Featherweight Java

Advanced Compiler Construction and Program Analysis

Lecture 9

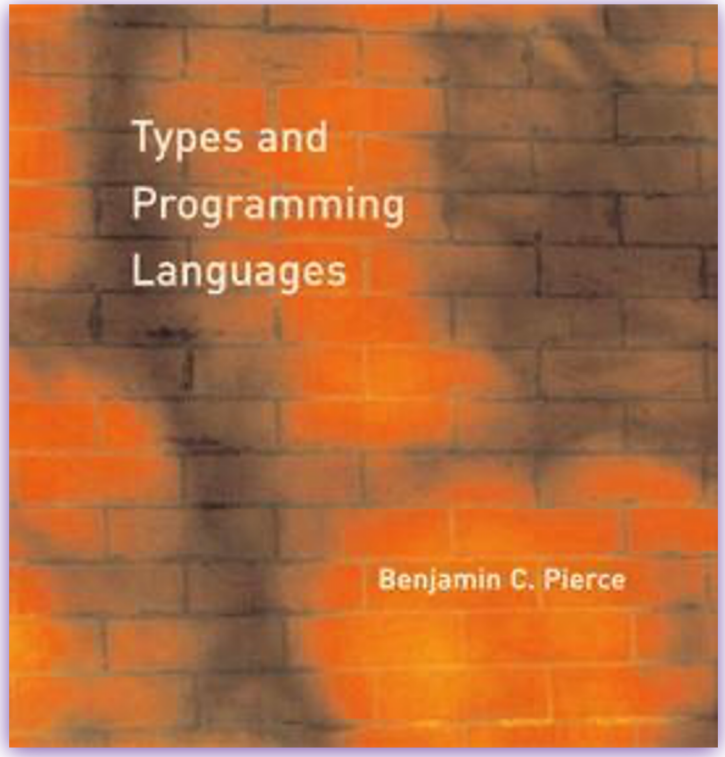
Innopolis University, Spring 2022

The topics of this lecture are covered in detail in…

Benjamin C. Pierce.

Types and Programming Languages

MIT Press 2002



Featherweight Java

1. A simplified core of Java
2. Useful for reasoning and proofs
3. A program in FJ is also a program in Java
4. Primarily useful to study language extensions for Java and how different extensions interoperate with each other

Featherweight Java: A Minimal Core Calculus for Java and GJ

ATSUSHI IGARASHI
University of Tokyo
BENJAMIN C. PIERCE
University of Pennsylvania
and
PHILIP WADLER
Avaya Labs

Several recent studies have introduced lightweight versions of Java: reduced languages in which complex features like threads and reflection are dropped to enable rigorous arguments about key properties such as type safety. We carry this process a step further, omitting almost all features of the full language (including interfaces and even assignment) to obtain a small calculus, Featherweight Java, for which rigorous proofs are not only possible but easy. Featherweight Java bears a similar relation to Java as the lambda-calculus does to languages such as ML and Haskell. It offers a similar computational "feel," providing classes, methods, fields, inheritance, and dynamic typecasts with a semantics closely following Java's. A proof of type safety for Featherweight Java thus illustrates many of the interesting features of a safety proof for the full language, while remaining pleasingly compact. The minimal syntax, typing rules, and operational semantics of Featherweight Java make it a handy tool for studying the consequences of extensions and variations. As an illustration of its utility in this regard, we extend Featherweight Java with *generic classes* in the style of GJ (Bracha, Odersky, Stoutamire, and Wadler) and give a detailed proof of type safety. The extended system formalizes for the first time some of the key features of GJ.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*;

Featherweight Java: overview

```
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);
    }
}
```

Featherweight Java: overview

```
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);
    }
}
```

1. Always include superclass

Featherweight Java: overview

```
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);
    }
}
```

2. Always name the receiver

Featherweight Java: overview

```
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);
    }
}
```

