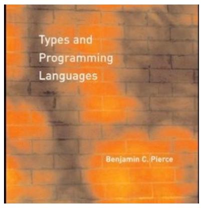
Featherweight Java

Lecture 12
CS 565



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

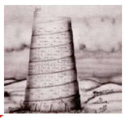
Featherweight Java (FJ)

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 this.x;}
  int gety() {return this.y;}
}

class ColorPoint extends Point {
  Color c;
  ColorPoint(int x, int y, color c){ super(x,y); this.c :=
   c;}
  Color getc() {return this.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 name, not on 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 are the (dis)advantages of these two approaches?

- type tags for runtime manipulation of types
- ease of typechecking
- dealing with recursive types; simplicity of presentation; extensibility; provability

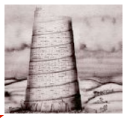
Object representation

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...t)       method invocation
     | new C(t...t)     object creation
     | (C) t            cast

v::=  new C(v...v)      value
```

Methods and classes

K ::= C(**C f**) {super(**f**); this.**f** = **f**}
constructor declarations

M ::= C m(**C x**) { return t;}
method declarations

CL::= class C extends C {**C f**; K **M**}
class declarations

Auxiliary Operations:Field Lookup

$$\texttt{fields(Object) = \{\}}$$

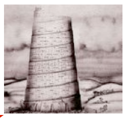
$$\frac{\texttt{CT(C) = class C extends D \{}\mathbf{C\ f;}\texttt{ K }\mathbf{M}\texttt{\}} \qquad \texttt{fields(D) = }\mathbf{D\ g}}{\texttt{fields(C) = }\mathbf{D\ g;\ C\ f}}$$

Auxiliary Operations:

$$\frac{\texttt{CT(C) = class C extends D \{}\mathbf{C\ f;}\texttt{ K }\mathbf{M}\texttt{\}} \qquad \texttt{B m (}\mathbf{B\ x}\texttt{) \{return t;\}} \in \mathbf{M}}{\texttt{mtype(m,C) = }\mathbf{B} \rightarrow \texttt{B}}$$

$$\frac{\texttt{CT(C) = class C extends D \{}\mathbf{C\ f;}\texttt{ K }\mathbf{M}\texttt{\}} \qquad \texttt{m} \notin \mathbf{M}}{\texttt{mtype(m,C) = mtype(m,D)}}$$

Auxiliary Operations:



$$\frac{\texttt{CT(C) = class C extends D \{}\mathbf{C\ f}\texttt{; K }\mathbf{M}\texttt{\}} \qquad \texttt{B m (}\mathbf{B\ x}\texttt{) \{return t;\} 2 }\mathbf{M}}{\texttt{mbody(m,C) = (}\mathbf{x}\texttt{,t)}}$$

$$\frac{\texttt{CT(C) = class C extends D \{}\mathbf{C\ f}\texttt{; K }\mathbf{M}\texttt{\}} \qquad \texttt{m} \notin \mathbf{M}}{\texttt{mbody(m,C) = mbody(m,D)}}$$

$$\frac{\texttt{mtype(m,D) = }\mathbf{D} \rightarrow \texttt{D}_0 \qquad \mathbf{C} = \mathbf{D} \text{ and } \mathbf{C}_0 = \mathbf{D}_0}{\texttt{override(m,D,}\mathbf{C} \rightarrow \mathbf{C}_0\texttt{)}}$$

Subtyping

As in Java, subtyping in FJ is declared.

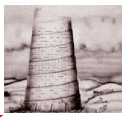
$$\texttt{C <: C}$$

$$\frac{\texttt{C <: D} \qquad \texttt{D <: E}}{\texttt{C <: E}}$$

$$\frac{\texttt{CT(C)=class extends D \{ ....\}}}{\texttt{C <: D}}$$

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

Term Typing



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

$$\frac{\Gamma \vdash \texttt{t}_0 \texttt{ : C}_0 \qquad \texttt{fields(C}_0\texttt{) = }\mathbf{C}\ \mathbf{f}}{\Gamma \vdash \texttt{t}_0\texttt{.f}_\texttt{i} \texttt{ : }\mathbf{C}_\texttt{i}}$$

Term Typing

$$\frac{\Gamma \vdash \texttt{t}_0 \texttt{ : C}_0 \qquad \texttt{mtype(m,C}_0\texttt{) = }\mathbf{D} \rightarrow \texttt{C} \qquad \Gamma \vdash \mathbf{t{:}C} \quad \mathbf{C{<:}D}}{\Gamma \vdash \texttt{t}_0\texttt{.m(}\mathbf{t}\texttt{) : C}}$$

