Type Systems

Lecture 7 Dec. 1st, 2004

Sebastian Maneth

http://lampwww.epfl.ch/teaching/typeSystems/2004

Today Featherweight Java

1. Recall Syntax of FJ
2. Static Semantics
3. Dynamic Semantics (Evaluation)
4. Type Safety
5. Extensions

Many of today's slides come from

→ CS510 (2003 at Princeton by D.Walker)
→ CMPSCI530 (2004/2002 at UMass Amherst by R. Harper)

1. Recall Syntax of 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; }
}
```

1. Recall Syntax of 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; }
}
```

1. Recall Syntax of 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 int extends Object { int() { super(); } }
class color extends Object { color() { super(); } }
```

1. Recall Syntax of FJ

```
Classes        C ::=  class C extends D { C f; K M }
Constructors   K ::=  C (C x) { super(x); this.f=x; }
Methods        M ::=  C m (C x) { return t; }
Terms          t ::=    x
                      | t.f
                      | t.m(t)
                      | new C(t)
                      | (C) t
```

<u>Underlining</u> indicates a sequence of arbitrary length ($\geq$ 0)

1. Recall Syntax of FJ

Objects are immutable: **no mutatation** of fields!

(→ cannot do a 'set method')

```
FJ Program = ( CT, t )
```

```
CT: class table
              (e.g., CT(int)=class int extends .. )

t: term to be evaluated
```

2. Static Semantics

Judgement forms:

A <: B	subtyping
$\Gamma \vdash$ t : C	term typing
m ok in C	well-formed method
C ok	well-formed class
T ok	well-formed class table
fields(C) = $\underline{\text{C f}}$	field lookup
mtype(m,C) = $\underline{\text{C}} \rightarrow$ C	method type lookup

2. Static Semantics

Subtyping

Subtype relation <: determined by CT only!

```
        CT(C) = class C extends D { … }
       ----------------------------------
                    C :< D


reflexive       C <: C

transitive      C <: D      D :< E
               --------------------
                     C :< E
```

2. Static Semantics

Environment Γ is mapping from variables to types (classes).

Variables can only appear in method bodies.

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

→ Variables must be declared

2. Static Semantics

Field selection:

$$\frac{\Gamma \vdash t_0 : C_0 \qquad fields(C_0) = \underline{C\ f}}{\Gamma \vdash t_0.f_i : C_i}$$

→ field f_i must be present in C_0

→ its type is specified in C_0

2. Static Semantics

Method invocation (message send):

$$\frac{\Gamma \vdash t_0 : C_0 \qquad mtype(m, C_0) = \underline{C'} \rightarrow D \qquad \Gamma \vdash \underline{t} : \underline{C} \qquad \underline{C} <: \underline{C'}}{\Gamma \vdash t_0.m(\underline{t}) : D}$$

→ method must be present

→ argument types must be subtypes of parameters

2. Static Semantics

Instantiation (object creation):

$$\frac{\Gamma \vdash \underline{\mathtt{t}}:\underline{\mathtt{C}} \qquad \underline{\mathtt{C}}<:\underline{\mathtt{C'}} \qquad \mathtt{fields(D)} = \underline{\mathtt{C'\ f}}}{\Gamma \vdash \mathtt{new\ D(}\underline{\mathtt{t}}\mathtt{)\ :\ D}}$$

→ class name must exists

→ initializers must be of subtypes of fields

2. Static Semantics

Casting:

(up or down)

$$\frac{\Gamma \vdash t_0 : C \qquad (C <: D \text{ or } D <: C)}{\Gamma \vdash (D)t_0 : D}$$

→ ALL casts (up/down) are statically acceptable!

→ stupid (side) casts can be detected:

$$\frac{\Gamma \vdash t_0 : C \qquad \text{not}(D <: C \text{ or } D <: D) \qquad \textit{give warning!}}{\Gamma \vdash (D)t_0 : D}$$

Why do we allow down-casts?

Needed for applying class-specific methods, e.g.:

```
((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd
```

→ At run-time, only up-casts will succeed.

Without the cast, typing of term fails:

$$\frac{}{\texttt{(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj}}$$

Without the cast, typing of term fails:

$$\frac{\Gamma \vdash t_0 : C_0 \qquad \text{fields}(C_0) = \underline{C\ f}}{\Gamma \vdash t_0.f_i : C_i}$$

```
new Pair(new Pair(new A(), new B()),          fields(Pair) =
        new A()).fst : Pair                          Obj fst, Obj snd
--------------------------------------------------------------------------
 (new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
```

2. Static Semantics

Without the cast, typing of term fails:

$$\frac{\Gamma \vdash t_0 : C_0 \qquad fields(C_0) = \underline{C\ f}}{\Gamma \vdash t_0.f_i : C_i}$$

