510: Programming Languages

Featherweight Java

David Walker

Overview

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

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

Abstract Syntax

The abstract syntax of FJ is given by the following grammar:

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

Underlining indicates "one or more".

Abstract Syntax

Classes in FJ have the form:

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

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

Abstract Syntax

Constructor expressions have the form

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

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

- Initializes super-class.

- Initializes sub-class.

Abstract Syntax

Methods have the form

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

- Result class c.

- Argument class(es) $\underline{c}$.

- Binds $\underline{x}$ and **this** in e.

Abstract Syntax

Minimal set of expressions:

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

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; }
}
```

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; }
}
```

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.

Static Semantics

Judgement forms:

τ <: τ'	*subtyping*
$c \trianglelefteq c'$	*subclassing*
$\Gamma \vdash e : \tau$	*expression typing*
d ok in c	*well-formed method*
c ok	*well-formed class*
T ok	*well-formed class table*
$\mathsf{fields}(c) = \underline{c\,f}$	*field lookup*
$\mathsf{type}(m, c) = \underline{c} \to c$	*method type*

Static Semantics

Variables:

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

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

Static Semantics

Field selection:

$$\frac{\Gamma \vdash e_0 : c_0 \qquad \mathsf{fields}(c_0) = \underline{c\, f}}{\Gamma \vdash e_0 . f_i : c_i}$$

- Field must be present.

- Type is specified in the class.

Static Semantics

Message send:

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \qquad \mathsf{type}(m, c_0) = \underline{c'} \to c \quad \underline{c} \mathrel{<:} \underline{c'}}{\Gamma \vdash e_0 . m(\underline{e}) : c}$$

- Method must be present.

- Argument types must be subtypes of parameters.

Static Semantics

Instantiation:

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \qquad \underline{c} \texttt{ <: } \underline{c'} \qquad \mathsf{fields}(c) = \underline{\underline{c'\,f}}}{\Gamma \vdash \texttt{new}\, c(\underline{e}) : c}$$

- Initializers must have subtypes of fields.

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.

Subclassing

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

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c'\, \{\underline{\ldots};\ \ldots\ \underline{\ldots}\}}{c \trianglelefteq c'}$$

Reflexivity, transitivity of sub-classing:

$$\frac{(T(c)\ \textit{defined})}{c \trianglelefteq c} \qquad \frac{c \trianglelefteq c' \quad c' \trianglelefteq c''}{c \trianglelefteq c''}$$

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

Subtyping

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

$$\frac{}{\tau <: \tau} \qquad \frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}$$

$$\frac{c \trianglelefteq c'}{c <: c'}$$

Subtyping is determined **solely** by subclassing.

Class Formation

Well-formed classes:

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

- 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.

Class Formation

Method overriding, relative to a class:

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{\ldots};\ \ldots\ \underline{\ldots}\} \\ \mathsf{type}(m, c') = \underline{c} \to c_0 \quad \underline{x : c}, \texttt{this}{:}c \vdash e_0 : c_0' \quad c_0' \,\texttt{<:}\, c_0 \end{array}}{c_0\, m\,\texttt{(}\underline{c\, x}\texttt{)}\ \{\texttt{return}\, e_0\texttt{;}\}\ \mathsf{ok\ in}\ c}$$

- 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.

Program Formation

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

$$\frac{\forall c \in \mathsf{dom}(T) \;\; T(c)\,\mathsf{ok}}{T\,\mathsf{ok}}$$

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

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

Method Typing

The type of a method is defined as follows:

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{\ldots;}\ \ldots\ \underline{d}\} \\ d_i = c_i\, m\texttt{(}\underline{c_i\, x}\texttt{)}\,\{\texttt{return}\, e\texttt{;}\} \end{array}}{\mathsf{type}(m_i, c) = \underline{c_i} \to c_i}$$

$$\frac{\begin{array}{c} T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{\ldots;}\ \ldots\ \underline{d}\} \\ m \notin \underline{d} \qquad \mathsf{type}(m_i, c') = \underline{c_i} \to c_i \end{array}}{\mathsf{type}(m, c) = \underline{c_i} \to c_i}$$

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.

