Type Systems

Lecture 11   Jan. 12th, 2005

Sebastian Maneth

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

Today FGJ = FJ + Generics

1. Intro to Generics
2. Syntax of FGJ
3. Static Semantics
4. Dynamic Semantics
5. Type Safety
6. Erasure Semantics

The Course

24.11.	FJ	FJ	132+12
1.12.			
8.12.	Polymorphism		
15.12.		lab	
22.12.		lab ← Written Assignment 100+60	
12.1.	FGJ	FGJ	
19.1.	Scala		
26.1.			132+40
2.2.			

Total: 364 (+112)

Your grade = (EX grade + oral exam grade) / 2

A Critique of Statically Typed PLs

- → Types are obtrusive: they overwhelm the code
  - ➔ Type Inference (Reconstruction)
- → Types inhibit code re-use: one version for each type.
  - ➔ Polymorphism

What is Polymorphism?

Generally: Idea that an operation can be applied to values of different types. ('poly'='many')

Can be achieved in many ways..

According to Strachey (1967, "Fundamental Concepts in PLs") and Cardelli/Wegner (1985, survey)

- polymorphism
  - Universal (true)
    - parametric
    - inclusion
  - Ad hoc (apparent)
    - overloading
    - coercion

Universal Polymorphism

Inclusion = Subtype Polymorphism

→ One object belongs to many classes.
E.g., a colored point
can be seen as a point.

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

Parametric Polymorphism

→ Use Type Variables

$$f = \lambda x:\ X\ .\lambda y:\ Y\ .\ x(x(y))$$

Y→Y	Y

"principal type" of $f = \lambda x.\lambda y.\ x(x(y))$

Universal Polymorphism

Combination of

Subtype Polymorphism and
Parametric Polymorphism

→ Based on lambda-calculus: System F-sub

```
λX<:{a:Nat}. λx:X. {orig=x, asucc=succ(x.a)};
```

→ Based on Featherweight Java (FJ): FGJ

FJ

```
class A extends Object { A(){super();} }
class B extends Object { B(){super();} }

class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

FJ + generic type parameters (generics)

```
class A extends Object { A(){super();} }
class B extends Object { B(){super();} }

class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);
  }
}
```

FJ + generic type parameters (generics)

```
class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst = fst; this.snd = snd;
}
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);}}
```

→ Classes AND methods may have generic type param's:

X, Y: type parameters of class Pair
Z: type parameter of method setfst

→ Each type parameter has a *bound*.

here: X,Y,Z all have bound Object

```
class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
…   }
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);}}
```

Instantiation of class/method:

→ concrete types must be supplied

```
new Pair<A,B>(new A(), new B()).setfst<B>(new B())
```

```
class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
…  }
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);}}
```

Instantiation of class/method:

→ concrete types must be supplied

```
new Pair<A,B>(new A(), new B()).setfst<B>(new B())
```

Evaluates to:

```
new Pair<B,B>(new B(), new B())
```

→ In GJ (Java), type parameters to generic method invocations are inferred!

Thus, the <B> in the invocation of setfst is NOT needed!

```
new Pair<A,B>(new A(), new B()).setfst(new B())
```

Why is this possible?

→ Type of a term is *local*: only depends on types of subterms, and not on context!

(for more info, see [Bracha/Odersky/Stoutamire/Wadler1998])

Notes:

→ Generic types can be simulated in Java (FJ) already:

a collection with elements of ANY type

is represented by

a collection with elements of type Object.

The "Generic Idiom"

MAIN MERIT
of adding direct support of generics:

→ LESS casts needed by programmer!!

(and, casts inserted by compilerer canNOT go wrong!)

GJ (Java) and the "Generic Legacy Problem"

→ What to do with all the code based on the generic idiom?

e.g. change type Collection into Collection<X>?

But don't want/can't change old code..

➔ GJ proposes "raw types".

A parametric type Collection<X> may be passed wherever the corresponding raw type Collection is expected.

Example:

Recall that

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

Does NOT type check! (cast is needed!)

With generics, we could write

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

.. which (should) type check..

Syntax of FJ

