



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.			
<u>Total:</u>			<u>364 (+112)</u>

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.

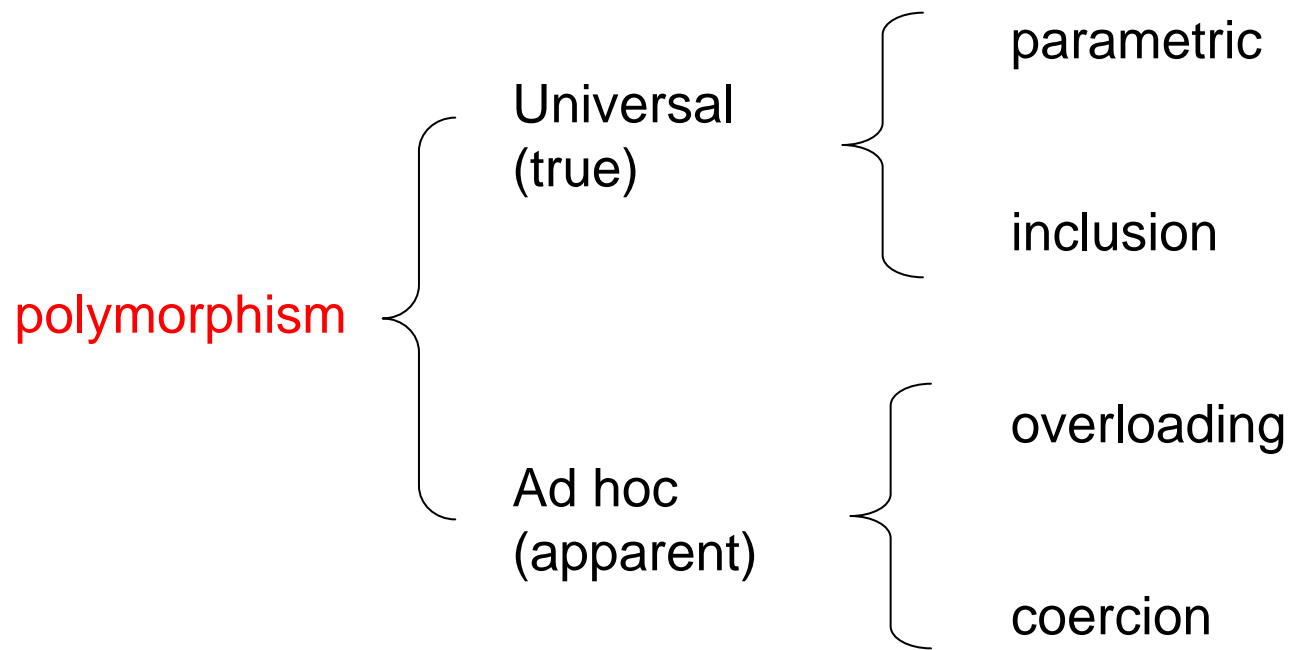
→ Polymorphism

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)



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**

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

$Y \rightarrow Y$	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**

$\lambda X <: \{a: \text{Nat}\}. \lambda x: X. \{\text{orig}=x, \text{asucc}=\text{succ}(x.a)\};$

→ 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);}}}
```

→ **C**lasses AND **m**ethods may have **g**eneric type param's:

X, Y: type parameters of **c**lass **P**air
Z: type parameter of **m**ethod **s**etfst

→ Each type parameter has a *bound*.


here: **X, Y, Z** all have bound **O**bject

```
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(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 `` 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	$C ::=$	<code>class C <D { <u>C</u> f; K <u>M</u> }</code>
Constructors	$K ::=$	<code>C (<u>C</u> <u>x</u>) { super(<u>x</u>); <u>this.f=x</u>; }</code>
Methods	$M ::=$	<code>C m (<u>C</u> <u>x</u>) { return t; }</code>
Terms	$t ::=$	<code>x</code> <code> t.f</code> <code> t.m(<u>t</u>)</code> <code> new C(<u>t</u>)</code> <code> (C) t</code>


2. Syntax of FGJ

List of type parameters w bounds



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

Types	$T ::=$	<code>X N</code>	
Bounds	$N ::=$	<code>C<<u>T</u>></code>	(= 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

$m \text{ ok in } C$ well-formed method

$C \text{ ok}$ well-formed class

$\text{fields}(C) = \underline{c_f}$ field lookup

$\text{mtype}(m, C) = \underline{c} \rightarrow C$ method type lookup

FGJ

Judgement forms:

$C \leq D$

subclassing

$\Delta \vdash S <: T$

subtyping

$\Delta; \Gamma \vdash t : T$

term typing

$\Delta \vdash T \text{ ok}$

type well-formedness

$m \text{ ok in } C \langle \underline{X} \langle \underline{N} \rangle \rangle$

method typing

$C \text{ ok}$

class typing

$\text{fields}(C \langle \underline{T} \rangle) = \underline{c_f}$

field lookup

$\text{mtype}(m, C \langle \underline{T} \rangle) = \langle \underline{X} \langle \underline{N} \rangle \rangle \underline{c} \rightarrow C$

method type lookup

3. Static Semantics of FGJ

Subclassing

subclass relation \leq determined by CT only!

$$\frac{\text{CT}(C) = \text{class } C \langle \underline{X} \langle \underline{N} \rangle \langle \underline{D} \langle \underline{T} \rangle \rangle \{ \dots \}}{C \leq D}$$

reflexive $C \leq C$

transitive $\frac{C \leq D \quad D \leq E}{C \leq E}$

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)

$$\text{CT}(C) = \text{class } C \langle \underline{X} \langle \underline{N} \rangle \langle \underline{N} \rangle \{ \dots \}$$

$$\Delta \vdash C \langle \underline{T} \rangle \langle :: \rangle [\underline{T}/\underline{X}] \underline{N}$$

reflexive $\Delta \vdash C \langle :: \rangle C$ $\Delta \vdash X \langle :: \rangle \Delta(X)$

transitive $\Delta \vdash C \langle :: \rangle D$ $\Delta \vdash D \langle :: \rangle E$

$$\Delta \vdash C \langle :: \rangle E$$

Static Semantics (FJ)

Field selection:

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

- field f_i must be present in C_0
- its type is specified in C_0

3. Static Semantics of FGJ

Field selection:

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

- 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):

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

→ method must be present

→ argument types must be subtypes of parameters

3. Static Semantics of FGJ

Method invocation (message send):

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash t_0 : C_0 \quad \text{mtype}(m, \Delta(T_0)) = \langle \underline{X} \langle \underline{N} \rangle \underline{U} \rangle \rightarrow U \\ \Gamma \vdash \underline{t} : \underline{S} \quad \Delta \vdash \underline{I} <: [\underline{I}/\underline{X}] \underline{N} \quad \Delta \vdash \underline{S} <: [\underline{T}/\underline{X}] \underline{U} \end{array}}{\Gamma; \Delta \vdash t_0.m \langle \underline{T} \rangle (\underline{t}) : [\underline{T}/\underline{X}] \underline{U}}$$

- method must be present
- argument parameters must respect bounds
- argument types must be subtypes of $[\underline{T}/\underline{X}]$ parameters

Static Semantics (FJ)

Instantiation (object creation):

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

- class name must exist
- initializers must be of subtypes of fields

3. Static Semantics of FGJ

Instantiation (object creation):

$$\frac{\Gamma; \Delta \vdash \underline{t} : \underline{S} \quad \Delta \vdash \underline{S} <: \underline{I} \quad \text{fields}(N) = \underline{I} \ \underline{f}}{\Gamma; \Delta \vdash \text{new } N(\underline{t}) : N}$$

- class name must exist
- initializers must be of subtypes of fields

Static Semantics (FJ)

Casting: (up or down)

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

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

→ stupid (side) casts can be detected:

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

3. Static Semantics of FGJ

Casting:

up

$$\frac{\Gamma; \Delta \vdash t_0 : T_0 \quad \Delta \vdash \Delta(T_0) \leq : N}{\Gamma; \Delta \vdash (N)t_0 : N}$$

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

3. Static Semantics of FGJ

Down Cast:

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

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

```
class C<X<B><C'<T> { ... }
```

appears, then X must equal
the set of type variables in T!

Static Semantics (FJ) → exactly same in FGJ

well-Formed Classes

$$K = C(\underline{D} \underline{g}, \underline{C} \underline{f}) \{ \text{super}(\underline{g}); \text{this}.\underline{f} = \underline{f}; \}$$
$$\text{fields}(D) = \underline{D} \underline{g} \quad \underline{M} \text{ ok in } C$$

Class C extends D { $\underline{C} \underline{f}; K \underline{M}$ } 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**

Static Semantics (FJ)

Well-Formed Methods

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

$$C_0 \text{ m } (\underline{C} \ \underline{x}) \{ \text{return } t_0; \} \text{ ok in } C$$

