Overview

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

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

Syntax

The syntax of FJ is given by the following grammar:

Classes

\[ C ::= \text{class} \cdot \text{extends} \cdot C' \{ f_j; k_d \} \]

Constructors

\[ k ::= \text{c}(e) \{ \text{super}(e); \text{this}, f_k; e \} \]

Methods

\[ d ::= \text{cm}(e) \{ \text{return} e; \} \]

Types

\[ e ::= x | e.m(e) | \text{new} c(e) | (e) \]

Expressions

\[ e ::= x | e.f | e.m | \text{new} c(e) | (e) \]

Underlining indicates “zero or more”.

Syntax

Classes in FJ have the form:

\[ \text{class} \cdot \text{extends} \cdot C' \{ f_j; k_d \} \]

- Class \( C \) is a sub-class of class \( C' \).
- Constructor \( k \) for instances of \( C \).
- Fields \( f_j \).
- Methods \( d \).

Syntax

Constructor expressions have the form

\[ c(l', e); \{ \text{super}(l'); \text{this}, f_k; e \} \]

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

Methods have the form

\[ cm(e) \{ \text{return} e; \} \]

- Result class \( e \).
- Argument class(es) \( c \).
- Binds \( x \) and this in \( e \).
Syntax

Minimal set of expressions:

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

- \( \tau <: \tau' \)  subtyping
- \( e <: e' \)  subclasing
- \( \Gamma \vdash e : \tau \)  expression typing
- \( \text{dok} \ e \)  well-formed method
- \( C \text{ok} \)  well-formed class
- \( T \text{ok} \)  well-formed class table
- \( \text{fields}(c) = e \)  field lookup
- \( \text{type}(m, c) \Rightarrow \tau 
\rightarrow e \)  method type

Static Semantics

Variables:

- \( \Gamma(x) = \tau \)
- \( \Gamma(x) : \tau \)

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

Field selection:
\[
\Gamma \vdash e : c_0 \\
\text{fields}(c_0) = e.f \\
\Gamma \vdash e_0 : e_i \\
\]

• Field must be present.

• Type is specified in the class.

Message send:
\[
\Gamma \vdash e : c_0 \\
\Gamma \vdash e : c \\
\text{type}(m, c_0) = d' \rightarrow c \\
\Gamma \vdash e_0 : m(c_0) : e \\
\]

• Method must be present.

• Argument types must be subtypes of parameter types.

Instantiation:
\[
\Gamma \vdash e : c \\
\Gamma \vdash e : c_0 \\
\text{fields}(c) = e.f \\
\Gamma \vdash \text{new}(c_0) : c \\
\]

• Initializers must have subtypes of field types.

Casting:
\[
\Gamma \vdash e : d \\
\Gamma \vdash e_0 : c \\
\]

• All casts are statically acceptable.

• Could try to detect casts that are guaranteed to fail at runtime.

Subclassing

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

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

Sub-classing only by explicit declaration!

Subtyping

Subtyping relation: \( \tau \subseteq \tau' \).
\[
\tau \subseteq \tau' \\
\tau' \subseteq \tau'' \\
\]

Subtyping is determined solely by subclassing.
Class Formation

Well-formed classes:

\[ k = c(x', x) \{ \text{super}(x'); \text{this}, \text{sub}(x) \} \]

\[ \text{fields}(x') = x' \}

\[ \text{super} = x' \}

\[ \text{this} = x \}

\[ \text{sub} = x \}

- 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 table 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} \}

\[ \theta \vdash e : \tau \]

\[ (T, \theta) \text{ ok} \]

Method Typing

The type of a method is defined as follows:

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots \} \]

\[ d_i = c_i \text{ (c, e) } \{ \text{return } e_i \} \]

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

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

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

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

- 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.

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.

Object values have the form \( \text{new } c(x', x) \)

where

- \( x' \) are the values of the super-class fields.
- \( x \) are the values of the sub-class fields.
- \( c \) indicates the “true” class of the instance.
Dynamic Semantics

Field selection:

\[ \text{fields}(c) = \text{f} \mapsto f \]

\[ \text{new}(\text{c'}, e) = f' \]

\[ \text{new}(\text{c'}, e) = f' \]

\[ \text{new}(\text{c'}, e) = f' \]

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

Dynamic Semantics

Message send:

\[ \text{body}(m, c) = \text{e} \rightarrow c \]

\[ \text{new}(\text{c'}, e) = f' \]

\[ \text{new}(\text{c'}, e) = f' \]

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

Dynamic Semantics

Cast:

\[ \text{c} \not\subseteq \text{c'} \rightarrow \text{e} \]

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

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

Dynamic Semantics

Search rules (CBV):

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, f = \text{e}_0, f' \]

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, m(c') = \text{e}_0, m(c) \]

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, m(c') = \text{e}_0, m(c) \]

Dynamic Semantics

Search rules (CBV), cont'd:

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, f = \text{e}_0, f' \]

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, m(c) = \text{e}_0, m(c) \]

\[ \text{e}_0 = \text{e}' \]

\[ \text{e}_0, m(c) = \text{e}_0, m(c) \]

Dynamic Semantics

Dynamic dispatch:

\[ T(c) = \text{class } c \text{ extends } \{ \ldots \} \]

\[ T(c) = \text{class } c \text{ extends } \{ \ldots \} \]

\[ \text{body}(m, e) = \text{e} \]

\[ \text{body}(m, c') = \text{e} \]

- 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 $e \rightarrow e'$, then $e' : \tau'$ for some $\tau' < \tau$.

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

Lemma 2 (Canonical Form)
If $e : c$ and $e$ val, then $e = \text{new} c(e_0)$ with $e_0 \subseteq c$ 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.

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 $(\text{new} c(e_0))$ with $e_0$ val and $\tau' \not> \tau$, or

3. there exists $e'$ such that $e \rightarrow e'$.

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.

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

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

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

Java adds array covariance:
\[ \tau[] <: \tau'[] \]

- 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.

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.