```
Conventions:   write A instead of A<>
                     B instead of B<> …

               write ◀ instead of extends
```

Syntax of FJ

```
Classes        C ::=  class C◀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
```

2. Syntax of FGJ

List of type parameters w bounds

```
Classes        C ::=  class C<X◀N>◀D<T> { C f; K M }
Constructors   K ::=  C (T x) { super(x); this.f=x; }
Methods        M ::=  <X◀N> T m (T x) { return t; }
Terms          t ::=    x
                      | t.f
                      | t.m<T>(t)
                      | new C(t)
                      | (C) t

Types          T ::=  X  |  N
Bounds         N ::=  C<T>          (= not variable)
```

FGJ Program = (CT, t)

CT: class table
(e.g., CT(Pair)=class Pair<X◀Obj..)

t: term to be evaluated

FJ

Judgement forms:

C <: D — subtyping (=subclassing!)
$\Gamma \vdash$ t : C — term typing

m ok in C — well-formed method
C ok — well-formed class

fields(C) = $\underline{\text{C f}}$ — field lookup
mtype(m,C) = $\underline{\text{C}}$ → C — method type lookup

FGJ

Judgement forms:

$C \leq D$	subclassing
$\Delta \vdash S <: T$	subtyping
$\Delta;\Gamma \vdash t : T$	term typing
$\Delta \vdash T$ ok	type well-formedness
m ok in $C<\underline{X} \blacktriangleleft \underline{N}>$	method typing
C ok	class typing
$fields(C<\underline{T}>) = \underline{C\ f}$	field lookup
$mtype(m,C<\underline{T}>) = <\underline{X} \blacktriangleleft \underline{N}>\underline{C} \rightarrow C$	method type lookup

3. Static Semantics of FGJ

Subclassing

Subclass relation ≤ determined by CT only!

$$\frac{CT(C) = \texttt{class } C\langle \underline{X} \blacktriangleleft \underline{N}\rangle \blacktriangleleft D\langle \underline{T}\rangle \ \{\ \ldots\ \}}{C \leq D}$$

reflexive $\quad C \leq C$

transitive

$$\frac{C \leq D \qquad D \leq E}{C \leq E}$$

3. Static Semantics of FGJ

Environment Γ is mapping from variables to types, written $\underline{x}:\underline{T}$

Type Environment Δ is mapping from type variables to nonvariable types (their bounds) written $\underline{X}<:\underline{N}$

$$\Gamma;\Delta \vdash x:\Gamma(x)$$

$$\Delta \vdash X<:\Delta(X)$$

→ Variables must be declared

3. Static Semantics of FGJ

Subtyping (=subclassing wrt type environment)

$$\frac{CT(C) = \texttt{class } C\langle \underline{X} \triangleleft \underline{N} \rangle \triangleleft N \{ \ \ldots \ \}}{\Delta \vdash C\langle \underline{T} \rangle <: [\underline{T}/\underline{X}]N}$$

reflexive $\Delta \vdash C <: C$ $\Delta \vdash X <: \Delta(X)$

transitive
$$\frac{\Delta \vdash C <: D \qquad \Delta \vdash D <: E}{\Delta \vdash C <: E}$$

Static Semantics (FJ)

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

3. Static Semantics of FGJ

Field selection:

$$\frac{\Gamma;\Delta \vdash t_0 : T_0 \qquad fields(\Delta(T_0)) = \underline{T}\ \underline{f}}{\Gamma;\Delta \vdash t_0.f_i : T_i}$$

→ field f_i must be present in $\Delta(T_0)$

→ its type is specified in $\Delta(T_0)$

Static Semantics (FJ)

Method invocation (message send):

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

→ method must be present

→ argument types must be subtypes of parameters

3. Static Semantics of FGJ

Method invocation (message send):

$$\frac{\Gamma;\Delta \vdash t_0 : C_0 \qquad \texttt{mtype}(m, \Delta(T_0)) = \langle \underline{X} \triangleleft \underline{N} \rangle \underline{U} \rightarrow U \qquad \Gamma \vdash \underline{t} : \underline{S} \qquad \Delta \vdash \underline{T} <: [\underline{T}/\underline{X}]\underline{N} \qquad \Delta \vdash \underline{S} <: [\underline{T}/\underline{X}]\underline{U}}{\Gamma;\Delta \vdash t_0.m\langle T \rangle(\underline{t}) : [T/X]U}$$

