Feather-Weight Java

COS 441
Princeton University
Fall 2004

Feather-Weight Java

- Core calculus defining “essence” of Java
  - Target language to explain generics
  Featherweight Java: A Minimal Core Calculus for Java and GJ by Atsushi Igarashi, Benjamin Pierce, and Philip Wadler.
- Harper’s Notes contains many small and big bugs!
- Exercise for today is bug hunting as well as understanding FJ

Abstract Syntax

```
Classes    C ::= class c extends C { f; k d }
Constructors k ::= c(x) { super(x); this.f=x; }
Methods    d ::= c m(c x) [ return e; ]
Types      τ ::= c
Expressions e ::= x | e.f | e.m(c) | new c(f) | (c) e
```

As a notational convenience we use “underbarring” to stand for sequences of phrases. For example, if d stands for a sequence of d’s, whose individual elements we designate d_1, ..., d_k, where k is the length of the sequence. We write c f for the sequence c_1 f_1, ..., c_k f_k, where k is the length of the sequences c and f. Similar conventions govern the other uses of sequence notation.

Example: Point

```
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; }
}
```

Example: Color Points

```
class Cpt extends Pt {
    color c;
    Cpt (int x, int y, color c) {
        super(x,y);
        this.c = c;
    }
    color getC () { return this.c; }
}
```

Class Tables and Programs

A class table T is a finite function assigning classes to class names. The classes declared in the class table are bound within the table so that all classes may refer to one another via the class table.

A program is a pair (T, e) consisting of a class table T and an expression e. We generally suppress explicit mention of the class table, and consider programs to be expressions.
Static Semantics

\[ \tau \triangleleft \tau' \quad \text{subtyping} \]
\[ \Gamma \vdash e : \tau \quad \text{expression typing} \]
\[ \text{ck in } c \quad \text{well-formed class} \]
\[ T \ldots \quad \text{well-formed class table} \]

fields(c) = \mathcal{f} \quad \text{field lookup}

\[ \text{type}(m, c) = \overline{c} \rightarrow c \quad \text{method type} \]

All judgments for dynamic and static semantics are implicitly parameterized by a single global class table.

---

Subtyping

\[ \tau \triangleleft \tau' \quad \tau \triangleleft \tau'' \]
\[ \Gamma \vdash e : \tau \quad \text{expression typing} \]

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

---

Fields of a Class

\[ \text{fields}(\text{Object}) = \bullet \]

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

\[ \text{fields}(c) = \mathcal{f}' \quad \mathcal{f} \]

---

Signature of a Method

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

\[ c'' = c_{1} \quad m(c_{2}, a) \{ \text{return } c_{1} \} \]

\[ \text{type}(m, c) = c_{1} \rightarrow c_{2} \]

---

Expression Typing

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

\[ \Gamma \vdash c_{0} : c_{0} \quad \text{fields}(c_{0}) = \mathcal{f} \]

\[ \Gamma \vdash c_{0}, f_{1} : c_{1} \]

\[ \Gamma \vdash c_{0} : c_{0} \quad \Gamma \vdash e : c \]

\[ \text{type}(m, c_{0}) = \mathcal{f}' \rightarrow c \quad c \triangleleft c' \]

\[ \Gamma \vdash c_{0}, m(e) : c \]

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

\[ \Gamma \vdash \text{new } c(e) : c \]
Typing for Casts

\[
\Gamma \vdash e_0 : c' \quad c' \ll c \\
\Gamma \vdash (c) \ e_0 : c
\]

Type-in Harper Should be \( c' \)

Treatment of cast in original FJ Paper

Well-Formed Method

\[
\text{Well-Formed Method} \\
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\text{type}(m, c') = \xi \rightarrow c_0 \quad \exists x, \text{this} \vdash e_0 : c_0 \\
c_0 \ m(\xi \ x) \ \{ \text{return } e_0 ; \} \ \text{ok in } c
\]

Doesn’t handle the case of extending a class with a new method only when overriding a class

Well-Formed Method

\[
\text{Well-Formed Method} \\
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\text{type}(m, c') = \xi \rightarrow c_0 \quad \exists x, \text{this} \vdash e_0 : c_0 \\
c_0 \ m(\xi \ x) \ \{ \text{return } e_0 ; \} \ \text{ok in } c
\]

Doesn’t allow subtyping for returned type

Well-Formed Method

\[
\text{Well-Formed Method} \\
Buggy \ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\text{type}(m, c') = \xi \rightarrow c_0 \quad \exists x, \text{this} \vdash e_0 : c_0 \\
c_0 \ m(\xi \ x) \ \{ \text{return } e_0 ; \} \ \text{ok in } c
\]

