Programming Language Theory

Featherweight Java (FJ)

Ralf Lämmel

This lecture is based on David Walker’s lecture: Computer Science 441, Programming Languages, Princeton University
Motivation

• What is the “essence” of OO?
  ✦ What expressiveness to include?
    ★ Classes, interfaces?
    ★ Static, instance fields?
    ★ ...

• What calculus to use as the foundation?
  ✦ Lambda calculus?
Overview

• Featherweight Java (FJ):
  • a minimal Java-like language;
  • models inheritance and subtyping;
  • immutable objects: no mutation of fields;
  • trivialized core language.
FJ -- Abstract syntax
Abstract Syntax

Classes in FJ have the form:

\[
\text{class } c \text{ extends } c' \{ c_f; k\ d\}
\]

- Class \( c \) is a sub-class of class \( c' \).
- Constructor \( k \) for instances of \( c \).
- Fields \( c_f \).
- Methods \( d \).

(underlining indicates “one or more”.)
Abstract Syntax

Constructor expressions have the form

\[ c(c'x', cx) \{ \text{super}(x'); \text{this.f}=x; \} \]

- Arguments correspond to super-class fields and sub-class fields.
- Initializes super-class.
- Initializes sub-class.
Abstract Syntax

Methods have the form

\[ cm(cx) \{ \text{return } e; \} \]

- Result class \( c \).
- Argument class(es) \( c \).
- Binds \( x \) and this in \( e \).
Abstract Syntax

Minimal set of expressions:

- Field selection: \( e.f \).
- Message send: \( e.m(e) \).
- Instantiation: \( \text{new } c(e) \).
- Cast: \( (c) e \).
class Pt extends Object {
    int x;
    int y;
    Pt (int x, int y) {
        super(); this.x = x; this.y = y;
    }
    int getx () { return this.x; }
    int gety () { return this.y; }
}
FJ Example

class CPt extends Pt {
    color c;
    CPt (int x, int y, color c) {
        super(x,y);
        this.c = c;
    }
    color getc () { return this.c; }
}
Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

- **Classes**  \( C ::= \text{class} \; c \; \text{extends} \; c' \{ c_f; k d \} \)
- **Constructors**  \( k ::= c(c\;x) \{ \text{super}(x); \; \text{this}.f=x; \} \)
- **Methods**  \( d ::= c\;m(c\;x) \{ \text{return} \; e; \} \)
- **Types**  \( \tau ::= c \)
- **Expressions**  \( e ::= x \mid e.f \mid e.m(e) \)
  \(\mid \text{new} \; c(e) \mid (c) \; e \)

Underlining indicates “one or more”.

If \( e \) appears in an inference rule and \( e_i \) does too, there is an implicit understanding that \( e_i \) is one of the \( e \)’s in \( e \). And similarly with other underlined constructs.
Class Tables and Programs

A **class table** $T$ is a finite function assigning classes to class names.

A **program** is a pair $(T, e)$ consisting of

- A class table $T$.
- An expression $e$. 
FJ -- Static semantics
Static Semantics

Judgement forms:

\[ \tau <: \tau' \] \hspace{1cm} \text{subtyping}

\[ c \subseteq c' \] \hspace{1cm} \text{subclassing}

\[ \Gamma \vdash e : \tau \] \hspace{1cm} \text{expression typing}

\[ d \text{ ok in } c \] \hspace{1cm} \text{well-formed method}

\[ c \text{ ok} \] \hspace{1cm} \text{well-formed class}

\[ T \text{ ok} \] \hspace{1cm} \text{well-formed class table}

\[ \text{fields}(c) = c.f \] \hspace{1cm} \text{field lookup}

\[ \text{type}(m, c) = c \rightarrow c \] \hspace{1cm} \text{method type}
Static Semantics

Variables:

\[ \Gamma(x) = \tau \]
\[ \Gamma \vdash x : \tau \]

- Must be declared, as usual.
- Introduced within method bodies.
Static Semantics

Field selection:

\[
\Gamma \vdash e_0 : c_0 \quad \text{fields}(c_0) = \underbrace{c_f} \\
\Gamma \vdash e_0 \cdot f_i : c_i
\]

- Field must be present.
- Type is specified in the class.
Static Semantics

Message send:

\[
\begin{align*}
\Gamma \vdash e_0 : c_0 & \quad \Gamma \vdash e : c \\
\text{type}(m, c_0) = c' \rightarrow c & \quad c < : c'
\end{align*}
\]

\[
\Gamma \vdash e_0.m(e) : c
\]

- Method must be present.
- Argument types must be subtypes of parameters.
Static Semantics

Instantiation:

\[ \Gamma \vdash e : c'' \quad c'' <: c' \quad \text{fields}(c) = c' f \]
\[ \Gamma \vdash \text{new } c(e) : c \]

- Initializers must have subtypes of fields.
Static Semantics

Casting:

\[
\Gamma \vdash e_0 : d \\
\Gamma \vdash (c)\ e_0 : c
\]

- **All** casts are statically acceptable!