→ method must be present

→ argument parameters must respect bounds

→ argument types must be subtypes of [T/X]parameters

Static Semantics (FJ)

Instantiation (object creation):

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

→ class name must exists

→ initializers must be of subtypes of fields

3. Static Semantics of FGJ

Instantiation (object creation):

$$\frac{\Gamma;\Delta \vdash \underline{t}:\underline{S} \qquad \Delta \vdash \underline{S}<:\underline{T} \qquad \mathtt{fields}(N) = \underline{T\ f}}{\Gamma;\Delta \vdash \mathtt{new}\ N(\underline{t}) : N}$$

→ class name must exists

→ initializers must be of subtypes of fields

Static Semantics (FJ)

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:

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

3. Static Semantics of FGJ

Casting:

up

$$\frac{\Gamma;\Delta \vdash t_0 : T_0 \qquad \Delta \vdash \Delta(T_0) <: N}{\Gamma;\Delta \vdash (N)t_0 : N}$$

$$\frac{\Gamma;\Delta \vdash t_0 : T_0 \qquad \Delta(T_0) = D\langle \underline{U} \rangle \qquad \text{not}(C \leq D \text{ or } D \leq C) \qquad \text{warning!}}{\Gamma;\Delta \vdash (C\langle \underline{T} \rangle)t_0 : C\langle \underline{T} \rangle}$$

3. Static Semantics of FGJ

Down Cast:

$$\frac{\Gamma;\Delta \vdash t_0 : T_0 \qquad \Delta \vdash \mathtt{C}\langle \mathtt{T}\rangle <: \Delta(T_0) = \mathtt{D}\langle \underline{\mathtt{U}}\rangle \qquad \mathtt{dcast(C,D)}}{\Gamma;\Delta \vdash (\mathtt{C}\langle \underline{\mathtt{T}}\rangle)t_0 : \mathtt{C}\langle \underline{\mathtt{T}}\rangle}$$

dcast(C,D) : climb up class hierachy, if

$\mathtt{class\ C}\langle \underline{\mathtt{X}} \blacktriangleleft \underline{\mathtt{B}}\rangle \blacktriangleleft \mathtt{C'}\langle \underline{\mathtt{T}}\rangle$ { … }

appears, then $\underline{\mathtt{X}}$ must equal
the set of type variables in T!

Static Semantics (FJ) → exactly same in FGJ

Well-Formed Classes

$$\frac{\texttt{K = C(}\underline{\texttt{D}}\ \underline{\texttt{g}}\texttt{, }\underline{\texttt{C}}\ \underline{\texttt{f}}\texttt{) \{ super(}\underline{\texttt{g}}\texttt{); this.}\underline{\texttt{f}}\texttt{ = }\underline{\texttt{f}}\texttt{; \}} \qquad \texttt{fields(D) = }\underline{\texttt{D}}\ \underline{\texttt{g}} \qquad \underline{\texttt{M}}\ \texttt{ok in C}}{\texttt{Class C extends D \{ }\underline{\texttt{C}}\ \underline{\texttt{f}}\texttt{; K }\underline{\texttt{M}}\texttt{ \} ok}}$$

→ 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

Static Semantics (FJ)

Well-Formed Methods

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

→ must return a subtype of the result type

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

3. Static Semantics of FGJ

Well-Formed Methods

$$\frac{\begin{array}{ll} \Delta = \langle\underline{X} \blacktriangleleft \underline{N}\rangle,\ \langle\underline{Y} \blacktriangleleft \underline{P}\rangle & \\ CT(C) = \texttt{class}\ C\langle\underline{X} \blacktriangleleft \underline{N}\rangle N\ \{\ \ldots\ \} & \\ \texttt{override}(m,\ N,\ \langle\underline{Y} \blacktriangleleft \underline{P}\rangle \underline{T} \rightarrow T) & \\ \Delta,\ \underline{x}:\underline{T},\ \texttt{this}:C\langle\underline{X}\rangle \vdash t_0 : S & \qquad \Delta \vdash S <: T \end{array}}{\langle\underline{Y} \blacktriangleleft \underline{P}\rangle T_0\ m\ (\underline{T}\ \underline{x})\ \{\ \texttt{return}\ t_0;\ \}\quad \textbf{ok in}\ C\langle\underline{X} \blacktriangleleft \underline{N}\rangle}$$

→ must return a subtype of the result type

Static Semantics (FJ)

Method Type Lookup