```
new Pair(new Pair(new A(), new B()),        fields(Pair) =
         new A()) : Pair                         Obj fst, Obj snd
-----------------------------------------------------------------
new Pair(new Pair(new A(), new B()),        fields(Pair) =
         new A()).fst : Pair                     Obj fst, Obj snd
-----------------------------------------------------------------
(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
```

With the cast typing succeeds!

$$\frac{\Gamma \vdash t_0 : C_0 \qquad fields(C_0) = \underline{C\ f}}{\Gamma \vdash t_0.f_i : C_i}$$

$$\frac{\texttt{(Pair) new Pair(new Pair(new A(), new B()), new A()).fst : Pair} \qquad \texttt{fields(Pair) = Obj fst, Obj snd}}{\texttt{(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj}}$$

(Pair)

With the cast typing succeeds!

```
new Pair(new Pair(new A(), new B()),
         new A()).fst : Obj                Pair <: Obj
-----------------------------------------------------------------
(Pair) new Pair(new Pair(new A(), new B()),    fields(Pair) =
         new A()).fst : Pair                   Obj fst, Obj snd
-----------------------------------------------------------------
 (new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
  \
   (Pair)
```

With the cast typing succeeds!

$$\frac{\Gamma \vdash \underline{t}:\underline{C} \qquad \underline{C}<:\underline{C'} \qquad \texttt{fields(D)} = \underline{C'\ f}}{\Gamma \vdash \texttt{new D(}\underline{t}\texttt{) : D}}$$

```
new Pair(new Pair(new A(), new B()),        fields(Pair) =
         new A()) : Pair                      Obj fst, Obj snd
-------------------------------------------------------------------
new Pair(new Pair(new A(), new B()),
         new A()).fst : Obj                   Pair <: Obj
-------------------------------------------------------------------
(Pair) new Pair(new Pair(new A(), new B()),   fields(Pair) =
         new A()).fst : Pair                  Obj fst, Obj snd
-------------------------------------------------------------------
(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
  \
   (Pair)
```

With the cast typing succeeds!

```
new Pair(new A(), new B()) : Pair                  Pair <: Obj
new A() : A                                           A <: Obj
-------------------------------------------------------------------
new Pair(new Pair(new A(), new B()),    fields(Pair) =
         new A()) : Pair                   Obj fst, Obj snd
-------------------------------------------------------------------
new Pair(new Pair(new A(), new B()),
         new A()).fst : Obj                 Pair <: Obj
-------------------------------------------------------------------
(Pair) new Pair(new Pair(new A(), new B()),    fields(Pair) =
         new A()).fst : Pair                   Obj fst, Obj snd
-------------------------------------------------------------------
(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
 \
  (Pair)
```

With the cast typing succeeds!

```
new Pair(new A(), new B()) : Pair                          Pair <: Obj
new A() : A     OK, because fields(A) = [ ]                   A <: Obj
-----------------------------------------------------------------------
new Pair(new Pair(new A(), new B()),      fields(Pair) =
         new A()) : Pair                     Obj fst, Obj snd
-----------------------------------------------------------------------
new Pair(new Pair(new A(), new B()),
         new A()).fst : Obj                  Pair <: Obj
-----------------------------------------------------------------------
(Pair) new Pair(new Pair(new A(), new B()),    fields(Pair) =
         new A()).fst : Pair                   Obj fst, Obj snd
-----------------------------------------------------------------------
(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
  \
   (Pair)
```

With the cast typing succeeds!

```
new A(): A             A <: Obj          fields(Pair) =
new B(): B             B <: Obj             Obj fst, Obj snd
--------------------------------------------------------------
new Pair(new A(), new B()) : Pair               Pair <: Obj
new A() : A     OK, because fields(A) = [ ]        A <: Obj
--------------------------------------------------------------
new Pair(new Pair(new A(), new B()),     fields(Pair) =
         new A()) : Pair                    Obj fst, Obj snd
--------------------------------------------------------------
new Pair(new Pair(new A(), new B()),
         new A()).fst : Obj                 Pair <: Obj
--------------------------------------------------------------
(Pair) new Pair(new Pair(new A(), new B()),     fields(Pair) =
         new A()).fst : Pair                    Obj fst, Obj snd
--------------------------------------------------------------
(new Pair(new Pair(new A(), new B()), new A()).fst).snd : Obj
 \
  (Pair)
```

2. Static Semantics

Well-Formed Classes