Fixed Rule

\[
\text{Well-Formed Method} \\
T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \\
\text{type}(m, c') = \xi \rightarrow c_0 \quad \exists x, \text{this} \vdash e_0 : c_0 \\
\text{if type}(m, c') = \text{super}(\xi) \ \\
\text{then } c = \text{super} \text{ and } c_0 = c_{\text{ret}} \\
c_0 \ m(\xi \ x) \ \{ \text{return } e_0 ; \} \ \text{ok in } c
\]

Well-Formed Class

\[
k = c(c', \xi, \chi) \ \{ \text{super}(\xi); \text{this.f=x;} \} \\
\text{fields}(c') = c' \ f' \ c'' \ \text{ok in } c
\]

Buggy
Well-Formed Class

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

\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \left\{ \ldots ; \ldots d \right\} \\
&\quad m \notin d \quad \text{type}(m, c') = x \to e \\
&\quad \text{body}(m, c) = x \to e
\end{align*}
\]

Class-Table and Programs

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

Dynamic Semantics

- Small-step semantics
- Implicitly index/parameterized by a global class table \( T \)
  - Left unspecified in the rules to avoid clutter
- Values define as

\[
\begin{align*}
\epsilon \text{ value} &\quad c_1 \text{ value} \quad \ldots \quad c_n \text{ value} \\
\text{new } c(c) \text{ value} &\quad \epsilon \text{ value}
\end{align*}
\]

Field Lookup

\[
\begin{align*}
\text{fields}(c) &= \epsilon \ f' \ c \ f' \ \epsilon \text{ value} \\
&\quad \text{new } c(c') \ f' \mapsto c'_i
\end{align*}
\]

Method Dispatch/Lookup

\[
\begin{align*}
\text{body}(m, c) &= x \to c_0 \ \epsilon \text{ value} \\
&\quad \epsilon \text{ value} \\
&\quad \text{new } c(c) . m(c') \mapsto \{ \epsilon / x \} \{ \text{new } c(c) / \text{this} \} c_0
\end{align*}
\]

\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \left\{ \ldots ; \ldots d \right\} \\
&\quad d_i = c_i \ m(c_i, x) \ \{ \text{return } c; \}
\end{align*}
\]

\[
\begin{align*}
\text{body}(m, c) &= x \to c
\end{align*}
\]

\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \left\{ \ldots ; \ldots d \right\} \\
&\quad m \notin d \quad \text{type}(m, c') = x \to e
\end{align*}
\]

\[
\begin{align*}
\text{body}(m, c) &= x \to e
\end{align*}
\]

\[
\begin{align*}
\text{body}(m, c) &= x \to c_0 \ \epsilon \text{ value} \\
&\quad \epsilon \text{ value} \\
&\quad \text{new } c(c) . m(c') \mapsto \{ \epsilon / x \} \{ \text{new } c(c) / \text{this} \} c_0
\end{align*}
\]

\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \left\{ \ldots ; \ldots d \right\} \\
&\quad d_i = c_i \ p(c_i, x) \ \{ \text{return } c; \}
\end{align*}
\]

\[
\begin{align*}
\text{should be } m \quad \text{body}(m, c) &= x \to c
\end{align*}
\]

\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \left\{ \ldots ; \ldots d \right\} \\
&\quad m \notin d \quad \text{type}(m, c') = x \to e
\end{align*}
\]

\[
\begin{align*}
\text{should be } \text{body} \quad \text{body}(m, c) &= x \to e
\end{align*}
\]
Class Cast

Runtime Check wrt particular class table

\[
\frac{c < c'}{\text{value}}
\]

\[
\frac{e \to e'}{\text{new } c(e) \to \text{new } c(e')}
\]

Search Rules

\[
\begin{align*}
& e_0 \to e'_0 \\
& c_0 \to c'_0 \to e_0 \cdot f \to e'_0 \cdot f \\
& c_0 \to c'_0 \\
& e_0 \cdot m(e) \to e'_0 \cdot m(e) \\
& e_0 \cdot m(e) \to e_0 \cdot m(e') \\
& e_0 \to e'_0 \\
& (c) e_0 \to (c) e'_0
\end{align*}
\]

Search Rules (cont.)

\[
\frac{e_1 \text{ value} \ldots e_{i-1} \text{ value} \quad e_i \to e'_i}{e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n \to e'_1, \ldots, e'_{i-1}, e'_i, e'_{i+1}, \ldots, e_n}
\]

Left to right evaluation

Safety Theorem

If \((T, e) \text{ ok and } e \mapsto e'\) then either

1. \(e'\) value, or
2. there exists \(e''\) such that \(e' \mapsto e''\), or
3. \(e'\) contains a bad cast of the form \((c) \text{ new } (c')(e_0)\) where \(c' \ll c\)

"Well formed programs only get stuck in well-defined ways!"

Proof of Type Safety

Theorem 25.1 (Preservation)

Assume that \(T\) is a well-formed class table. If \(e : \tau \text{ and } e \mapsto e'\), then \(e' : \tau'\) for some \(\tau'\) such that \(\tau' \ll \tau\).

