Design Principles of Programming Languages

Case Study: Featherweight Java

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong

Peking University

The essence of objects

➢ Dynamic dispatch

➢ Encapsulation of state with behavior

➢ Behavior-based subtyping

➢ Inheritance (incremental definition of behaviors)

➢ Access of super class

➢ Open recursion through this

What's missing (wrt. Java, say)

We haven't really captured the peculiar status of classes which are *both run-time and compile-time* things) — we've captured the run-time aspect, but not the way in which classes get used as *types* in Java.

Also not *named* types with *declared* subtyping

Nor recursive types

Nor run-time type analysis (casting, etc.)

(… nor lots of other stuff)

Modeling Java

About models (of things in general)

No such thing as a "*perfect model*" — The nature of a model is to abstract away from details!

So models are never just "good" [or "bad"]: they are always "good [or bad] for *some specific set of purposes*."

Models of Java

Lots of different purposes ⟶ lots of different kinds of models

- – Source-level vs. bytecode level
- – Large (inclusive) vs. small (simple) models
- – Models of type system vs. models of run-time features (not entirely separate issues)
- – Models of specific features (exceptions, concurrency, reflection, class loading, …)
- – Models designed for extension

Featherweight Java

Purpose: model "core OO features" and their types and *nothing else*.

History:

– Originally proposed by a Penn visiting student (Atsushi Igarashi) as a tool for analyzing GJ ("Java plus generics"), which later became Java 1.5

– Since used by many others for studying a wide variety of Java features and proposed extensions

Things left out of FJ

➢ Reflection, concurrency, class loading, inner classes, …

➢ Exceptions, loops, …

➢ Interfaces, overloading, …

➢ Assignment (!!)

Things left in FJ

➢ Classes and objects

➢ Methods and method invocation

➢ Fields and field access

➢ Inheritance (including open recursion through this)

➢ Casting

Example

```
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); }
}
```

Conventions

For syntactic regularity...

- Always include superclass (even when it is Object)
- Always write out constructor (even when trivial)
- Always call super from constructor (even when no arguments are passed)
- Always explicitly name receiver object in method invocation or field access (even when it is this)
- Methods always consist of a single return expression
- Constructors always
    - Take same number (and types) of parameters as fields of the class
    - Assign constructor parameters to "local fields"
    - Call super constructor to assign remaining fields
    - Do nothing else

Formalizing FJ

Nominal type systems

Big dichotomy in the world of programming languages:

➤ *Structural* type systems:

- What matters about a type (for typing, subtyping, etc.) is just its structure.

- Names are just convenient (but inessential) abbreviations.

➤ *Nominal* type systems:

- Types are always named.

- Typechecker mostly manipulates names, not structures.

- Subtyping is declared explicitly by programmer (and checked for consistency by compiler).

Advantages of Structural Systems

Somewhat simpler, cleaner, and more elegant (no need to always work wrt. a set of "*name definitions*")

Easier to extend (e.g. with parametric polymorphism)

Caveat: when recursive types are considered, some of this simplicity and elegance slips away…

Advantages of Nominal Systems

Recursive types fall out easily

Using names everywhere makes typechecking (and subtyping, etc.) easy and efficient

Type names are also useful at run-time (for casting, type testing, reflection, …).

Java (like most other mainstream languages) is a nominal system.

Representing objects

Our decision to omit assignment has a *nice side effect*...

The only ways in which two objects can differ are (1) their classes and (2) the parameters passed to their constructor when they were created.

All this information is available in the new expression that creates an object. So we can *identify* the created object with the new expression.

Formally: object values have the form $\text{new } C(\overline{v})$

FJ Syntax

Syntax (terms and values)

```
t ::=                                    terms
      x                                     variable
      t.f                                   field access
      t.m(t̄)                                method invocation
      new C(t̄)                              object creation
      (C) t                                 cast


v ::=                                    values
      new C(v̄)                              object creation
```

Syntax (methods and classes)

$K ::=$ *constructor declarations*

    $C(\overline{C}\ \overline{f})\ \{\mathtt{super}(\overline{f});\ \mathtt{this}.\overline{f}=\overline{f};\}$

$M ::=$ *method declarations*

    $C\ m(\overline{C}\ \overline{x})\ \{\mathtt{return}\ t;\}$

$CL ::=$ *class declarations*

    $\mathtt{class}\ C\ \mathtt{extends}\ C\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}$

Subtyping

Subtyping

As in Java, subtyping in FJ *is declared*.

Assume we have a (global, fixed) *class table* CT mapping class names to definitions.

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

$$\texttt{C} <: \texttt{C}$$

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

More auxiliary definitions

From the class table, we can read off a number of other useful properties of the definitions (which we will need later for typechecking and operational semantics)...

