Featherweight Java, FJ

Type Systems Course,
Spring 2012, Computer Science dept.,
Lund University

Amr Ergawy
23 May 2012
Outline

- References.
- Prerequisites and Discussion Scope.
- FJ: What and Why?
- Type systems: Nominal vs. Structural.
- FJ details: syntax, evaluation, and typing.
- FJ soundness: progress and preservation.
- An extension: Generic FJ.
- Optional discussion: detecting logical type errors with FJ?
- Suggested exercises.
References

1. Types and Programming Languages, Benjamin C. Pierce.
2. Slides from week 13 of the course Software foundations course given at EPFL by Martin Odersky.
http://doi.acm.org/10.1145/503502.503505.
Prerequisites and Discussion Scope

• From the course book [1], discussing FJ depends on these topics:
  - Induction, grammars, semantics, evaluations, derivations, and typed BN.
  - The typing relation, λ-Calculus, typed λ-Calculus, simple extensions of λ-Calculus, and subtyping.

• In the course book [1], these topics are discussed in chapters 2, 3, 5, 8, 9, 11, 15.

• Excuse me! I can not engage in discussions outside this scope! 😞
FJ: What and Why? 1/2

- FJ is a compact formal model that enables studying properties of Java and its extensions.
- Formally:
  - FJ is a subset of Java.
  - This subset includes key features of Java.
  - This subset excludes complex features of Java.
- As a result:
  - We can apply well-understood theory to this subset, i.e. FJ.
  - Incrementally, we can apply the same theory to extensions, e.g. Generic FJ and Feature FJ.
- Every FJ program is a functional Java Program.
FJ: What and Why? 2/2

• Applying the theory of type-safety on FJ focuses on these central features:
  – Mutually recursive class definition, Subtyping.
  – Object creation, Casting.
  – Field access.
  – Method invocation, override, recursion through “this” keyword.

• Applying the theory of type-safety on FJ excludes these features:
  – Assignment, interfaces, overloading, using “super” keyword, null pointers, base types, abstract methods, inner classes, shadowing super class fields, access control, and exceptions
  – Other more advanced features, e.g. concurrency and reflection.
Example FJ Program 1/3

• Omitting the assignment defines pure functional FJ:
  – Objects only initialized via constructors.
  – No state modifications.
• “=” and “super” only appear in a constructor.
• The method setfst() returns a new object, but does not modify the existing one.

```java
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);
    }
}

Copied from reference [2]
```
• In a constructor, the order of parameters is the same as the order of the class fields:
  – Accessing a field is just selecting the constructor parameter that has the same order.

```java
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);
    }
}
```
Copied from reference [2]
A value in FJ is a “new”-term.
A method invocation looks up the method in the “new”-term that exists before its preceding ‘.’:
  – Enabling method overriding.
A method invocation substitutes its parameters:
  – “this” is considered a variable and is substituted by the object itself.
  – Enabling recursion through “self”.

```java
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);
    }
}
```
Copied from reference [2]
Nominal type systems:
- Type names are essential, e.g. used in a class table.
- Subtyping is explicitly declared.

Structural type systems:
- Type structures may be used in exchange with type names.
- Subtyping is directly defined on the structures of types, e.g. $T$-RcdWidth on records subtyping, Chapter 11 in the course book [1].

“Nominal vs. Structural” is a continuous discussion in the research community.

In this presentation, we focus on the advantages of nominal type system, as it is used by FJ.
Advantages of nominal type systems:

- Easy run-time type tests, e.g. using “`instanceof`”.
- It is simple to support:
  - Recursive types.
  - Mutually recursive types.
- It is required to only check for once that a type is “structurally” a sub-type of another that is explicitly declared its “super” type. Then subtyping is checked from a type table.
- It prevents ”spurious* subsumption” : a compiler ”structurally” accepts a subtyping relation of two completely unrelated types.

* `spurious`: fake, unreal
FJ Syntax 1/5

• A dashed term or class symbol: a list of ‘,’ separated terms or class symbols.

• A dashed term or class symbol followed by ‘;’: a list of ‘;’ separated terms or class symbols.

• A value in FJ is a “new”-term.

\[
\begin{align*}
  t & \ ::= \\
    & \quad \quad x \\
    & \quad \quad t.f \\
    & \quad \quad t.m(t) \\
    & \quad \quad \text{new C(t)} \\
    & \quad \quad (C) t \\

  v & \ ::= \\
    & \quad \quad \text{new C(v)}
\end{align*}
\]

terms
  variable
  field access
  method invocation
  object creation
  cast

values
  object creation

Copied from reference [2]
• In the method declaration, M, \( \bar{x} \) is bound in t.