(built-in subsumption)

$$\frac{\texttt{fields(C) = }\mathbf{D}\ \mathbf{f} \qquad \Gamma \vdash \mathbf{t{:}C} \quad \mathbf{C{<:}D}}{\Gamma \vdash \texttt{new C(}\mathbf{t}\texttt{) : C}}$$

Casts

$$\frac{\Gamma \vdash t_0 : D \qquad D <: C}{\Gamma \vdash (C)\,t_0 : C} \qquad \text{(upcast)}$$

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

$$\text{warning} \qquad \frac{\Gamma \vdash t_0 : D \quad \neg(C <: D) \quad \neg(D <: C)}{\Gamma \vdash (C)\; t_0 : C}$$

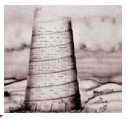
consider `(A) (Object) new B()` $\Rightarrow$ `(A) new B()`

Method and Class Typing

$$\frac{\begin{array}{l}\mathbf{x}:\mathbf{C},\texttt{this}:C \vdash t_0 : E_0 \quad E_0 <: C_0 \\ CT(C) = \texttt{class } C \texttt{ extends } D \;\{\ldots\} \\ \texttt{override}(m,\ D,\ \mathbf{C} \rightarrow C_0)\end{array}}{C_0\ m\ (\mathbf{C}\ \mathbf{x})\ \{\ \texttt{return}\ t_0;\}\ \text{OK in } C}$$

$$\frac{\begin{array}{c}K = C(\mathbf{D}\ \mathbf{g},\ \mathbf{C}\ \mathbf{f}) \\ \{\texttt{super}(\mathbf{g});\ \texttt{this}.\mathbf{f} = \mathbf{f};\} \\ \texttt{fields}(D) = \mathbf{D}\ \mathbf{g} \qquad \mathbf{M}\ \text{OK in } C\end{array}}{\texttt{class } C \texttt{ extends } D\ \{\ \mathbf{C}\ \mathbf{f};\ K\ \mathbf{M}\}\ \text{OK}}$$

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.

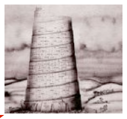
Typing: class table and program

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



$$\frac{\mathtt{fields(C)} = \mathbf{C}\ \mathbf{f}}{\mathtt{(new\ C(}\mathbf{v}\mathtt{)).f}_i \longrightarrow \mathtt{v}_i}$$

$$\frac{\mathtt{mbody(m,C)} = \mathtt{(}\mathbf{x}\mathtt{,t}_0\mathtt{)}}{\mathtt{(new\ C(}\mathbf{v}\mathtt{)).m(}\mathbf{u}\mathtt{)} \longrightarrow \mathtt{[}\mathbf{x}\mathtt{/}\mathbf{u}\mathtt{,\ this/new\ C(}\mathbf{v}\mathtt{)]t}_0}$$

$$\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{(D)\ (new\ C(}\mathbf{v}\mathtt{))} \longrightarrow \mathtt{new\ C(}\mathbf{v}\mathtt{)}}$$

Congruence Rules

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\mathtt{t}_0\mathtt{.f} \longrightarrow \mathtt{t}_0'\mathtt{.f}}$$

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\mathtt{t}_0\mathtt{.m(}\mathbf{t}\mathtt{)} \longrightarrow \mathtt{t}_0'\mathtt{.m(}\mathbf{t}\mathtt{)}}$$

$$\frac{\mathtt{t}_i \longrightarrow \mathtt{t}_i'}{\mathtt{v}_0\mathtt{.m(}\mathbf{v}\mathtt{,t}_i\mathtt{,}\mathbf{t}\mathtt{)} \longrightarrow \mathtt{v}_0\mathtt{.m(}\mathbf{v}\mathtt{,t}_i'\mathtt{,}\mathbf{t}\mathtt{)}}$$

$$\frac{\mathtt{t}_i \longrightarrow \mathtt{t}_i'}{\mathtt{new\ C(}\mathbf{v}\mathtt{,t}_i\mathtt{,}\mathbf{t}\mathtt{)} \longrightarrow \mathtt{new\ C(}\mathbf{v}\mathtt{,t}_i'\mathtt{,}\mathbf{t}\mathtt{)}}$$

$$\frac{\mathtt{t}_0 \longrightarrow \mathtt{t}_0'}{\mathtt{(C)\ t}_0\ \mathbf{!}\ \mathtt{(C)\ t}_0'}$$

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` $\longrightarrow$ `new B()`

Casting:

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

$\longrightarrow$ `new B()`

Why is the cast necessary?

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)new C(`$\mathbf{v}$`)` where `C` is not `<: 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′ <: 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)