

Overview

Featherweight Java (FJ), a minimal Java-like language.

- Models inheritance and subtyping.
- Immutable objects: **no mutation** of fields.
- Trivialized core language.

1

Syntax

The syntax of FJ is given by the following grammar:

Classes $C ::= \text{class } c \text{ extends } c' \{ \underline{c\ f}; k\ \underline{d} \}$
Constructors $k ::= c(\underline{c\ x}) \{ \text{super}(\underline{x}); \text{this}.\underline{f=x}; \}$
Methods $d ::= c\ m(\underline{c\ x}) \{ \text{return } e; \}$
Types $\tau ::= c$
Expressions $e ::= x \mid e.f \mid e.m(\underline{e}) \mid \text{new } c(\underline{e}) \mid (c) e$

Underlining indicates "zero or more".

2

Syntax

Classes in FJ have the form:

$\text{class } c \text{ extends } c' \{ \underline{c\ f}; k\ \underline{d} \}$

- Class c is a sub-class of class c' .
- Constructor k for instances of c .
- Fields $\underline{c\ f}$.
- Methods \underline{d} .

3

Syntax

Constructor expressions have the form

$c(\underline{c'\ x'}, \underline{c\ x}) \{ \text{super}(\underline{x}'); \text{this}.\underline{f=x}; \}$

- Arguments correspond to super-class fields and sub-class fields.
- Initializes super-class.
- Initializes sub-class.

4

Syntax

Methods have the form

$c\ m(\underline{c\ x}) \{ \text{return } e; \}$

- Result class c .
- Argument class(es) \underline{c} .
- Binds \underline{x} and **this** in e .

5

Syntax

Minimal set of expressions:

- Field selection: $e.f$.
- Message send: $e.m(e)$.
- Instantiation: $\text{new } c(e)$.
- Cast: $(c) e$.

6

FJ Example

```
class Pt extends Object {
  int x;
  int y;
  Pt (int x, int y) {
    super(); this.x = x; this.y = y;
  }
  int getx () { return this.x; }
  int gety () { return this.y; }
}
```

7

FJ Example

```
class CPt extends Pt {
  color c;
  CPt (int x, int y, color c) {
    super(x,y);
    this.c = c;
  }
  color getc () { return this.c; }
}
```

8

Class Tables and Programs

A **class table** T is a finite function assigning classes to class names.

A **program** is a pair (T, e) consisting of

- A class table T .
- An expression e .

9

Static Semantics

Judgement forms:

$\tau <: \tau'$	<i>subtyping</i>
$c \leq c'$	<i>subclassing</i>
$\Gamma \vdash e : \tau$	<i>expression typing</i>
$d \text{ ok in } c$	<i>well-formed method</i>
$C \text{ ok}$	<i>well-formed class</i>
$T \text{ ok}$	<i>well-formed class table</i>
$\text{fields}(c) = c.f$	<i>field lookup</i>
$\text{type}(m, e) = c \rightarrow c$	<i>method type</i>

10

Static Semantics

Variables:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

- Must be declared, as usual.
- Introduced within method bodies.

11

Static Semantics

Field selection:

$$\frac{\Gamma \vdash e_0 : c_0 \quad \text{fields}(c_0) = c.f}{\Gamma \vdash e_0.f_i : c_i}$$

- Field must be present.
- Type is specified in the class.

12

Static Semantics

Message send:

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash e : e \quad \text{type}(m, c_0) = e' \rightarrow c \quad e <: e'}{\Gamma \vdash e_0.m(e) : c}$$

- Method must be present.
- Argument types must be subtypes of parameter types.

13

Static Semantics

Instantiation:

$$\frac{\Gamma \vdash e : e \quad e <: c \quad \text{fields}(c) = c.f}{\Gamma \vdash \text{new } c(e) : c}$$

- Initializers must have subtypes of field types.

14

Static Semantics

Casting:

$$\frac{\Gamma \vdash e_0 : d}{\Gamma \vdash (c) e_0 : c}$$

- **All** casts are statically acceptable!
- Could try to detect casts that are guaranteed to fail at run-time.

15

Subclassing

Sub-class relation is implicitly relative to a class table.

$$\frac{T(c) = \text{class } c \text{ extends } c' \{ \dots; \dots \}}{c \triangleleft c'}$$

Reflexivity, transitivity of sub-classing:

$$\frac{(T(c) \text{ defined})}{c \triangleleft c} \quad \frac{c \triangleleft c' \quad c' \triangleleft c''}{c \triangleleft c''}$$

Sub-classing **only** by explicit declaration!