```
K = C(D g, C f) { super(); this.f = f; }
fields(D) = D g        M ok in C
-------------------------------------------
 Class C extends D { C f; K M } ok
```

→ constructor has arguments for all super-class fields
and for all new fields

→ initialize super-class before new fields

→ new methods must be well-formed

2. Static Semantics

Well-Formed Methods

$$\frac{CT(C) = \texttt{class } C \texttt{ extends } D \{ \ldots \} \quad mtype(m,D) \texttt{ equals } \underline{C} \rightarrow C_0 \texttt{ or undefined} \quad \underline{x:C}, this:C \vdash t_0 : E_0 \qquad E_0 <: C_0}{C_0 \; M \; (\underline{C \; x}) \; \{ \texttt{return } t_0; \} \quad \texttt{ok in } C}$$

→ must return a subtype of the result type

→ if overriding, then type of method must be same as before

2. Static Semantics

Well-Formed Class Table

$$\frac{\texttt{for all C} \in \texttt{dom(CT), T(C) ok}}{\texttt{CT ok}}$$

→ All classes in CT must be well-formed

Well-Formed Program

$$\frac{\texttt{CT ok} \qquad \vdash \texttt{t : C}}{\texttt{(CT, t) ok}}$$

2. Static Semantics

Method Type Lookup

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

$$\frac{\mathtt{CT(C) = class\ C\ extends\ D\ \{\ \underline{C\ f};\ K\ \underline{M}\ \}} \qquad \mathtt{m\ not\ defined\ in}\ \underline{\mathtt{M}}}{\mathtt{mtype(m,C) = mtype(m,D)}}$$

Method Body Lookup works exactly the same.

→ returns $(\underline{\mathtt{x}}, \mathtt{t})$

2. Static Semantics

Field Lookup

$$\texttt{fields(Object) = [ ]}$$

$$\frac{\texttt{CT(C) = class C extends D \{ }\underline{\texttt{C f}}\texttt{; K }\underline{\texttt{M}}\texttt{ \}} \qquad \texttt{fields(D) = }\underline{\texttt{D g}}}{\texttt{fields(m,C) = }\underline{\texttt{D g}}\texttt{, }\underline{\texttt{C f}}}$$

→ Concatenation of super-class fields, plus new ones

3. Dynamic Semantics (Evaluation)

Object values have the form new c($\underline{s}$,$\underline{t}$)

where $\underline{s}$ are the values of super-class fields and $\underline{t}$ are the values of C's fields.

field selection

$$\frac{\text{fields(C)} = \underline{\text{C f}}}{\text{(new C}(\underline{v})\text{).}f_i \rightarrow v_i}$$

method invocation

$$\frac{\text{mbody(m,C)} = (\underline{x}, t_0)}{\text{(new C}(\underline{v})\text{).m}(\underline{u}) \rightarrow [x \rightarrow u,\ \text{this} \rightarrow \text{new C}(\underline{v})]\ t_0}$$

casting

$$\frac{\text{C <: D}}{\text{(D)(new C}(\underline{v})) \rightarrow \text{new C}(\underline{v})}$$

3. Dynamic Semantics (Evaluation)

Object values have the form new $c(\underline{s},\underline{t})$

where $\underline{s}$ are the values of super-class fields and $\underline{t}$ are the values of C's fields.

$$\frac{\texttt{fields(C)} = \underline{\texttt{C f}}}{\texttt{(new C(}\underline{\texttt{v}}\texttt{)).f}_i \rightarrow \texttt{v}_i}$$

field selection

$$\frac{\texttt{mbody(m,C)} = (\underline{\texttt{x}},\texttt{t}_0)}{\texttt{(new C(}\underline{\texttt{v}}\texttt{)).m(}\underline{\texttt{u}}\texttt{)} \rightarrow [\texttt{x}\rightarrow\texttt{u, this}\rightarrow\texttt{new C(}\underline{\texttt{v}}\texttt{)]}\ \texttt{t}_0}$$

method invocation

$$\frac{\texttt{C <: D}}{\texttt{(D)(new C(}\underline{\texttt{v}}\texttt{))} \rightarrow \texttt{new C(}\underline{\texttt{v}}\texttt{)}}$$

stuck, if C is not a subtype of D!!!

casting

3. Dynamic Semantics (Evaluation)

Object values have the form new $\texttt{c(}\underline{\texttt{s}}\texttt{,}\underline{\texttt{t}}\texttt{)}$

where $\underline{\texttt{s}}$ are the values of super-class fields
and $\underline{\texttt{t}}$ are the values of C’s fields.

$$\frac{\texttt{fields(C) = }\underline{\texttt{C f}}}{\texttt{(new C(}\underline{\texttt{v}}\texttt{)).f}_i \rightarrow \texttt{v}_i}$$

$$\frac{\texttt{mbody(m,C) = (}\underline{\texttt{x}}\texttt{,t}_0\texttt{)}}{\texttt{(new C(}\underline{\texttt{v}}\texttt{)).m(}\underline{\texttt{u}}\texttt{)} \rightarrow \texttt{[x}\rightarrow\texttt{u, this}\rightarrow\texttt{new C(}\underline{\texttt{v}}\texttt{)] t}_0}$$

