Featherweight Java

Chapter 19

Benjamin Pierce

Types and Programming Languages

Eiffel, 1989

Cook, W.R. (1989) - *A Proposal for Making Eiffel Type-Safe*, in Proceedings of ECOOP'89. S. Cook (ed.), pp. 57-70. Cambridge University Press.

Betrand Meyer, on unsoundness of Eiffel:
"Eiffel users universally report that they almost never run into such problems in real software development."

Ten years later: Java

Java is not type-safe

Vijay Saraswat
AT&T Research, 180 Park Avenue, Florham Park NJ 07932

Java$_{light}$ is Type-Safe — Definitely

Tobias Nipkow and David von Oheimb*
Fakultät für Informatik, Technische Universität München
http://www4.informatik.tu-muenchen.de/~{nipkow|oheimb}

Proving Java Type Soundness

Don Syme*
email: drs1004@cl.cam.ac.uk

June 17, 1997

Java is Type Safe — Probably

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Interesting Aspects of Java

- Object Oriented
  - (Almost) everything is an object
  - Single inheritance
    - Adding fields
    - Method override
    - Open recursion
  - Interfaces
  - Encapsulation
- Reflection
- Concurrency
- Libraries
- Type safety
  - Well typed programs have no undefined semantics

Featherweight Java

- (Minimal) Purely functional object oriented strict subset of Java
- Supports
  - Everything is an object
  - Single inheritance
    - Adding fields
    - Method override
    - Open recursion
- Simple
  - Operational Semantics
  - Type Checking
  - Proof of safety
- Extensions
  - Interface
  - Inner classes
  - Polymorphism

Featherweight Java

CL ::= class declarations

class c extends C { C f ; K M }

K ::= constructor declarations

C (C f) { super(f) ; this.f=f ;}

M ::= method declarations

C m(C x) { return t;}

t ::= terms

x	variable
t.f	field access
t.m(t)	method invocation
new C(t)	object creation
(C) t	cast

v::= values

new C(v)

A Simple Example

```
class Bicycle extends object {
  int currentSpeed ; // field
  int currentGear ; // field
  Biycle(int s, int g) { // constructor
       super() ;
       this.currentSpeed= s ;
       this.currentGear= g ;
  }
Bicycle UpShift ()     {
  return new Bicycle(this.currentSpeed,
this.currentGeer+1) ; }
 }
```

```
class MountainBike extends Bicycle {
  int LowerGear;
  MountainBike(int s, int g, int l) {
       super(s, g) ;
       this.LowerGear= l ;    }
Bicycle UpShift ()
    {  … }
 }
```

```
class Main extends object {
      Bicycle  b;
      Main() {
          super() ; this.b = new MountainBike(3, 3, 5); }
      Bicycle UpShift() {
           return this.b.UpShift() ; }
}
```

Running example

```
class A extends Object { A() { supper(); } }

class B extends Object { B() { supper(); } }

class Pair extends Object {
   Object first;
   Object second;
   Pair(Object fst, Object snd) {
    supper();
    this first=fst;
    this second = snd;
    }
   Pair SetFst(Object newfst) {
      return new Pair(newfst, this.snd);
    }
}
```

Nominal vs. Structural Type Systems

- When are two types equal:
- Structural equivalence
  - Two isomorphic types are identical
  - NatPair = {fst: Nat, snd: Nat}
- Nominal (name equivalence) type systems
  - Compound types have name
  - The name caries significant information
  - Type name must match

Nominal vs. Structural Type Systems

Nominal

- Type name is useful at runtime
  - "Generic" programming
  - Efficient runtime checks
- Naturally supports recursive types
- Efficient subtyping checks
- Prevent "spurious" subsumption

Structural

- Type expressions are closed entities
- Supports type abstractions
  - Parametric polymorphism
  - Abstract data types
  - User defined type operators
  - …

The Class Table

- Maps class names to their class definitions (excluding objects)

Running example

```
class A extends Object { A() { supper(); } }
```
A ↦ class A extends Object {…}

```
class B extends Object { B() { supper(); } }
```
B ↦ class B extends Object {…}

```
class Pair extends Object {
   Object first;
   Object second;
   Pair(Object fst, Object snd) {
    supper();
    this first=fst;
    this second = snd;
    }
   Pair SetFst(Object newfst) {
      return new Pair(newfst, this.snd);
    }
}
```
Pair ↦ class Pair extends Object {…}