Field(s) lookup

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

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}} \quad fields(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{fields(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

Method type lookup

$$CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$$
$$\texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return t;}\} \in \overline{\texttt{M}}$$
$$\overline{\phantom{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}}}$$
$$mtype(\texttt{m}, \texttt{C}) = \overline{\texttt{B}}{\to}\texttt{B}$$

$$CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$$
$$\texttt{m is not defined in } \overline{\texttt{M}}$$
$$\overline{\phantom{CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}}}$$
$$mtype(\texttt{m}, \texttt{C}) = mtype(\texttt{m}, \texttt{D})$$

Method body lookup

$$CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$$
$$\texttt{B m }(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\texttt{return t};\} \in \overline{\texttt{M}}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t})$$

$$CT(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$$
$$\texttt{m is not defined in } \overline{\texttt{M}}$$

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$mbody(\texttt{m}, \texttt{C}) = mbody(\texttt{m}, \texttt{D})$$

Valid method overriding

$$mtype(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0 \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0$$
$$\overline{\hspace{8cm}}$$
$$override(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \rightarrow \mathtt{C}_0)$$

Evaluation

The example again

```
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); }
}
```

Evaluation

Projection:

$$\text{new Pair(new A(), new B()).snd} \quad \longrightarrow \quad \text{new B()}$$

Evaluation

Casting:

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

Evaluation

Method invocation:

$$\text{new Pair(new A(), new B()).setfst(new B())}$$

$$\longrightarrow \left[\begin{array}{l} \text{newfst} \mapsto \text{new B(),} \\ \text{this} \mapsto \text{new Pair(new A(),new B())} \end{array} \right]$$
$$\text{new Pair(newfst, this.snd)}$$

i.e., new Pair(new B(), new Pair(new A(), new B()).snd)

```
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); }
}
```

Evaluation

Method invocation:

$$((Pair)\ \underline{(new\ Pair(\underline{new\ Pair(new\ A(),new\ B()),\ new\ A())}}{.fst}).snd$$

$\longrightarrow$ $\underline{((Pair)new\ Pair(new\ A(),new\ B()))}.snd$

$\longrightarrow$ $\underline{new\ Pair(new\ A(),\ new\ B())}.snd$

$\longrightarrow$ $new\ B()$

```
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); }
}
```

Evaluation rules

$$\frac{fields(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{v}})).\texttt{f}_i \longrightarrow \texttt{v}_i} \quad (\text{E-ProjNew})$$

$$\frac{mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{t}_0)}{\substack{(\texttt{new C}(\overline{\texttt{v}})).\texttt{m}(\overline{\texttt{u}}) \\ \longrightarrow [\overline{\texttt{x}} \mapsto \overline{\texttt{u}}, \texttt{this} \mapsto \texttt{new C}(\overline{\texttt{v}})]\texttt{t}_0}} \quad (\text{E-InvkNew})$$

$$\frac{\texttt{C} <: \texttt{D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{v}})) \longrightarrow \texttt{new C}(\overline{\texttt{v}})} \quad (\text{E-CastNew})$$

Evaluation rules

$$\frac{t_0 \longrightarrow t_0'}{t_0.f \longrightarrow t_0'.f} \qquad (\text{E-Field})$$

$$\frac{t_0 \longrightarrow t_0'}{t_0.m(\overline{t}) \longrightarrow t_0'.m(\overline{t})} \qquad (\text{E-Invk-Recv})$$

$$\frac{t_i \longrightarrow t_i'}{v_0.m(\overline{v},\ t_i,\ \overline{t}) \longrightarrow v_0.m(\overline{v},\ t_i',\ \overline{t})} \quad (\text{E-Invk-Arg})$$

$$\frac{t_i \longrightarrow t_i'}{\texttt{new } C(\overline{v},\ t_i,\ \overline{t}) \longrightarrow \texttt{new } C(\overline{v},\ t_i',\ \overline{t})} \quad (\text{E-New-Arg})$$

$$\frac{t_0 \longrightarrow t_0'}{(C)t_0 \longrightarrow (C)t_0'} \qquad (\text{E-Cast})$$

Typing

Notes

FJ has no rule of subsumption (because we want to follow Java). The typing rules are algorithmic.

(Where would this make a difference?...)

Typing rules

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

Typing rules

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \textit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{C}_i} \qquad (\text{T-FIELD})$$

Typing rules

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

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

Typing rules

$$\Gamma \vdash t_0 : C_0$$
$$mtype(m, C_0) = \overline{D} \rightarrow C$$
$$\Gamma \vdash \overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}$$
$$\overline{\Gamma \vdash t_0.m(\overline{t}) : C}$$

$$(\text{T-Invk})$$

Note that this rule "has subsumption built in" — i.e., the typing relation in FJ is written in the *algorithmic* style of chapter 16, not the declarative style of chapter 15.