16

Subtyping

Subtyping relation: $\tau <: \tau'$.

$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \quad \frac{c \triangleleft c'}{c <: c'}$$

Subtyping is determined **solely** by subclassing.

17

Class Formation

Well-formed classes:

$$\frac{k = c(\underline{c}', \underline{c}x) \{ \text{super}(\underline{c}'); \text{this}.f=x; \} \quad \text{fields}(\underline{c}') = \underline{c}' f' \underline{d} \text{ ok in } c}{\text{class } c \text{ extends } \underline{c}' \{ \underline{c} f; k \underline{d} \} \text{ ok}}$$

- Constructor has arguments for each super- and sub-class field.
- Constructor initializes super-class before sub-class.
- Sub-class methods must be well-formed relative to the super-class.

18

Class Formation

Method overriding, relative to a class:

$$\frac{T(c) = \text{class } c \text{ extends } \underline{c}' \{ \underline{c}_1; \dots \underline{c}_n \} \quad \text{type}(m, \underline{c}') = \underline{c} \rightarrow c_0 \quad \underline{c} : \underline{c}, \text{this}:c \vdash e_0 : \underline{c}'_0 \quad \underline{c}'_0 <: c_0}{c_0 m(\underline{c}x) \{ \text{return } e_0; \} \text{ ok in } c}$$

- Sub-class method must return a subtype of the super-class method's result type.
- Argument types of the sub-class method must be exactly the same as those for the super-class.
- Need another case to cover method extension.

19

Program Formation

A class table is well-formed iff all of its classes are well-formed:

$$\frac{\forall c \in \text{dom}(T) \ T(c) \text{ ok}}{T \text{ ok}}$$

A program is well-formed iff its class table is well-formed and the expression is well-formed:

$$\frac{T \text{ ok} \quad \emptyset \vdash e : \tau}{(T, e) \text{ ok}}$$

20

Method Typing

The type of a method is defined as follows:

$$\frac{T(c) = \text{class } c \text{ extends } \underline{c}' \{ \underline{c}_1; \dots \underline{d} \} \quad d_i = c_i m(\underline{c}_i x) \{ \text{return } e; \}}{\text{type}(m, \underline{c}) = \underline{c}_i \rightarrow c_i}$$

$$\frac{T(c) = \text{class } c \text{ extends } \underline{c}' \{ \underline{c}_1; \dots \underline{d} \} \quad m \notin \underline{d} \quad \text{type}(m, \underline{c}') = \underline{c}_i \rightarrow c_i}{\text{type}(m, \underline{c}) = \underline{c}_i \rightarrow c_i}$$

21

Dynamic Semantics

Transitions: $e \mapsto_T e'$.

Transitions are indexed by a (well-formed) class table!

- Dynamic dispatch.
- Downcasting.

We omit explicit mention of T in what follows.

22

Dynamic Semantics

Object values have the form

$$\text{new } c(\underline{c}', \underline{e})$$

where

- \underline{c}' are the values of the super-class fields.
- \underline{e} are the values of the sub-class fields.
- c indicates the "true" class of the instance.

23

Dynamic Semantics

Field selection:

$$\frac{\text{fields}(c) = c' f', c f \quad e' \text{ val} \quad e \text{ val}}{\text{new } c(e', e) . f_i \mapsto e'_i}$$

$$\frac{\text{fields}(c) = c' f', c f \quad e' \text{ val} \quad e \text{ val}}{\text{new } c(e', e) . f_i \mapsto e_i}$$

- Fields in sub-class must be disjoint from those in super-class.
- Selects appropriate field based on name.

24

Dynamic Semantics

Message send:

$$\frac{\text{body}(m, c) = \underline{x} \rightarrow e_0 \quad e \text{ val} \quad e' \text{ val}}{\text{new } c(e) . m(e') \mapsto [e', \text{new } c(e) / \underline{x}, \text{this}]e_0}$$

- The identifier `this` stands for the object itself.
- Compare with recursive functions in PCF.

25

Dynamic Semantics

Cast:

$$\frac{c \triangleleft c' \quad e \text{ val}}{(c') \text{ new } c(e) \mapsto \text{new } c(e)}$$

- No transition (stuck) if c is not a sub-class of c' !
- Sh/could introduce error transitions for cast failure.