Featherweight Java with subtyping

CL ::= class declarations

class c extends C { C f ; K M }

K ::= constructor declarations

C (C f) { super(f) ; this.f=f ;}

M ::= method declarations

C m(C x) { return t;}

t ::= terms

x variable

t.f field access

t.m(t) method invocation

new C(t) object creation

(C) t cast

v::= values

new C(v)

Subtyping

$$C <: D$$

$$\frac{}{C <: C}$$

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

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

The Class Table

- Maps class names to their class definitions (excluding objects)
- A program is a class table and a term
- Consistency requirements
  - <: is acyclic (<: is a partial order)
  - CT(C)= class C … for every C in dom(CT)
  - Object $\notin$ dom(CT)
  - For every class C appearing in CT except Object, c $\in$ dom(CT)
- fields(C) = $\underline{C}\ \underline{f}$ are the fields declared in C
- mbody(m, C) = ($\underline{x}$, t) where $\underline{x}$ are the formal arguments and t is m's body

Plan

- A small step operational semantics
- Potential runtime errors
- Basic type system
- Corrections

New Evaluation

$$\frac{t_i \rightarrow t'_i}{\text{new } C(\underline{v}, t_i, \underline{t}) \rightarrow \text{new } C(\underline{v}, t'_i, \underline{t})} \quad \text{(E-New-Arg)}$$

Field Projection

$$\frac{\text{fields}(C) = \underline{C}\ \underline{f}}{\text{new } C(\underline{v}).f_i \rightarrow v_i} \quad \text{(E-ProjNew)}$$

new Pair(new A(), new Pair(new A(), new B())).snd $\rightarrow$ new Pair (new A(), new B())

$$\frac{t_0 \rightarrow t'_0}{t_0.f \rightarrow t'_0.f} \quad \text{(E-Field)}$$

new Pair(new A(), new Pair(new A(), new B())).snd.fst $\rightarrow$ new A()

Method Invocation

- Use the (actual) class to determine the exact method
- Bind actual parameters to formals
- Benefit from absence of side effects

Method Invocation

$$\frac{\text{mbody}(m, C) = (\underline{x}, t_0)}{\text{new } C(\underline{v}).m(\underline{u}) \rightarrow [\underline{x} \mapsto \underline{u}, \text{this} \mapsto \text{new } C(\underline{v})]\ t_0} \quad \text{(E-InvNew)}$$

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) → (E-ProjNew)

new Pair(new B(), new B())

Method Invocation

$$\frac{\text{mbody}(m, C) = (\underline{x}, t_0)}{\text{new } C(\underline{v}).m(\underline{u}) \rightarrow [\underline{x} \mapsto \underline{u}, \text{this} \mapsto \text{new } C(\underline{v})]\ t_0} \quad \text{(E-InvNew)}$$

$$\frac{t_0 \rightarrow t'_0}{t_0.\ m(\underline{t}) \rightarrow t'_0.m(\underline{t})} \quad \text{(E-InvkRecv)}$$

$$\frac{t_i \rightarrow t'_i}{v_0.\ m(\underline{v}, t_i, \underline{t}) \rightarrow v_0.m(\underline{v}, t'_i, \underline{t})} \quad \text{(E-InvArg)}$$

Cast Invocation

- Assure that the casting is valid
- Convert the type

Cast Invocation

$$\frac{C <: D}{(D)\ (\text{new } C(\underline{v})) \rightarrow \text{new } C(\underline{v})} \quad \text{(E-CastNew)}$$

$$\frac{t_0 \rightarrow t'_0}{(C)\ t_0 \rightarrow\ (C)\ t'_0} \quad \text{(E-Cast)}$$

((Pair) new Pair(new Pair (new A(), new B()), new A()).fst).snd $\rightarrow$ (E-ProjNew)

((Pair) new Pair(new A(), new B())).snd $\rightarrow$ (E-CastNew)

new Pair(new A(), new B()).snd $\rightarrow$ (E-ProjNew)

new B()