$$\frac{\texttt{C <: D}}{\texttt{(D)(new C(}\underline{\texttt{v}}\texttt{))} \rightarrow \texttt{new C(}\underline{\texttt{v}}\texttt{)}}$$

… plus usual CBV evaluation rules!

3. Dynamic Semantics (Evaluation)

Method Body Lookup

$$\frac{CT(C) = \texttt{class C extends D \{ } \underline{\texttt{C f}}\texttt{; K } \underline{\texttt{M}} \texttt{ \}} \quad \texttt{B m (}\underline{\texttt{B x}}\texttt{) \{ return t; \}} \in \underline{\texttt{M}}}{mbody(m,C) = (\underline{x},t)}$$

$$\frac{CT(C) = \texttt{class C extends D \{ } \underline{\texttt{C f}}\texttt{; K } \underline{\texttt{M}} \texttt{ \}} \quad \texttt{m not defined in } \underline{\texttt{M}}}{mbody(m,C) = mbody(m,D)}$$

→ “Dynamic Dispatch” - climbs up the class hierarchy searching for the method

→ static semantics guarantees that method exists!

Easy Questions:

1. How can you (Church-) encode Booleans in FJ?
2. What is the smallest nonterminating FJ program?
3. Why is FJ Turing complete?
4. Why can casts not be (fully) statically checked?

4. Type Safety

Theorem (Preservation)

Let CT be a well-formed class table.

If `t:C` and `t→t'` then `t':C'` for some `C'<:C`.

→ Proof by induction on the length of evaluations.

→ Type may get “smaller” during execution, due to casting!

how?

4. Type Safety

Canonical Forms Lemma.

If `v:C`, then `v = new D(`t_0`)` with `D<:C` and t_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.

4. Type Safety

Theorem (Progress)

Let CT be a well-formed class table.
If `t:C` then either

1. `t` is a value, or
2. `t = (C) new D(`v_0`)` and not(`D <: C`), or
3. there exists `t'` such that `t` → `t'`.

→ Proof by induction on typing derivations.

→ Well-typed programs CAN GET STUCK!! But only because of casts..

→ Precludes “message not understood” error.

5. Extensions

Which static type check can we easily generalize?

5. Extensions

Which static type check can we easily generalize?
→ Method Override!

Well-Formed Methods

$$\frac{CT(C) = \texttt{class } C \texttt{ extends } D \{ \ldots \} \quad mtype(m,D) \text{ equals } \underline{C} \to C_0 \text{ or undefined} \quad \underline{x:C}, this:C \vdash t_0 : E_0 \quad E_0 <: C_0}{C_0 \; M \; (\underline{C\ x}) \; \{ \texttt{return } t_0; \} \;\; \text{ok in } C}$$

→ must return a subtype of the result type

→ if overriding, then type of method must be same as before

5. Extensions

A more flexible static semantics of override:

→ result type is subtype of superclass result type

→ argument types are supertypes of the corresponding superclass argument types.

just as for functions! covariant in result, contravariant in argument.

5. Extensions

Why does this work out?

Assume C <: C′ and t_0:C. We want that also t_0:C′.

mtype(m, C) = D → D
mtype(m, C′) = D′ → D′

Consider t_0.m(t)

→ Type of message send is D and D<:D′, so of type D′.

→ Type of t might be D′, hence D, so message send is OK.

5. Extensions

Java adds array covariance:

```
     C <: D
-----------------
 C [] <: D []
```

→ No problem for FJ, which does not support assignment.

→ With assignment, might store a supertype value in an array of the subtype. Subsequent retrival at suptype unsound!

→ Java inserts a per-assignment run-time check to ensure safety

5. Extensions

Static Fields:

→ Must be initialized as part of the class definition (not by the constructor)

→ In what order are initializers evaluated? – could require initialization to a constant.

Static Methods:

→ Essentially just recursive functions

→ no overriding

→ static dispatch to the class, no the instance.

5. Extensions

Final Methods:

→ Preclude override in a subclass

Final Fields:

→ Only sensible in the presence of mutation!

Abstract Methods:

→ Some methods are undefined (but declared)

→ Cannot form an instance if any method is abstract

5. Extensions

Interfaces:

→ Essentially “fully abstract” classes

→ No instances admitted

→ Allow “multiple inheritance”. No dispatch ambiguity because no instance!

5. Extensions

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 another