Dynamic Semantics

Object values have the form

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

where

- $\underline{e'}$ 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.

Dynamic Semantics

Field selection:

$$\frac{\mathsf{fields}(c) = \underline{c'\,f', c\,f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\,f'_i \mapsto e'_i}$$

$$\frac{\mathsf{fields}(c) = \underline{c'\,f', c\,f} \quad \underline{e'}\ \mathsf{value} \quad \underline{e}\ \mathsf{value}}{\mathtt{new}\, c(\underline{e'}, \underline{e})\,.\,f_i \mapsto e_i}$$

- Fields in sub-class must be disjoint from those in super-class.

- Selects appropriate field based on name.

Dynamic Semantics

Message send:

$$\frac{\mathsf{body}(m, c) = x \to e_0 \quad \underline{e} \text{ value} \quad \underline{e'} \text{ value}}{\texttt{new}\, c\texttt{(}\underline{e}\texttt{)}\texttt{.}m\texttt{(}\underline{e'}\texttt{)} \mapsto \{\underline{e'}/\underline{x}\}\{\texttt{new}\, c\texttt{(}\underline{e}\texttt{)}/\texttt{this}\}e_0}$$

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

Dynamic Semantics

Cast:

$$\frac{c \trianglelefteq c' \quad \underline{e} \text{ value}}{(c')\,\mathtt{new}\,c(\underline{e}) \mapsto \mathtt{new}\,c(\underline{e})}$$

- No transition (stuck) if c is not a sub-class of c'!

- Sh/could introduce error transitions for cast failure.

Dynamic Semantics

Search rules (CBV):

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

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

$$\frac{e_0 \ \mathsf{value} \qquad \underline{e} \mapsto \underline{e'}}{e_0 . m(\underline{e}) \mapsto e_0 . m(\underline{e'})}$$

Dynamic Semantics

Search rules (CBV), cont'd:

$$\frac{\underline{e} \mapsto \underline{e'}}{\texttt{new}\, c(\underline{e}) \mapsto \texttt{new}\, c(\underline{e'})}$$

$$\frac{e_0 \mapsto e_0'}{(c)\; e_0 \mapsto (c)\; e_0'}$$

Dynamic Semantics

Dynamic dispatch:

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{\ldots};\ \ldots\ \underline{d}\} \quad d_i = c_i\, m\texttt{(}\underline{c_i\, x}\texttt{)}\ \{\texttt{return}\, e\texttt{;}\}}{\mathsf{body}(m_i, c) = x \to e}$$

$$\frac{T(c) = \texttt{class}\, c\, \texttt{extends}\, c' \,\{\underline{\ldots};\ \ldots\ \underline{d}\} \quad m \notin \underline{d} \quad \mathsf{body}(m, c') = x \to e}{\mathsf{body}(m, c) = x \to e}$$

- Climbs the class hierarchy searching for the method.

- Static semantics ensures that the method must exist!

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 τ' <: τ.

- Proved by induction on transition relation.

- Type may get "smaller" during execution due to casting!

Type Safety

Lemma 2 (Canonical Forms)
If $e : c$ *and* e value, *then* $e = \texttt{new}\, d(\underline{e_0})$ *with* $d \trianglelefteq c$ *and* e_0 value.

- Values of class type are objects (instances).

- The **dynamic** class of an object may be lower in the subtype hierarchy than the **static** class.

Type Safety

Theorem 3 (Progress)

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

1. v value, *or*

2. e *has the form* $(c)\,\texttt{new}\,d(\underline{e_0})$ *with* e_0 value *and* $d \not\trianglelefteq c$, *or*

3. *there exists* e' *such that* $e \mapsto e'$.

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.

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.

Variations and Extensions

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

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

- Type of message send is d, and $d \trianglelefteq d'$, so of type d'.

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

Variations and Extensions

Java adds array covariance:

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

- 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.

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.

Variations and Extensions

Static methods:

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

Variations and Extensions

Final methods:

- Preclude override in a sub-class.

Final fields:

- Sensible only in the presence of mutation!

Variations and Extensions

Abstract methods:

- Some methods are undefined (but are declared).

- Cannot form an instance if any method is abstract.

Variations and Extensions

Interfaces:

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

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.