编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang
赵海燕，王迪

Peking University, Spring Term 2023

Chap 19: Case Study: Featherweight Java

Completeness or Compactness

Syntax

Evaluation

Typing

Properties

Review: Object-Oriented Programming (OOP)

Summary

OOP principles:

I **Multiple representations**: same interface can have different implementations.
II **Encapsulation**: internal representation is hidden.
III **Subtyping**: Object-interface subtyping enables cross-interface code reusing.
IV **Inheritance**: classes provide a mechanism to organize inheritance-based code reusing.
V **Open recursion**: `self` gets bound during object creation instead of class definition.

Remark (Two Approaches to Defining a Language)

- Embedding in lambda-calculus (previous chapter)
- Formalizing from scratch (this chapter): **treat objects as primitive**

Formalizing

Completeness or Compactness?

"Inside every language is a small language struggling to get out ..."

Formal Modeling

- Describe some aspects precisely.
- Boost the design of real-world artifacts.
- **Completeness**: address more aspects in the model at the same time.
- **Compactness**: try to keep the scale of the model as small as possible.

Principle (Formalizing a Language from Scratch)

We often choose a model that is less complete but more compact.

- Capture the essence as early as possible!
- Extend the model incrementally to improve completeness.

Featherweight Java (FJ)[1]

- FJ is a minimal core calculus for modeling Java's type system.
- The design of FJ favors compactness over completeness.
- The goal in designing FJ is to make its proof of type safety as concise as possible, while still capturing the essence of the safety argument for the central features of full Java.

FJ has had a large impact on programming-language research.

- **Be used directly as a base calculus**, e.g., J. Li et al. 2015. SWIN: Towards Type-Safe Java Program Adaptation between APIs. In *Workshop on Partial Evaluation and Program Manipulation* (PEPM'15), 91–102. DOI: 10.1145/2678015.2682534.
- **Motivate others' design**, e.g., Featherweight Typestate: R. Garcia et al. 2014. Foundations of Typestate-Oriented Programming. *Trans. on Prog. Lang. and Syst.*, 36, 12, 12:1–12:44, 4. DOI: 10.1145/2629609.

[1] A. Igarashi, B. C. Pierce, and P. Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Object-Oriented Prog., Syst., Lang., and Applications* (OOPSLA'99), 132–146. DOI: 10.1145/320385.320395.

An Overview of FJ

An FJ Program is (almost) a Java Program

An FJ program = a set of class definitions + a term to be evaluated.

A Set of Class Definitions

```
class A extends Object { A() { super(); } }
class B extends Object { B() { super(); } }
class Pair extends Object {
  Object fst;
  Object snd;
  // Constructor:
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd; }
  // Method definition:
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); } }
```

Question

Can we embed those definitions in lambda-calculus as we did in the previous chapter?

FJ Terms

Five Forms of Terms

- **Object constructors**: **new** `Pair(...,...)`
- **Method invocations**: `... .setfst(...)`
- **Field accesses**: **this**`.snd`
- **Variables**: `newfst`, **this**
- **Casts**: `(Object)`**new** `Pair(...,...)`

Evaluation

- Everything is an object: values are object creations ($v ::=$ **new** $\mathtt{C}(\overline{v})$).
- No side effects: evaluation is a binary relation on terms ($\mathtt{t} \longrightarrow \mathtt{t}'$).

Examples of FJ Evaluation

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

$$\begin{aligned}
\textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}()).\mathtt{setfst}(\textbf{new}\ \mathtt{B}()) &\longrightarrow \begin{bmatrix} \mathtt{newfst} & \mapsto & \textbf{new}\ \mathtt{B}() \\ \textbf{this} & \mapsto & \textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}()) \end{bmatrix} \\
&\ \textbf{new}\ \mathtt{Pair}(\mathtt{newfst}, \textbf{this}.\mathtt{snd}) \\
&= \textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{B}(), \textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}()).\mathtt{snd}) \\
&\longrightarrow \textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{B}(), \textbf{new}\ \mathtt{B}())
\end{aligned}$$

$$(\mathtt{Object})\textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}()) \longrightarrow \textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}())$$

Question

What's the evaluation result of $((\mathtt{Pair})(\textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{Pair}(\textbf{new}\ \mathtt{A}(), \textbf{new}\ \mathtt{B}()), \textbf{new}\ \mathtt{A}())).\mathtt{fst}).\mathtt{snd}$?