FJ Semantics Summary

$$\frac{\text{fields}(C) = \underline{C}\ \underline{f}}{\text{new } C(\underline{v}).f_i \rightarrow v_i} \quad \text{(E-ProjNew)}$$

$$\frac{t_0 \rightarrow t'_0}{t_0.f \rightarrow t'_0.f} \quad \text{(E-Field)}$$

$$\frac{\text{mbody}(m, C) = (\underline{x}, t_0)}{\text{new } C(\underline{v}).m(\underline{u}) \rightarrow [\underline{x} \mapsto \underline{u}, \text{this} \mapsto \text{new } C(\underline{v})]\ t_0} \quad \text{(E-InvNew)}$$

$$\frac{t_0 \rightarrow t'_0}{t_0.\ m(\underline{t}) \rightarrow\ t'_0.m(\underline{t})} \quad \text{(E-InvkRecv)}$$

$$\frac{t_i \rightarrow t'_i}{v_0.\ m(\underline{v}, t_i, \underline{t}) \rightarrow\ v_0.m(\underline{v}, t'_i, \underline{t})} \quad \text{(E-InvArg)}$$

$$\frac{C <: D}{(D)\ (\text{new } C(\underline{v})) \rightarrow\ \text{new } C(\underline{v})} \quad \text{(E-CastNew)}$$

$$\frac{t_i \rightarrow t'_i}{\text{new } C(\underline{v}, t_i, \underline{t}) \rightarrow \text{new } C(\underline{v}, t'_i, \underline{t})} \quad \text{(E-New-Arg)}$$

$$\frac{t_0 \rightarrow t'_0}{(C)\ t_0 \rightarrow\ \ (C)\ t'_0} \quad \text{(E-Cast)}$$

Potential Runtime Errors

- Incompatible constructor invocation
  - new Pair(new A())
- Incompatible field selection
  - new Pair(new A(), new B()).thrd
- Incompatible method invocation “message not understood”
  - new A().setfst(new B())
- Incompatible arguments of methods
- Incompatible return value of methods
- Incompatible downcasts

The Class Table

- Maps class names to their class definitions (excluding objects)
- A program is a class table and a term
- fields(C) = $\underline{C}\ \underline{f}$ are the fields declared in C
- mbody(m, C) = ($\underline{x}$, t) where $\underline{x}$ are the formal arguments and t is the method body
- mtype(m, C) = $\underline{B} \rightarrow B$ where $\underline{B}$ is the type of arguments and B is the type of the results
- override(m, D, $\underline{C} \rightarrow C_0$) holds if the method m with arguments $\underline{C}$ is redefined in a subclass of D

Featherweight Java Type Rules

$\Gamma \vdash t : C$

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C <: D \quad C \mathrel{!}= D}{\Gamma \vdash (C)\, t_0 : C} \text{ (T-DCAST)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad fields(C_0) = \underline{C}\ \underline{f}}{\Gamma \vdash t_0 .f_i : C_i} \text{ (T-FIELD)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \underline{D} \to C \qquad \Gamma \vdash \underline{t} : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash t_0 .m(\underline{t}) : C} \text{ (T-INVK)}$$

$$\frac{fields(C) = \underline{D}\ \underline{f} \qquad \Gamma \vdash \underline{t} : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash \text{new } C\ (\underline{t}) : C} \text{ (T-NEW)}$$

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

Method Typing M OK in C

$$\frac{\underline{x} : \underline{C}, \text{this} : C \vdash t_0 : E_0 \quad E_0 <: C_0 \qquad CT(C) = \text{class C extends D } \{\ldots\} \qquad override(m, D, \underline{C} \to C_0)}{C_0\ m(\underline{C}\ \underline{x})\ \{\ \text{return } t_0\ ;\ \}\ \text{OK in C}}$$

Class Typing C OK

$$\frac{K = C(\underline{D}\ \underline{g}, \underline{C}\ \underline{f})\ \{\text{super}(\underline{g}); \text{this}.\underline{f} = \underline{f};\} \qquad fields(D) = \underline{D}\ \underline{g} \qquad \underline{M}\ \text{ok in C}}{\text{class C extends D } \{\ \underline{C}\ \underline{f}\ K\ \underline{M}\ \}\ \text{OK}}$$

Type Safety(19.5)

- Well typed programs cannot go wrong
  - No undefined semantics
  - No runtime checks
- If t is well typed then either t is a value or there exists an evaluation step $t \rightarrow t'$ [Progress]
- If t is well typed and there exists an evaluation step $t \rightarrow t'$ then t' is also well typed [Preservation]

Type Preservation: Take 1

- If $\Gamma \vdash$ t : C and t $\rightarrow$ t' then $\Gamma \vdash$ t' : C
- Counterexample