- Could try to detect casts that are guaranteed to fail at run-time.
Subclassing

Sub-class relation is implicitly relative to a class table.

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \ldots \} \]

\[ c \sqsubseteq c' \]

Reflexivity, transitivity of sub-classing:

\[ (T(c) \text{ defined}) \quad \begin{align*}
& \quad c \sqsubseteq c' \quad c' \sqsubseteq c'' \\
& \quad c \sqsubseteq c''
\end{align*} \]

Sub-classing only by explicit declaration!
Subtyping

Subtyping relation: $\tau <: \tau'$. 

$\tau <: \tau$  $\frac{\tau <: \tau'}{\tau <: \tau''}$  $\frac{\tau' <: \tau''}{\tau <: \tau''}$

$c \triangleleft c'$  $\frac{c <: c'}{c <: c'}$

Subtyping is determined **solely** by subclassing.
Class Formation

Well-formed classes:

\[
k = c(c' x', c x) \{ \text{super}(x'); \text{this}.f=x; \}
\]

fields(c') = c' f' d_i ok in c

\[
\text{class } c \text{ extends } c' \{ c f; k d \} \text{ ok}
\]

- Constructor has arguments for each super- and sub-class field.

- Constructor initializes super-class before sub-class.

- Sub-class methods must be well-formed relative to the super-class.
Class Formation

Method overriding, relative to a class:

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \ldots \} \]
\[ \text{type}(m, c') = c \rightarrow c_0 \quad x : c, \text{this} : c \vdash e_0 : c'_0 \quad c'_0 <: c_0 \]
\[ c_0 \ m(c.x) \{ \text{return } e_0; \} \ \text{ok in } c \]

- Sub-class method must return a subtype of the super-class method's result type.

- Argument types of the sub-class method must be exactly the same as those for the super-class.

- Need another case to cover method extension.
Program Formation

A class table is well-formed iff all of its classes are well-formed:

\[ \forall c \in \text{dom}(T) \quad T(c) \text{ ok} \]

\[ \frac{}{T \text{ ok}} \]

A program is well-formed iff its class table is well-formed and the expression is well-formed:

\[ T \text{ ok} \quad \emptyset \vdash e : \tau \]

\[ \frac{}{(T, e) \text{ ok}} \]
Method Typing

The type of a method is defined as follows:

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]
\[
d_i = c_i \ m(c_i x) \{ \text{return } e; \}
\]
\[
\text{type}(m_i, c) = c_i \rightarrow c_i
\]

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \}
\]
\[
m \notin d \quad \text{type}(m_i, c') = c_i \rightarrow c_i
\]
\[
\text{type}(m, c) = c_i \rightarrow c_i
\]
FJ -- Dynamic semantics
Dynamic Semantics

Transitions: $e \xrightarrow{T} e'$.

Transitions are indexed by a (well-formed) class table!

- Dynamic dispatch.
- Downcasting.

We omit explicit mention of $T$ in what follows.
Dynamic Semantics

Object values have the form

$$\text{new } c(e', e)$$

where

- $e'$ are the values of the super-class fields.
- $e$ are the values of the sub-class fields.
- $c$ indicates the “true” (dynamic) class of the instance.

Use this judgement to affirm an expression is a value:

$$\text{new } c(e', e) \text{ value}$$

Rules

$$\dfrac{\text{new Object value} \quad e_i \text{ value} \quad e_i \text{ value}}{\text{new } c(e', e) \text{ value}}$$
Dynamic Semantics

Field selection:

\[
\begin{align*}
\text{fields}(c) &= c' f', c f & e' \text{ value} & e \text{ value} \\
\text{new } c(e', e) . f'_i & \rightarrow e'_i \\
\end{align*}
\]

\[
\begin{align*}
\text{fields}(c) &= c' f', c f & e' \text{ value} & e \text{ value} \\
\text{new } c(e', e) . f_i & \rightarrow e_i \\
\end{align*}
\]

- Fields in sub-class must be disjoint from those in super-class.
- Selects appropriate field based on name.
Dynamic Semantics

Message send:

\[
\begin{align*}
\text{body}(m, c) &= x \to e_0 \quad e \text{ value} \quad e' \text{ value} \\
\text{new } c(e) \cdot m(e') &\mapsto \{e'/x\}\{\text{new } c(e)/\text{this}\}e_0
\end{align*}
\]

- The identifier `this` stands for the object itself.
Dynamic Semantics

Cast:

\[ \frac{c \trianglelefteq c' \quad e \text{ value}}{(c') \text{ new } c(e) \leftrightarrow \text{ new } c(e)} \]

- No transition (stuck) if \( c \) is not a sub-class of \( c' \)!