Question

When does an FJ evaluation get stuck?

FJ Types

Structural Type Systems

Recall the **type names** we have seen in the course, e.g., `NatPair = {fst:Nat, snd:Nat}`.

- What matters about a type (for typing, subtyping, etc.) is just its structure.
- Names are just convenient (but inessential) abbreviations.

Nominal Type Systems

However, here in FJ (as well as Java), **type names** play a significant role.

- Types are always named.
- Typechecker mostly manipulates names, not structures.
- Subtyping is declared explicitly by the programmer.

Question

Which style is more popular? Why?

Formalizing FJ

Syntax

$$
\begin{array}{lll}
\mathsf{CL} & ::= & & \textit{class declarations:} \\
 & & \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} & \\
\mathsf{K} & ::= & & \textit{constructor declarations:} \\
 & & \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\textbf{super}(\overline{\mathtt{f}});\ \textbf{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} & \\
\mathsf{M} & ::= & & \textit{method declarations:} \\
 & & \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\textbf{return}\ \mathtt{t};\} & \\
\mathsf{t} & ::= & & \textit{terms:} \\
 & & \mathtt{x} & \textit{variable} \\
 & & \mathtt{t.f} & \textit{field access} \\
 & & \mathtt{t.m}(\overline{\mathtt{t}}) & \textit{method invocation} \\
 & & \textbf{new}\ \mathtt{C}(\overline{\mathtt{t}}) & \textit{object creation} \\
 & & (\mathtt{C})\mathtt{t} & \textit{cast} \\
v & ::= & & \textit{values:} \\
 & & \textbf{new}\ \mathtt{C}(\overline{v}) & \textit{object creation}
\end{array}
$$

Subtyping

PRINCIPLE

For nominal type systems, we usually work with a **global** collection of type names and associated definitions.

Let CT (class table) be a mapping from class names $\mathtt{C}$ to class definitions CL.

Subtyping ($\mathtt{C} <: \mathtt{D}$)

$$\frac{}{\mathtt{C} <: \mathtt{C}} \qquad \frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}} \qquad \frac{CT(\mathtt{C}) = \mathbf{\mathtt{class}}\ \mathtt{C}\ \mathbf{\mathtt{extends}}\ \mathtt{D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

We assume CT does not induce cycles in the subtype relation.

Auxiliary Definitions

PRINCIPLE

Encode each auxiliary function/relation as a system of derivation rules.

Field Lookup ($\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$)

$$\frac{}{\mathit{fields}(\mathtt{Object}) = \bullet}$$

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

Method Type Lookup ($\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{C}$)

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\ldots\} \in \overline{\mathtt{M}} \end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \textbf{class}\ \mathtt{C}\ \textbf{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{m}\ \text{is not defined in}\ \overline{\mathtt{M}} \end{array}}{\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})}$$

Auxiliary Definitions

Method Body Lookup ($mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t})$)

$$\frac{CT(\mathtt{C}) = \mathbf{class}\ \mathtt{C}\ \mathbf{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{M}\} \qquad \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\mathbf{return}\ \mathtt{t};\} \in \overline{M}}{mbody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{t})}$$

$$\frac{CT(\mathtt{C}) = \mathbf{class}\ \mathtt{C}\ \mathbf{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{M}\} \qquad \mathtt{m} \text{ is not defined in } \overline{M}}{mbody(\mathtt{m}, \mathtt{C}) = mbody(\mathtt{m}, \mathtt{D})}$$

Valid Method Overriding ($override(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)$)

$$\frac{mtype(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0}{override(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)}$$

