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 ::= \text{class } c \text{ extends } c' \{ c_f ; k \ldots \}

- **Constructors** $k ::= c(c_x) \{ \text{super}(c) ; \text{this}.f = x ; \}

- **Methods** $d ::= c.m(c_x) \{ \text{return } e ; \}

- **Types** $\tau ::= c

- **Expressions** $e ::= x | e.f | e.m(c) | \text{new } c(c) | (c) e$

Underlining indicates "one or more".

Abstract Syntax

Classes in FJ have the form:

- $\text{class } c \text{ extends } c' \{ c_f ; k \ldots \}$

- Class $c$ is a sub-class of class $c'$.

- Constructor $k$ for instances of $c$.

- Fields $c.f$.

- Methods $d$.

Abstract Syntax

Methods have the form

- $c.m(c_x) \{ \text{return } e ; \}$

- Result class $c$.

- Argument class(es) $c_x$.

- Binds $x$ and this in $e$.

Abstract Syntax

Constructor expressions have the form

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

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

- Initializes super-class.

- Initializes sub-class.
Abstract Syntax

Minimal set of expressions:

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

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; }
}

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:

- $\tau \leq \tau'$ subtyping
- $c \leq c'$ subclassing
- $\Gamma \vdash e : \tau$ expression typing
- $d\in \Gamma \vdash c$ well-formed method
- $C\in \Gamma \vdash c$ well-formed class
- $T\in \Gamma \vdash c$ well-formed class table
- $fields(c) = c\in \Gamma$ field lookup
- $type(m, c) = c : \tau \rightarrow c$ method type

Variables:

- $\Gamma(x) = \tau$

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

Field selection:
\[ \Gamma \vdash \sigma_0: \sigma_0 \quad \text{fields}(\sigma_0) = \epsilon_f \]
\[ \Gamma \vdash \sigma_0.\sigma_i: \sigma_i \]

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

Instantiation:
\[ \Gamma \vdash \epsilon: \sigma \quad \epsilon: \epsilon' \quad \text{fields}(\epsilon) = \epsilon' \]
\[ \Gamma \vdash \text{new}(\epsilon): \epsilon \]

- Initializers must have subtypes of fields.

Casting:
\[ \Gamma \vdash \epsilon_0: \delta \quad \Gamma \vdash \epsilon: \epsilon' \quad \text{type}(m, \epsilon_0) = \epsilon' \rightarrow \epsilon \quad \epsilon: \epsilon' \]
\[ \Gamma \vdash \epsilon_0.\epsilon(m(\epsilon)): \epsilon \]

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

Subtyping

Subtyping relation is determined solely by subclassing.
\[ \tau \leq \tau' \]
\[ \tau \subseteq \tau' \]
\[ \tau' \subseteq \tau'' \]

Reflexivity, transitivity of sub-classing:
\[ (T(c) \text{ defined}) \]
\[ \tau \leq \tau' \]
\[ \tau' \leq \tau'' \]
\[ \tau \leq \tau'' \]

Subtyping is determined solely by subclassing.

Subclassing

Sub-class relation is implicitly relative to a class table.
\[ T(c) = \text{class}\ c \text{ extends } c' \{ \ldots \} \]

Sub-classing only by explicit declaration!
Class Formation

Well-formed classes:

\[ k = \epsilon(c', c) \mid \text{super}(c') : \text{this} \in c \]

\[ \text{fields}(c') = c' \mid \text{ok} \]

\[ \text{class} \in \text{extends} (c') \mid \text{ok} \]

- 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.

Program Formation

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

\[ \forall c \in \text{dom}(T), \quad T(c) \text{ ok} \]

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

\[ T \text{ ok}, \quad \emptyset \vdash e : \tau \]

\[ T(e) \text{ ok} \]

Method Typing

The type of a method is defined as follows:

\[ T(c) = \text{class} \in \text{extends} (c') \mid \text{ok} \]

\[ T(c) = \text{type} (m, c') = c_i \rightarrow c_1 \]

\[ \text{type}(m, c') = c_i \rightarrow c_1 \]

Dynamic Semantics

Transitions: \( e \rightarrow_T e' \).

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

\[ \text{new} (c', \xi) \]

where

- \( c' \) are the values of the super-class fields.

- \( \xi \) are the values of the sub-class fields.

- \( c \) indicates the “true” class of the instance.
Dynamic Semantics

Field selection:
\[
\begin{align*}
\text{fields}(c) &= c' \quad \text{if } c = c' \\
\text{new } c &\Rightarrow \text{new } c' \\
b &\Rightarrow b' \\
b \in (c) &\Rightarrow (c) b' \\
b &\Rightarrow b' \\
b &\Rightarrow b \\
e b &\Rightarrow e' \\
e &\Rightarrow e \\
e b &\Rightarrow e' \\
\text{new } c &\Rightarrow \text{new } c' \\
\text{new } e &\Rightarrow \text{new } e' \\
\end{align*}
\]

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

Dynamic Semantics

Cast:
\[
\begin{align*}
e &\triangleleft c' \\
\text{new } c &\Rightarrow \text{new } c'
\end{align*}
\]

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

Dynamic Semantics

Search rules (CBV), cont’d:
\[
\begin{align*}
e &\Rightarrow c' \\
\text{new } c &\Rightarrow \text{new } c' \\
e_0 &\Rightarrow e_0' \\
(c) &\Rightarrow (c) e_0' \\
\end{align*}
\]

Dynamic Semantics

Dynamic dispatch:
\[
\begin{align*}
T(c) &= \text{class } c \text{ extends } c' \{ \ldots \} \\
d_i &= c_i \text{ if } c \text{ } \{ \text{return } c; \} \\
\text{body}(m, c) &= \text{if } c \text{ } \{ \text{body}(m, c'); \} \\
\end{align*}
\]

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

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 $\tau' \subseteq \tau$.

- 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 $e = \text{new } c'(e_0)$ with $c' \subseteq c$ 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.

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

Suppose that $c \subseteq c'$ and $a : c$. Then we wish $a : c'$ as well.

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

- Type of message send is $d$, and $d \subseteq d'$, so of type $d'$.
- Type of $e$ might be $d'$, hence $d$, so message send is OK.
Variations and Extensions

Java adds array covariance:

\[
\tau <: \tau' \\
\tau[\cdot] <: \tau'[\cdot]
\]

- Perfectly OK for \(xJ\) 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.

Variations and Extensions

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.