Java typing is algorithmic

The Java typing relation is defined in the algorithmic style, for (at least) two reasons:

1. In order to perform static *overloading resolution*, we need to be able to speak of "the type" of an expression

2. We would otherwise run into trouble with typing of conditional expressions

Java typing must be algorithmic

We haven't included them in FJ, but full Java has both *interfaces* and *conditional expressions*.

The two together actually make the declarative style of typing rules unworkable!

Java conditionals

$$\frac{t_1 \in \mathtt{bool} \qquad t_2 \in T \qquad t_3 \in T}{t_1 \ ? \ t_2 \ : \ t_3 \in T}$$

Algorithmic version

$$\frac{t_1 \in \mathtt{bool} \qquad t_2 \in T_2 \qquad t_3 \in T_3}{t_1 \ ? \ t_2 \ : \ t_3 \in T_2 \vee T_3}$$

Requires *Joins* !

Java has no joins

But, in full Java (with interfaces), there are types that have no join!

E.g.:

        interface I {...}
        interface J {...}
        interface K extends I,J {...}
        interface L extends I,J {...}

K and L have no join (least upper bound) — both I and J are common upper bounds, but neither of these is less than the other.

Some primitive types also do not have joins (e.g., int and char)

Compilation will fail at such cases.

FJ Typing rules

$$\frac{\begin{array}{cc} \textit{fields}(\mathrm{C}) = \overline{\mathrm{D}}\ \overline{\mathrm{f}} \\ \Gamma \vdash \overline{\mathrm{t}} : \overline{\mathrm{C}} \qquad \overline{\mathrm{C}} <: \overline{\mathrm{D}} \end{array}}{\Gamma \vdash \mathtt{new}\ \mathrm{C}(\overline{\mathrm{t}}) : \mathrm{C}} \qquad (\text{T-New})$$

Typing rules (methods, classes)

$$\frac{\begin{array}{c} \overline{x} : \overline{C}, \text{this} : C \vdash t_0 : E_0 \qquad E_0 <: C_0 \\ CT(C) = \text{class C extends D } \{\ldots\} \\ \textit{override}(m, D, \overline{C} {\rightarrow} C_0) \end{array}}{C_0 \text{ m } (\overline{C} \ \overline{x}) \ \{\text{return } t_0;\} \text{ OK in C}}$$

$$\frac{\begin{array}{c} K = C(\overline{D} \ \overline{g}, \ \overline{C} \ \overline{f}) \ \{\text{super}(\overline{g}); \ \text{this}.\overline{f} = \overline{f};\} \\ \textit{fields}(D) = \overline{D} \ \overline{g} \qquad \overline{M} \text{ OK in C} \end{array}}{\text{class C extends D } \{\overline{C} \ \overline{f}; \ K \ \overline{M}\} \text{ OK}}$$

Properties

Progress

Problem: well-typed programs can get stuck.

How?

Cast failure:

$$(A)\text{new Object}()$$

Formalizing Progress

Solution: Weaken the statement of the progress theorem to

A well-typed FJ term is either a value or can reduce one step or is stuck at a failing cast.

Read the book for the concrete solution.

Preservation

Theorem [Preservation]: If $\Gamma \vdash t : C$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : C'$ for some $C' <: C$.

Proof: Straightforward induction.

???

Preservation?

Surprise: well-typed programs can step to ill-typed ones! (How?)

$$(A)(Object)\underline{new\ B()} \longrightarrow (A)new\ B()$$

Solution: "Stupid Cast" typing rule

Add *another typing rule*, marked "*stupid*" to indicate that an implementation should generate a warning if this rule is used.

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

This is an example of a modeling technicality; not very interesting or deep, but we have to get it right if we're going to claim that the model is an accurate representation of (this fragment of) Java.

Correspondence with Java

Let's try to state precisely what we mean by "FJ corresponds to Java":

Claim:

1. Every syntactically well-formed FJ program is also a syntactically well-formed Java program.

2. A syntactically well-formed FJ program is typable in FJ (without using the T-SCast rule.) iff it is typable in Java.

3. A well-typed FJ program behaves the same in FJ as in Java. (E.g., evaluating it in FJ diverges iff compiling and running it in Java diverges.)

Of course, without a formalization of full Java, we cannot prove this claim. But it's still very useful to say precisely what we are trying to accomplish—e.g., it provides a rigorous way of judging counterexamples. (Cf. "conservative extension" between logics.)

Homework☺

- 18.11.1
- Please submit electronically