Evaluation ($t \longrightarrow t'$)

$$\frac{\mathit{fields}(C) = \bar{C}\ \bar{f}}{(\mathbf{new}\ C(\bar{v})).f_i \longrightarrow v_i} \quad \text{E-ProjNew}$$

$$\frac{\mathit{mbody}(m, C) = (\bar{x}, t_0)}{(\mathbf{new}\ C(\bar{v})).m(\bar{u}) \longrightarrow [\bar{x} \mapsto \bar{u}, \mathbf{this} \mapsto \mathbf{new}\ C(\bar{v})]t_0} \quad \text{E-InvkNew}$$

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

$$\frac{t_0 \longrightarrow t_0'}{t_0.f \longrightarrow t_0'.f} \quad \text{E-Field}$$

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

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

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

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

Remark

A run-time cast does not change an object.

Typing

Term Typing ($\Gamma \vdash t : C$)

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

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

$$\frac{\mathit{fields}(C) = \bar{D}\ \bar{f} \qquad \Gamma \vdash \bar{t} : \bar{C} \qquad \bar{C} <: \bar{D}}{\Gamma \vdash \mathbf{new}\ C(\bar{t}) : C} \text{ T-NEW} \qquad \frac{\Gamma \vdash t_0 : D \qquad D <: C}{\Gamma \vdash (C)t_0 : C} \text{ T-UCAST}$$

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

Remark (Casts)

There is a **stupid-cast** rule; its only use is to prove type preservation. Consider the evaluation

$$(A)(\mathtt{Object})\mathbf{new}\ B() \longrightarrow (A)\mathbf{new}\ B()$$

On the left is an upcast followed by a downcast, but on the right is a stupid cast.

Typing

Method Typing (M OK in C)

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathbf{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \qquad CT(\mathtt{C}) = \mathbf{class}\ \mathtt{C}\ \mathbf{extends}\ \mathtt{D}\ \{\ldots\} \qquad override(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}_0)}{\mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathbf{return}\ \mathtt{t}_0;\}\ \text{OK in C}}$$

Class Typing (C OK)

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

Properties

THEOREM (PRESERVATION)

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

THEOREM (PROGRESS)

Suppose t is a closed, well-typed normal form. Then either (1) t is a value, or (2) for some evaluation context E, we can express t as $t = E[(C)(\textbf{new}\ D(\overline{v}))]$, with $D \not<: C$.

A **evaluation context** is basically a term with a hole (written $[\,]$):

$$E ::= [\,] \mid E.f \mid E.m(\overline{t}) \mid v.m(\overline{v}, E, \overline{t}) \mid \textbf{new}\ C(\overline{v}, E, \overline{t}) \mid (C)E$$

We write $E[t]$ for the ordinary term obtained by replacing the hole in E with t.

Review: Object-Oriented Programming (OOP)

Summary

OOP principles:

I **Multiple representations**: same interface can have different implementations.

II **Encapsulation**: internal representation is hidden.

III **Subtyping**: Object-interface subtyping enables cross-interface code reusing.

IV **Inheritance**: classes provide a mechanism to organize inheritance-based code reusing.

V **Open recursion**: `self` gets bound during object creation instead of class definition.

Remark (Two Approaches to Defining a Language)

- Embedding in lambda-calculus (previous chapter)
- Formalizing from scratch (this chapter): **treat objects as primitive**

Homework

Question (Exercise 18.11.1)

Use the `fullref` checker to implement the following extensions to the classes above:

1. Rewrite `instrCounterClass` so that it also counts calls to `get`.
2. Extend your modified `instrCounterClass` with a subclass that adds a `reset` method, as in §18.4.
3. Add another subclass that also supports backups, as in §18.7.

Please submit electronically.

Aside

For those who still have not finalized the design of their final project, below are a few ideas:

- Embed a language's core in lambda-calculus: prototype-based OOP, C with **`goto`**, array programming, ...
- Formalize a language's core in a "featherweight" style.
- Compile FJ to lambda-calculus and prove correctness of the compilation.