26

Dynamic Semantics

Search rules (CBV):

$$\frac{e_0 \mapsto e'_0}{e_0 . f \mapsto e'_0 . f}$$

$$\frac{e_0 \mapsto e'_0}{e_0 . m(e) \mapsto e'_0 . m(e)}$$

$$\frac{e_0 \text{ val} \quad e \mapsto e'}{e_0 . m(e) \mapsto e_0 . m(e')}$$

27

Dynamic Semantics

Search rules (CBV), cont'd:

$$\frac{e \mapsto e'}{\text{new } c(e) \mapsto \text{new } c(e')}$$

$$\frac{e_0 \mapsto e'_0}{(c) e_0 \mapsto (c) e'_0}$$

28

Dynamic Semantics

Dynamic dispatch:

$$\frac{T(c) = \text{class } c \text{ extends } c' \{ \underline{_}; \dots d \} \quad d_i = c_i m(c_i x) \{ \text{return } e; \}}{\text{body}(m, c) = \underline{x} \rightarrow e}$$

$$\frac{T(c) = \text{class } c \text{ extends } c' \{ \underline{_}; \dots d \} \quad m \notin d \quad \text{body}(m, c') = \underline{x} \rightarrow e}{\text{body}(m, c) = \underline{x} \rightarrow e}$$

- Climbs the class hierarchy searching for the method.
- Static semantics ensures that the method must exist!

29

Type Safety

Theorem 1 (Preservation)

Assume that T is a well-formed class table. If $e : \tau$ and $e \mapsto e'$, then $e' : \tau'$ for some $\tau' \prec \tau$.

- Proved by induction on transition relation.
- Type may get “smaller” during execution due to casting!

30

Type Safety

Lemma 2 (Canonical Forms)

If $e : c$ and e val, then $e = \text{new } c'(e_0)$ with $c' \preceq c$ and e_0 val.

- Values of class type are objects (instances).
- The **dynamic** class of an object may be lower in the subtype hierarchy than the **static** class.

31

Type Safety

Theorem 3 (Progress)

Assume that T is a well-formed class table. If $e : \tau$ then either

1. e val, or
2. e has the form $(c) \text{new } c'(e_0)$ with e_0 val and $c' \not\preceq c$, or
3. there exists e' such that $e \mapsto e'$.

32

Type Safety

Comments on the progress theorem:

- Well-typed programs can get stuck! But only because of a cast
- Precludes “message not understood” error.
- Proof is by induction on typing.

33

Variations and Extensions

A more flexible static semantics for override:

- Subclass result type is a **subtype** of the superclass result type.
- Subclass argument types are **supertypes** of the corresponding superclass argument types.

34

Variations and Extensions

Suppose that $c \preceq c'$ and $o : c$. Then we wish $o : c'$ as well.

Consider $o.m(\underline{e})$, where $\text{type}(m, c) = \underline{d} \rightarrow d$ and $\text{type}(m, c') = \underline{d}' \rightarrow d'$.

- Type of message send is d , and $d \preceq d'$, so of type d' .
- Type of \underline{e} might be \underline{d}' , hence \underline{d} , so message send is OK.

35

Variations and Extensions

Java adds array covariance:

$$\frac{\tau <: \tau'}{\tau [] <: \tau' []}$$

- Perfectly OK for FJ, which does not support assignment.
- With assignment, might store a supertype value in an array of the subtype. Subsequent retrieval at subtype is unsound.
- Java inserts a **per-assignment** run-time check to ensure safety.

36

Variations and Extensions

Static fields:

- Must be initialized as part of the class definition (not by the constructor).
- In what order are initializers to be evaluated? Could require initialization to a constant.

37

Variations and Extensions

Static methods:

- Essentially just recursive functions.
- No overriding.
- Static dispatch to the class, not the instance.

38

Variations and Extensions

Final methods:

- Preclude override in a sub-class.

Final fields:

- Sensible only in the presence of mutation!

39

Variations and Extensions

Abstract methods:

- Some methods are undefined (but are declared).
- Cannot form an instance if any method is abstract.

40

Variations and Extensions

Interfaces:

- Essentially "fully abstract" classes.
- No instances admitted.
- Allow "multiple inheritance." No dispatch ambiguity because no instances!

41

Class Tables

Type checking requires the **entire** program!

- Class table is a global property of the program and libraries.
- Cannot type check classes separately from one another.