K ::= \hspace{1cm} \textit{constructor declarations}
    C(C \bar{f}) \{super(\bar{f}); this.\bar{f}=\bar{f};\}

M ::= \hspace{1cm} \textit{method declarations}
    C m(C \bar{x}) \{return t;\}

CL ::= \hspace{1cm} \textit{class declarations}
    class C extends C \{C \bar{f}; K \bar{M}\}

Copied from reference [2]
**FJ Syntax 3/5**

- CT is the "class table".
- For sanity conditions of CT, see page 256 of the course book [1].
- Because of the CT sanity and explicit declaration of inheritance, i.e. subtyping:
  - class Object is assured to be the "top" of the class hierarchy.
  - No need for a "Top" rule.
  - The book answer to exercise 19.4.1.

\[
CT(C) = \text{class } C \text{ extends } D \{ \ldots \} \\
C <: D \\
C <: C \\
C <: D, \quad D <: E \\
C <: E
\]

Copied from reference [2]
• Lookups: fields, method type, and method body.

• Note the role of the super class.

\[ CT(C) = \text{class } C \text{ extends } D \; \{ \text{C } \bar{f}; \text{ K M} \} \]
\[ B \; m \; (\bar{B} \; \bar{x}) \; \{ \text{return } t; \} \in \bar{M} \]
\[ mbody(m, C) = (\bar{x}, t) \]

\[ CT(C) = \text{class } C \text{ extends } D \; \{ \text{C } \bar{f}; \text{ K M} \} \]
\[ m \text{ is not defined in M} \]
\[ mbody(m, C) = mbody(m, D) \]

\[ \text{fields(Object) = } \emptyset \]
\[ CT(C) = \text{class } C \text{ extends } D \; \{ \text{C } \bar{f}; \text{ K M} \} \]
\[ \text{fields(D) = } \bar{D} \; \bar{g} \]
\[ \text{fields(C) = } \bar{D} \; \bar{g}, \text{ C } \bar{f} \]

Copied from reference [2]
To override a method of a super class in a subclass, we must keep the same parameters and return types.

\[ mtype(m, D) = \bar{D} \rightarrow \bar{D}_0 \implies \bar{C} = \bar{D} \text{ and } C_0 = D_0 \]

\[ override(m, D, \bar{C} \rightarrow C_0) \]

Copied from reference [2]
In these computation evaluation rules, a non-well typed "program" sticks when:

- Attempting to access a field not in a class: field lookup fails.
- Attempting to invoke a method not in a class: method lookup fails.
- Solution: compiler's job?

A well-typed program may stick in evaluation using E-CastNew if C is not a subclass of D:

- In the course book [1], chapter 15, page 195, run-time check is suggested to recover preservation. One suggest recovering mechanism is raising exceptions.
- In chapter 19, exercise 19.4.4 suggests following chapter 14 to extend FJ with exceptions.

\[
\begin{align*}
\text{fields}(C) &= \bar{C} \ f \\
\text{(new C(\bar{v}))}.f_i &\rightarrow v_i \quad \text{(E-PROJNEW)} \\
\text{mbody}(m, C) &= (\bar{x}, t_0) \\
\text{(new C(\bar{v}))}.m(\bar{u}) &\rightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new C(\bar{v})}]t_0 \quad \text{(E-INVKNEW)} \\
C &\triangleleft D \\
(D)(\text{new C(\bar{v}))} &\rightarrow \text{new C(\bar{v})} \quad \text{(E-CASTNEW)}
\end{align*}
\]

Copied from reference [2]
• Congruence rules:

\[
\begin{align*}
    t_0 & \rightarrow t'_0 \\
    t_0.f & \rightarrow t'_0.f \\
    t_0 & \rightarrow t'_0 \\
    t_0.m(t) & \rightarrow t'_0.m(t) \\
    t_i & \rightarrow t'_i \\
    v_0.m(\overline{v}, t_i, \overline{t}) & \rightarrow v_0.m(\overline{v}, t'_i, \overline{t}) \\
    t_i & \rightarrow t'_i \\
    \text{new } C(\overline{v}, t_i, \overline{t}) & \rightarrow \text{new } C(\overline{v}, t'_i, \overline{t}) \\
    t_0 & \rightarrow t'_0 \\
\end{align*}
\]

(E-FIELD)  
(E-INVK-RECV)  
(E-INVK-ARG)  
(E-NEW-ARG)  
(E-CAST)