(Object) new B() $\rightarrow$ new B() (E-CastNew)

Type Preservation: Take 2

- If $\Gamma \vdash$ t : C and t $\rightarrow$ t' then there exists C' such that C' <: C and $\Gamma \vdash$ t' : C'
- Counterexample
  - (Object) new B() $\rightarrow$ new B() (E-CastNew)
  - (A) (Object) new B() $\rightarrow$ (A) new B() (E-Cast)

$\Gamma \vdash t : C$

Featherweight Java Type Rules

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C} \text{ (T-VAR)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C \not<: D \qquad D \not<: C}{\Gamma \vdash (C)\, t_0 : C} \text{ stupid warning (T-SCAST)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad fields(C_0) = \underline{C}\ \underline{f}}{\Gamma \vdash t_0 .f_i : C_i} \text{ (T-FIELD)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad mtype(m, C_0) = \underline{D} \to C \qquad \Gamma \vdash \underline{t} : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash t_0 .m(\underline{t}) : C} \text{ (T-INVK)}$$

$$\frac{fields(C) = \underline{D}\ \underline{f} \qquad \Gamma \vdash \underline{t} : \underline{C} \qquad \underline{C} <: \underline{D}}{\Gamma \vdash \text{new } C\ (\underline{t}) : C} \text{ (T-NEW)}$$

Method Typing — M OK in C

$$\frac{\underline{x}: \underline{C}, \text{this}: C \vdash t_0 : E_0 \quad E_0 <: C_0 \qquad CT(C) = \text{class C extends D } \{\ldots\} \qquad override(m, D, \underline{C} \to C_0)}{C_0\ m(\underline{C}\ \underline{x})\ \{\ \text{return } t_0\ ;\ \}\ \text{OK in C}}$$

Class Typing — C OK

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

$$\frac{\Gamma \vdash t_0 : D \qquad C <: D \qquad C \mathrel{!}= D}{\Gamma \vdash (C)\, t_0 : C} \text{ (T-DCAST)}$$

$$\frac{K = C(\underline{D}\ \underline{g}, \underline{C}\ \underline{f})\ \{\text{super}(\underline{g}); \text{this}.\underline{f} = \underline{f};\} \qquad fields(D) = \underline{D}\ \underline{g} \qquad \underline{M}\ \text{ok in C}}{\text{class C extends D } \{\ \underline{C}\ \underline{f}\ K\ \underline{M}\ \}\ \text{OK}}$$

Progress Theorem

- If a program is well typed then the only way to get stuck is if it reaches a point in which it cannot perform a downcast
- If t is a well typed term
  - If t = new $C_0(\underline{t}).f$ then fields(C_0) = $\underline{C}\ \underline{f}$ and $f \in \underline{f}$
  - If t = new $C_0(\underline{t}).m(\underline{s})$ then mbody(m, C_0) = ($\underline{x}$, t_0) and $|\underline{x}| = |\underline{s}|$
- if t is a closed well typed in a normal form then either t is a value or “t is a cast”

Evaluation Contexts

- Terms with a hole

  E ::= evaluation contexts

  [] hole

  E.f field access

  E.m(t) method invocation (receiver)

  v.m(v, E, t) method invocation (arg)

  new C(v, E, t) object creation(arg)

  (C) E cast

- E[t] denotes the term obtained by replacing the hole with t
- If $t \rightarrow t'$ then t= E(r) and t'= E(r') where E, r, and r' are unique and $r \rightarrow r'$ is one of the rules E-ProjNew, E-InvNew and E-CastNew
- If t is closed well defined normalized term then either t is a value or for some context E we can express t as t = E[(C)(new D(v)] where D $\not<:$ C (Progress theorem)

Encoding vs. Primitive Objects

- Two approaches for semantics and typing OO programs
  - Typed lambda calculus with records, references and subtypes (Chapter 18)
  - Object and classes are primitive mechanisms

Summary

- Establishing type safety of real programming language can be useful
- Mechanized proof systems (Isabele, Coq) can help
  - Especially useful in the design phase
- But indentifying a core subset is also useful

Quotes

- “Inside every large language there is a small language struggling to come out”
  - Igarashi, Pierce, and Wadler (1999)
- “Inside every large program there is a small program struggling to come out”
  - Sir Tony Hoare, Efficient Production of large programs (1970)
- “I’m fat but I’m thin inside”
  - George Orwell, Coming Up from Air (1939)