3. Constructors are simple

Featherweight Java: overview

```
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);
    }
}
```

4. Method body is **return ...**

Featherweight Java: terms

Object constructors

```
new A()
new B()
new Pair(…, …)
```

Method invocations

```
….setfst(…)
….getsnd()
```

Field access

```
this.fst
….x.setfst(…).z
```

Variables

```
this
newfst
```

Cast

```
(Pair)pair.fst
```

Featherweight Java: example

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

Featherweight Java: example

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

⇒ `new Pair(new B(), new B())`

Featherweight Java: example

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

⇒ new Pair(new B(), new B())


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

Featherweight Java: example

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

⇒ `new Pair(new B(), new B())`

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

⇒ `new B()`

Featherweight Java: evaluation preview

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

Featherweight Java: evaluation preview

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

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

Featherweight Java: evaluation preview

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

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

Featherweight Java: evaluation preview

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

new Pair(new A(), new B()).setfst(new B())
⇒              [newfst ↦ new B()]
               [this   ↦ new Pair(new A(), new B())]
               new Pair(newfst, this.snd)
```

Featherweight Java: evaluation preview

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

new Pair(new A(), new B()).setfst(new B())
⇒              [newfst ↦ new B()]
               [this   ↦ new Pair(new A(), new B())]
               new Pair(newfst, this.snd)
⇒              new Pair(new B(), (new Pair(new A(), new
B())).snd)
```

Featherweight Java: evaluation preview

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

new Pair(new A(), new B()).setfst(new B())
⇒              [newfst ↦ new B()]
               [this   ↦ new Pair(new A(), new B())]
               new Pair(newfst, this.snd)
⇒              new Pair(new B(), (new Pair(new A(), new
B()))).snd)
```

Featherweight Java: evaluation preview

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

new Pair(new A(), new B()).setfst(new B())
⇒              [newfst ↦ new B()]
               [this   ↦ new Pair(new A(), new B())]
               new Pair(newfst, this.snd)
⇒              new Pair(new B(), (new Pair(new A(), new
B()))).snd)

(Pair)new Pair(new A(), new B())
⇒              new Pair(new A(), new B())
```

Nominal vs Structural Type Systems

Structural	Nominal
`NatPair = {fst:Nat, snd:Nat}`	`class NatPair extends Object { Nat fst; Nat snd; … }`
`{fst:Nat, snd:Nat, thd:Nat} <: {fst:Nat, snd:Nat}`	`class NatTriple extends NatPair … NatTriple <: NatPair`
Difficult to typecheck at runtime	Easy to typecheck at runtime (`instanceOf`, downcasting, printing, serializing, reflection, …)
Fixpoint combinator for recursive types	Easy recursive types
Non-trivial subtyping tests	Easy subtyping test
Single-constructor types (wrappers)	Explicit subtyping

