}) \rightarrow \text{new } C(\mathbf{v}, t'_i, \mathbf{t})}$$

$$\frac{t_0 \rightarrow t'_0}{(C) t_0 \rightarrow (C) t'_0}$$

Example Revisited

```
class A extends Object { A() { super ();} }
```

```
class B extends Object { B() { super();} }
```

```
class Pair extends Object {
```

```
    Object fst;
```

```
    Object snd;
```

```
    Pair(Object fst, Object snd) { super(); this.fst = fst; this.snd = snd; }
```

```
    Pair setFst(Object newFst) { return new Pair(newFst, this.snd); }
```

```
}
```

Evaluation

- Projection:

`new Pair(new A(), new B()).snd → new B()`

- Casting:

`((Pair)new Pair(new Pair(new A(), new B()), new A()).fst).snd
→ new B()`

Why is the cast necessary?

Method Invocation

`new Pair(new A(), new B()).setfst(new B())`

`newfst` \mapsto `new B()`,

\rightarrow `this` \mapsto `new Pair(new A(), new B())`

`new Pair(newfst, this.snd)`

`new Pair(new B(), new Pair(new A(), new B()).snd)`

Properties

Preservation:

If $\Gamma \vdash t : C$ and $t \rightarrow t'$ then $\Gamma \vdash t' : C'$ for $C' \prec C$.

Non-trivial. (see Exercise 19.5.1)

Given rules without "stupid casts", well-typed programs can type to ill-typed ones.

Stupid Casts

consider $(A) (\text{Object}) \text{ new } B() \rightarrow (A) \text{ new } B()$

$\Gamma \vdash t_0 : D \quad C \not\prec D \quad D \not\prec C \quad (\text{stupid cast})$

warning

$\Gamma \vdash (C)t_0 : C$

Theorems

Progress: Assume that CT is a well-formed class table. If $\Gamma \vdash t : C$ then either:

- (1) t is a value
- (2) t contains an expression of the form $(D)\text{new } C(v)$ where C is not $\prec : D$
- (3) there exists t' such that $t \rightarrow t'$.

Preservation: Assume that CT is a well-formed class table. If $\Gamma \vdash t : C$ and $t \rightarrow t'$ then $\Gamma \vdash t' : C'$ for some $C' \prec : C$.

Correspondence with Java

- Every syntactically well-formed FJ program corresponds to a syntactically well-formed Java program.
- A syntactically well-formed FJ program is typable in FJ (without using stupid casts) iff it is typable in Java.
- A well-typed FJ program behaves the same in FJ as in Java (e.g, a divergent FJ program will also diverge in Java)