Copied from reference [2]
FJ Typing 1/3

\[ x : C \in \Gamma \]
\[ \Gamma \vdash x : C \]  \hspace{1cm} (T-VAR)

\[ \Gamma \vdash t_0 : C_0 \]
\[ \text{fields}(C_0) = \overline{C} f \]
\[ \Gamma \vdash t_0.f_i : C_i \]  \hspace{1cm} (T-FIELD)

\[ \Gamma \vdash t_0 : C_0 \]
\[ \text{mtype}(m, C_0) = \overline{D} \rightarrow C \]
\[ \Gamma \vdash t : \overline{C} \]
\[ \overline{C} <: \overline{D} \]
\[ \Gamma \vdash t_0.m(t) : C \]  \hspace{1cm} (T-INVK)

\[ \text{fields}(C) = \overline{D} f \]
\[ \Gamma \vdash t : \overline{C} \]
\[ \overline{C} <: \overline{D} \]
\[ \Gamma \vdash \text{new } C(t) : C \]  \hspace{1cm} (T-NEW)

Copied from reference [2]
(A)(Object)newB() has two halves that both seem well typed:
- one down-cast and another up-cast
- but calling by value evaluates the term to a casting from a type that has no relation to the casting type.

We evaluated from a well-type term to an ill-typed, breaking preservation.

Just because such term is possible to exist:
- T-Scast exists so that type preservation proofs passes failed casting.
- A type checker produces "stupid warning" if it uses that rule.

A question: to make real use of T-Dcast, do not we need to add a rule called: “E-DownCastNew” and a runtime type check like “instanceof”?

\[
\begin{align*}
(A)(\text{Object})\text{new } B() & \rightarrow (A)\text{new } B() \\
\Gamma \vdash t_0 : D & \quad C \not\leq D & D \not\leq C \\
\text{stupid warning} & \\
\hline
\Gamma \vdash (C)t_0 : C
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_0 : D & \quad D \leq C \\
\Gamma \vdash (C)t_0 : C
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_0 : D & \quad C \leq D & C \neq D \\
\Gamma \vdash (C)t_0 : C
\end{align*}
\]
In "M Ok in C", \( t_0 \), the body of \( m \), is of type \( E_0 \), which is subclass of \( C_0 \), the type of the body of the same method in class D.

"C Ok" defines the conditions that C must satisfy to be accepted as extending class D.

\[
\begin{align*}
\bar{x}: \overline{C}, \text{this}: C \vdash t_0 : E_0 & \quad E_0 <: C_0 \\
CT(C) = \text{class } C \text{ extends } D \{\ldots\} & \quad \text{override}(m, D, \overline{C} \rightarrow C_0) \quad C_0 \ m \ (\overline{C} \ (\bar{x})) \ \{\text{return } t_0;\} \ \text{OK in } C \\
K = C(\overline{D} \overline{g}, \overline{C} \overline{f}) \ \{\text{super}(\overline{g}); \ \text{this.}\overline{f} = \overline{f};\} & \quad \text{fields}(D) = \overline{D} \overline{g} \quad M \ \text{OK in } C \\
\text{class } C \text{ extends } D \ \{\overline{C} \overline{f}; \ K \overline{M}\} \ \text{OK}
\end{align*}
\]

Copied from reference [2]
FJ Soundness 1/2

• Given $E$-CastNew, a program with this term sticks.

• For the current set of FJ evaluation rules, we redefine the progress property as:

$E[(C)(\text{new } D(v))]$, with $!(D <: C)$ means the next subterm to be reduced is stuck at $E$-CastNew.

– *Solution*: extend FJ with raising exceptions?

• Proof is by induction on typing derivations: I attempted it executing T-VAR and proving it for T-FIELD, ... etc.
Theorem [Preservation]: If $\Gamma \vdash t : C$ and $t \rightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' <: C$.

Copied from reference [2]

- As discussed before, an apparent well-typed term may reduce to an ill-typed cast term.
- T-SCast exists so that type preservation proofs passes such failed casting.
- A type checker produces "stupid warning" if it uses that rule.
An extension: Generic FJ

• If you are interested, let's go through pages 408 to 415 on the reference paper [3].
Optional discussion: detecting logical type errors with FJ?

• Does FJ require any extension to use it for detecting logical errors?

• Or, is it enough to define a class of each logical type we are interested in?
  – Is not this similar to the solution based on single-field variants in the course book [1], chapter 11, on page 139.
Suggested exercises

• Suggested exercise:
  – Should be simple: 19.4.5, 19.4.7. I already attempted them if any one is interested.
  – More demanding: 19.4.3, 19.4.4