Theorem 25.2 (Progress)

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

1. \(v\) value, or
2. \(e\) contains an instruction of the form \((c) \text{ new } (e_0)\) with \(e_0\) value and \(d \ll c\), or
3. there exists \(e'\) such that \(e \mapsto e'\).

Lemma 25.3 (Canonical Forms)

If \(e : c\) and \(e\) value, then \(e\) has the form \(\text{new } (e_0)\) with \(e_0\) value and \(d \ll c\).

Proof of Type Safety

Theorem 25.1 (Preservation)

Assume that \(T\) is a well-formed class table. If \(e : \tau \text{ and } e \mapsto e'\), then \(e' : \tau'\) for some \(\tau'\) such that \(\tau' \ll \tau\).

Theorem 25.2 (Progress)

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

1. \(v\) value, or
2. \(e\) contains an instruction of the form \((c) \text{ new } (e_0)\) with \(e_0\) value and \(d \ll c\), or
3. there exists \(e'\) such that \(e \mapsto e'\).

Lemma 25.3 (Canonical Forms)

If \(e : c\) and \(e\) value, then \(e\) has the form \(\text{new } (e_0)\) with \(e_0\) value and \(d \ll c\).
Factorial in FJ

Define a new abstract class Nat

class Nat extends Object {
    Nat(){ super(); }
    Nat add(Nat n) { /* loop */
        return this.add(n); }
    Nat mul(Nat m) { /* loop */
        return this.mul(m); }
    Nat fact() { /* loop */
        return this.fact(); }
}

The Zero Subclass

Define Zero which is a subclass of Nat

class Zero extends Nat {
    Zero() { super(); }
    Nat add(Nat n){ return n; }
    Nat mul(Nat m){return new Zero(); }
    Nat fact() {
        return new Succ(new Zero()); }
}

The Succ Subclass

class Succ extends Nat {
    Nat n;
    Succ(Nat n){super(); this.n = n;}
    Nat add(Nat n) { return new Succ(this.n.add(n)); }
    Nat mul(Nat m) { return m.add(m.mul(this.n)); }
    Nat fact() { return this.mul(this.n.fact()); }
}

Summary

• We can apply operational techniques to describe the core of Java
• Does it scale to full language?
• Semantics provide description of high-level “reference interpreter” less ambiguous than English text
  – More inscrutable to non-experts
  – Should not be too inscrutable to a COS441 student

Summary of Corrections to Notes

Page 203

\[
\begin{align*}
\Gamma &\vdash v_0 : d \\
\Gamma &\vdash (c) v_0 : e
\end{align*}
\]

Rule 25.5

\[
\begin{align*}
k = c(c', c'; c') \{ \text{super}(y'); \text{this}.f = x; \}
\text{fields}(c') = c' f' c'' \text{ ok in } c
\end{align*}
\]

Rule 25.9

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

should be \( d \)
Buggy Rule 25.10

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

Fixed Rule 25.10

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; \ldots \} \]
\[ \forall x : c, \text{this}: c \vdash b : c_0 \quad c_0 <: c_0 \]
\[ \text{if } \text{type}(m, c) = \text{c}_\text{aug} \rightarrow \text{c}_\text{ref} \text{ then } c = \text{c}_\text{aug} \text{ and } c_0 = \text{c}_\text{ref} \]
\[ c_0 \text{ m}(x) \{ \text{return } c_0 \}; \text{ ok in } c \]

Page 204

Rule 25.15

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; c'' \} \]
\[ c'' = \text{m}(c_0, c) \{ \text{return } c_1 \} \]
\[ \text{should be } d \]

Rule 25.16

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; c'' \} \]
\[ \text{type}(m_j, c) = c_j \rightarrow c_j \]
\[ \text{should be } d \]

Page 206

Rule 25.27

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots ; d \} \]
\[ d_i = i_j \text{ m}(c_j, x) \{ \text{return } c \} \]
\[ \text{should be } m \]

Rule 25.28

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

Page 207

Theorem 25.1 (Preservation)
Assume that \( T \) is a well-formed class table. If \( c : \tau \) and \( e \rightarrow c' \), then \( c' : \tau' \) for some \( \tau' \) such that \( \tau' < \tau \).

Theorem 25.2 (Progress)
Assume that \( T \) is a well-formed class table. If \( c : \tau \) then either
1. \( \tau \) value, or
2. \( e \) contains an instruction of the form \( (c) \text{ used } (a) \) with \( c_0 \) value and \( d \notin c_0 \).
3. there exists \( c' \) such that \( e \rightarrow c' \), should be \( d \).

Lemma 25.3 (Canonical Forms)
If \( c : \tau \text{ and } \tau \text{ value}, then \( c \) has the form \( \text{used } (a) \text{ with } c_0 \text{ value and } d \notin c \).