- Sh/could introduce error transitions for cast failure.
Dynamic Semantics

Search rules (CBV):

\[
\begin{align*}
    e_0 & \overset{\text{value}}{\mapsto} e_0' \\
    e_0 \cdot f & \overset{\text{call}}{\mapsto} e_0' \cdot f \\
    e_0 & \overset{\text{call}}{\mapsto} e_0' \\
    e_0 \cdot m(e) & \overset{\text{apply}}{\mapsto} e_0' \cdot m(e)
\end{align*}
\]
Dynamic Semantics

Search rules (CBV), cont’d:

\[
\begin{align*}
e & \mapsto e' \\
\text{new } c(e) & \mapsto \text{new } c(e') \\
e_0 & \mapsto e'_0 \\
(c) e_0 & \mapsto (c) e'_0
\end{align*}
\]
Dynamic Semantics

Dynamic dispatch:

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \} \]
\[ d_i = c_i m(c_i x) \{ \text{return } e; \} \]
\[ \text{body}(m_i, c) = x \rightarrow e \]

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots; \ldots d \} \]
\[ m \notin d \quad \text{body}(m, c') = x \rightarrow e \]
\[ \text{body}(m, c) = x \rightarrow e \]

- Climbs the class hierarchy searching for the method.
- Static semantics ensures that the method must exist!
Type safety = Preservation + Progress
Type Safety

Theorem 1 (Preservation)
Assume that $T$ is a well-formed class table. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau'$ for some $\tau' <: \tau$.

- Proved by induction on transition relation.

- Type may get “smaller” during execution due to casting!
Type Safety

Lemma 2 (Canonical Forms)

If $e : c$ and $e$ value, then $e \equiv \text{new } d(e_0)$ with $d \triangleleft c$ and $e_0$ value.

- Values of class type are objects (instances).

- The **dynamic** class of an object may be lower in the subtype hierarchy than the **static** class.
Type Safety

Theorem 3 (Progress)
Assume that $T$ is a well-formed class table. If $e : \tau$ then either

1. $e$ value, or

2. $e$ has the form $(c)\text{new } d(e_0)$ with $e_0$ value and $d \not\subseteq c$, or

3. there exists $e'$ such that $e \rightarrow e'$. 
Type Safety

Comments on the progress theorem:

- Well-typed programs can get stuck! But only because of a cast . . . .

- Precludes “message not understood” error.

- Proof is by induction on typing.
• **Summary**: The essence of OO
  ✦ Designated calculus (instead of lambda calculus)
  ✦ Operational semantics with type safety
  ✦ Subclasses / subtyping covered
  ✦ Immutable instead of imperative fields modeled

• **Prepping**: “Types and Programming Languages”
  ✦ Chapters 19 but also 15-18

• **Outlook**:
  ✦ Process calculi
  ✦ Denotational semantics
Variations and extensions
Variations and Extensions

A more flexible static semantics for override:

- Subclass result type is a \textbf{subtype} of the superclass result type.

- Subclass argument types are \textbf{supertypes} of the corresponding superclass argument types.
Variations and Extensions

Java adds arrays and covariant array subtyping:

\[
\tau \ll : \tau' \quad \frac{}{\tau[\ ] \ll : \tau'[\ ]}
\]

What effect does this have?
Variations and Extensions

Java adds array covariance:

\[
\frac{\tau <: \tau'}{
\tau [ ] <: \tau' [ ]}
\]

- Perfectly OK for FJ, which does not support mutation and assignment.

- With assignment, might store a supertype value in an array of the subtype. Subsequent retrieval at subtype is unsound.

- Java inserts a per-assignment run-time check and exception raise to ensure safety.
Variations and Extensions

Static fields:

- Must be initialized as part of the class definition (not by the constructor).

- In what order are initializers to be evaluated? Could require initialization to a constant.
Variations and Extensions

Static methods:

- Essentially just recursive functions.
- No overriding.
- Static dispatch to the class, not the instance.
Variations and Extensions

Final methods:

- Preclude override in a sub-class.

Final fields:

- Sensible only in the presence of mutation!
Variations and Extensions

Abstract methods:

- Some methods are undefined (but are declared).
- Cannot form an instance if any method is abstract.