→ must return a subtype of the result type

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

3. Static Semantics of FGJ

Well-Formed Methods

$$\begin{array}{l} \Delta = \langle \underline{X} \langle \underline{N} \rangle, \langle \underline{Y} \langle \underline{P} \rangle \rangle \\ \text{CT}(C) = \text{class } C \langle \underline{X} \langle \underline{N} \rangle \rangle N \{ \dots \} \\ \text{override}(m, N, \langle \underline{Y} \langle \underline{P} \rangle \rangle \underline{I} \rightarrow \underline{T}) \\ \Delta, \underline{x} : \underline{I}, \text{this} : C \langle \underline{X} \rangle \vdash t_0 : S \qquad \Delta \vdash S <: T \\ \hline \langle \underline{Y} \langle \underline{P} \rangle \rangle T_0 \text{ m } (\underline{I} \ \underline{x}) \{ \text{return } t_0; \} \text{ ok in } C \langle \underline{X} \langle \underline{N} \rangle \rangle \end{array}$$

→ must return a subtype of the result type

Static Semantics (FJ)

Method Type Lookup

$$\text{CT}(C) = \text{class } C \text{ extends } D \{ \underline{C} \underline{f}; K \underline{M} \}$$
$$B \ m \ (\underline{B} \ \underline{x}) \{ \text{return } t; \} \in \underline{M}$$

$$\text{mtype}(m, C) = \underline{B} \rightarrow B$$
$$\text{CT}(C) = \text{class } C \text{ extends } D \{ \underline{C} \underline{f}; K \underline{M} \}$$
$$m \text{ not defined in } \underline{M}$$

$$\text{mtype}(m, C) = \text{mtype}(m, D)$$

Method Body Lookup works exactly the same.

→ returns (\underline{x}, t)

3. Static Semantics of FGJ

Method Type Lookup

$$\text{CT}(C) = \text{class } C \langle \underline{X} \langle \underline{N} \rangle \langle \underline{N} \rangle \{ \underline{s} \ \underline{f}; \ \underline{K} \ \underline{M} \} \\ \langle \underline{Y} \langle \underline{P} \rangle \rangle \ \underline{U} \ \underline{m} \ (\underline{U} \ \underline{x}) \ \{ \text{return } \underline{t}; \} \in \underline{M}$$

$$\text{mtype}(m, C \langle \underline{T} \rangle) = [\underline{T}/\underline{X}] (\langle \underline{Y} \langle \underline{P} \rangle \rangle \underline{U} \rightarrow \underline{U})$$
$$\text{CT}(C) = \text{class } C \langle \underline{X} \langle \underline{N} \rangle \langle \underline{N} \rangle \{ \underline{s} \ \underline{f}; \ \underline{K} \ \underline{M} \} \\ \underline{m} \ \text{not defined in } \underline{M}$$

$$\text{mtype}(m, C \langle \underline{T} \rangle) = \text{mtype}(m, [\underline{T}/\underline{X}] \underline{N})$$

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

Static Semantics (FJ)

Field Lookup

`fields(Object) = []`

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

→ Concatenation of super-class fields, plus new ones

3. Static Semantics of FGJ

Field Lookup

`fields(Object) = []`

$$\text{CT}(C) = \text{class } C \langle X \langle N \rangle \langle N \rangle \{ \underline{s_f}; K \underline{M} \}$$
$$\text{fields}([T/X]N) = \underline{u_g}$$

$$\text{fields}(m, C \langle T \rangle) = \underline{u_g}, [T/X]\underline{s_f}$$

→ Concatenation of super-class fields, plus new ones

Dynamic Semantics (FJ)

object values have the form $\text{new } C(\underline{s}, \underline{t})$

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

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

field
selection

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

method
invocation

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

casting

4. Dynamic Semantics FGJ

object values have the form $\text{new } C\langle I \rangle(\underline{s}, \underline{t})$

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

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

field
selection

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

method
invocation

$$\frac{\emptyset \vdash N\langle P \rangle}{\emptyset \vdash N\langle P \rangle}$$
$$\frac{\emptyset \vdash N\langle P \rangle}{(P)(\text{new } N(\underline{t})) \rightarrow \text{new } N(\underline{t})}$$

casting



Example of a type derivation in FGJ

$\emptyset; \emptyset \vdash (\text{new Pair}\langle \text{Pair}\langle A, B \rangle, A \rangle (\text{new Pair}\langle A, B \rangle (\text{new A}(), \text{new B}()), \text{new A}()).\text{fst}).\text{snd}: \text{Obj}$

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 $\emptyset; \emptyset \vdash t:T$ then either

1. t is a value, or
2. $t = (D) \text{ new } C(v_0)$ and $\text{not}(C <: D)$, or
3. there exists t' such that $t \rightarrow t'$.

→ 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 \longrightarrow FJ program
erasure

```

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