$$\frac{\begin{array}{c} CT(C) = \texttt{class } C \texttt{ extends } D \ \{ \ \underline{C} \ \underline{f}; \ K \ \underline{M} \ \} \\ B \ m \ (\underline{B} \ \underline{x}) \ \{ \ \texttt{return } t; \ \} \in \underline{M} \end{array}}{mtype(m,C) = \underline{B} \rightarrow B}$$

$$\frac{\begin{array}{c} CT(C) = \texttt{class } C \texttt{ extends } D \ \{ \ \underline{C} \ \underline{f}; \ K \ \underline{M} \ \} \\ m \text{ not defined in } \underline{M} \end{array}}{mtype(m,C) = mtype(m,D)}$$

Method Body Lookup works exactly the same.

→ returns $(\underline{x},t)$

3. Static Semantics of FGJ

Method Type Lookup

$$\frac{\begin{array}{c}\mathtt{CT(C) = class\ C{<}\underline{X}\blacktriangleleft\underline{N}{>}\blacktriangleleft N\ \{\ \underline{S\ f};\ K\ \underline{M}\ \}} \\ \mathtt{{<}\underline{Y}\blacktriangleleft\underline{P}{>}\ U\ m\ (\underline{U}\ \underline{x})\ \{\ return\ t;\ \}} \in \underline{\mathtt{M}}\end{array}}{\mathtt{mtype(m,C{<}\underline{T}{>}) = [\underline{T}/\underline{X}]({<}\underline{Y}\blacktriangleleft\underline{P}{>}\underline{U}\rightarrow U)}}$$

$$\frac{\begin{array}{c}\mathtt{CT(C) = class\ C{<}\underline{X}\blacktriangleleft\underline{N}{>}\blacktriangleleft N\ \{\ \underline{S\ f};\ K\ \underline{M}\ \}} \\ \mathtt{m\ not\ defined\ in\ \underline{M}}\end{array}}{\mathtt{mtype(m,C{<}\underline{T}{>}) = mtype(m,[\underline{T}/\underline{X}]N)}}$$

Method Body Lookup works exactly the same.
→ returns ($\underline{x}$,t)

Static Semantics (FJ)

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. Static Semantics of FGJ

Field Lookup

$$fields(Object) = [\ ]$$

$$\frac{CT(C) = \texttt{class}\ C\langle\underline{X}\triangleleft\underline{N}\rangle\triangleleft N\ \{\ \underline{S\ f};\ K\ \underline{M}\ \} \qquad fields([T/X]N) = \underline{U\ g}}{fields(m, C\langle T\rangle) = \underline{U\ g},\ [T/X]\underline{S\ f}}$$

→ Concatenation of super-class fields, plus new ones

Dynamic Semantics (FJ)

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.

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

field selection

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

method invocation

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

casting

4. Dynamic Semantics FGJ

Object values have the form new $C\langle\underline{T}\rangle(\underline{s},\underline{t})$

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

$$\frac{\text{fields}(N) = \underline{T}\ \underline{f}}{(\text{new } N(\underline{t})).f_i \rightarrow t_i}$$

field selection

$$\frac{\text{mbody}(m\langle\underline{V}\rangle, N) = (\underline{x}, t_0)}{(\text{new } N(\underline{t})).m\langle\underline{V}\rangle(\underline{d}) \rightarrow [\underline{d}/\underline{x},\ \text{new } N(\underline{t})/\text{this}]\ t_0}$$

method invocation

$$\frac{\varnothing \vdash N <: P}{(P)(\text{new } N(\underline{t})) \rightarrow \text{new } N(\underline{t})}$$

casting

Example of a type derivation in FGJ

$$\dfrac{}{\emptyset;\emptyset \vdash \texttt{(new Pair<Pair<A,B>,A>(new Pair<A,B>(new A(), new B()), new A()).fst).snd: Obj}}$$

5. Type Safety

Theorem (Preservation)

If $\Gamma;\Delta \vdash$ `t:T` and `t`$\rightarrow$`t'` then

$\Gamma;\Delta \vdash$ `t':T'` for some $\Delta \vdash$ `T'<:C`.

→ Proof by induction on the length of evaluations.

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

5. Type Safety

Theorem (Progress)

Let CT be a well-formed class table.
If $\varnothing; \varnothing \vdash$ `t:T` then either

1. `t` is a value, or

2. `t = (D) new C(`v_0`)` and not(`C <: D`), 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.

6. Erasure Semantics

→ Current GJ/Java compiler translates into standard JVM (maintains NO runtime info on type param's)

→ Same is possible for FGJ/FJ:

FGJ program ——erasure——→ FJ program