Featherweight Java: syntax

```
CL ::= class C extends C {C f; K; M}
K ::= C(C f) { super(f); this.f = f}
M ::= C m(C x) { return t; }

t ::=
  x
  t.f
  t.m(t)
  new C(t)
  (C) t

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

class declaration
constructor declaration
method declaration

terms
variable
field access
method invocation
object construction
cast

values
object construction

Featherweight Java: subtyping

$$S <: S$$

$$\frac{S <: U \qquad U <: T}{S <: T}$$

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

Featherweight Java: subtyping with Object

Exercise 9.1. Explain why we do not need a special subtyping rule for `Object`, like we did for the `Top` type.

Featherweight Java: field lookup

$$\mathit{fields}(\mathtt{Object}) = \emptyset$$

$$\frac{\mathtt{CT(C) = class\ C\ extends\ D\ \{A\ f;\ K;\ M\}} \qquad \mathit{fields}(\mathtt{D}) = \mathtt{B\ g}}{\mathit{fields}(\mathtt{C}) = \mathtt{A\ f,\ B\ g}}$$

Featherweight Java: method type lookup

$$\frac{\texttt{CT(C) = class C extends D \{A f; K; M\}} \quad \texttt{B m(E x) \{…\}} \in \texttt{M}}{\textit{mtype}\texttt{(m, C) = E} \rightarrow \texttt{B}}$$

$$\frac{\texttt{CT(C) = class C extends D \{A f; K; M\}} \quad \texttt{m not in M}}{\textit{mtype}\texttt{(m, C) = }\textit{mtype}\texttt{(m, D)}}$$

Featherweight Java: method body lookup

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \ \{\underline{A\ f};\ K;\ \underline{M}\} \qquad B\ m(\underline{E\ x})\ \{\ \textbf{return } t;\ \} \in \underline{M}}{\mathit{mbody}(m, C) = (\underline{x}, t)}$$

$$\frac{CT(C) = \textbf{class } C \textbf{ extends } D \ \{\underline{A\ f};\ K;\ \underline{M}\} \qquad m \text{ not in } \underline{M}}{\mathit{mtype}(m, C) = \mathit{mbody}(m, D)}$$

Featherweight Java: valid method overloading

$$\frac{\mathit{mtype}(\mathtt{m},\ \mathtt{D}) = \underline{\mathtt{E}} \to \mathtt{F} \text{ implies } \underline{\mathtt{E}}=\underline{\mathtt{A}} \text{ and } \mathtt{F}=\mathtt{B}}{\mathit{override}(\mathtt{m},\ \mathtt{D},\ \underline{\mathtt{A}} \to \mathtt{B})}$$

Featherweight Java: evaluation

$$\frac{\mathit{fields}(C) = \underline{A\ f}}{(\textbf{new}\ C(\underline{v})).f_n \longrightarrow v_n}$$

$$\frac{\mathit{mbody}(m, C) = (\underline{x}, t)}{(\textbf{new}\ C(\underline{v})).m(\underline{u}) \longrightarrow [\text{this} \mapsto \textbf{new}\ C(\underline{v}),\ \underline{x} \mapsto \underline{u}]t}$$

$$\frac{C <: D}{(D)(\text{new}\ C(\underline{v})) \longrightarrow \text{new}\ C(\underline{v})}$$

Featherweight Java: term typing

$$\frac{\Gamma \vdash t : C \qquad \mathit{fields}(C) = \underline{A\ f}}{\Gamma \vdash t.f_n : A_n}$$

$$\Gamma, x{:}T \vdash x : T$$

$$\frac{\Gamma \vdash t : C \qquad \mathit{mtype}(m, C) = \underline{A} \to B \qquad \Gamma \vdash \underline{s} : \underline{D} \qquad \underline{D} <: \underline{A}}{\Gamma \vdash t.m(\underline{s}) : B}$$

$$\frac{\mathit{fields}(C) = \underline{A\ f} \qquad \Gamma \vdash \underline{t} : \underline{B} \qquad \underline{B} <: \underline{A}}{\Gamma \vdash \mathbf{new}\ C(\underline{t}) : C}$$

Featherweight Java: term typing (casts)

$$\frac{\Gamma \vdash t : D \qquad D <: C}{\Gamma \vdash (C)t : C}$$

$$\frac{\Gamma \vdash t : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)t : C}$$

$$\frac{\Gamma \vdash t : D \qquad D \not<: C \qquad D \not<: C \qquad \textit{issue a warning}}{\Gamma \vdash (C)t : C}$$

Featherweight Java: “stupid” cast example

```
(A) ((Object) new B())   ⇒   (A) (new B())
```

Featherweight Java: “stupid” cast example

Exercise 9.2. Extend FJ with means to assign new values to the fields in the bodies of methods. Use references.

Exercise 9.3. Extend FJ with versions of **`raise`** and **`try`**.

Exercise 9.4. Show how to extend FJ with interfaces.

Summary

- ❏ Featherweight Java

See you next time!