```
class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  <Z extends Object> Pair<Z,Y> setfst(Z newfst) {
    return new Pair<Z,Y>(newfst, this.snd);}}
```

erases to

```
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super();
    this.fst = fst;
    this.snd = snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);}}
```

```
New Pair<A,B>(new A(), new B()).snd
```

erases to

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

Erasure semantics:

→ Types are erased to (the erasure of) their bounds.

→ Field/Method lookup:

A subclass may extend an instantiated superclass!

```
class Pair<X extends Object, Y extends Object>
      extends Object {
  X fst;
  Y snd;
  Pair(X fst, Y snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  Pair<X,Y> setfst(X newfst) {
    return new Pair<X,Y>(newfst, this.snd);
}}
```

→ Has the SAME ERASURE as the Pair class of before!!

```
class PairOfA extends Pair<A,A> {
  PairOfA(A fst, A snd) { super(fst, snd); }
  PairOfA setfst(A newfst) {
    return new PairOfA(newfst, this.snd);
  }
}
```

Covariant subtype of setfst in Pair<A,A>

```
class PairOfA extends Pair<A,A> {
  PairOfA(A fst, A snd) { super(fst, snd); }
  PairOfA setfst(A newfst) {
    return new PairOfA(newfst, this.snd);
  }
}
```

Is erased to

```
class PairOfA extends Pair {
  PairOfA(Object fst, Object snd) { super(fst, snd); }
  PairOfA setfst(Object newfst) {
    return new PairOfA((A) newfst, (A) this.snd);
  }
}
```

All chosen to correspond to types in Pair,
The highest superclass in which the fields/methods
Are defined!!

In GJ/Java, erasure introduces bridge methods:

Erasure of PairOfA would be

```
class PairOfA extends Pair {
  PairOfA(Object fst, Object snd) {
    super(fst, snd);
  }

  PairOfA setfst(A newfst) {
    return new PairOfA(newfst, (A) this.snd);
  }

  PairOfA setfst(Object newfst) {
    return this.setfst((A) newfst);
  }
}
```

Bridge method which overrides setfst in Pair