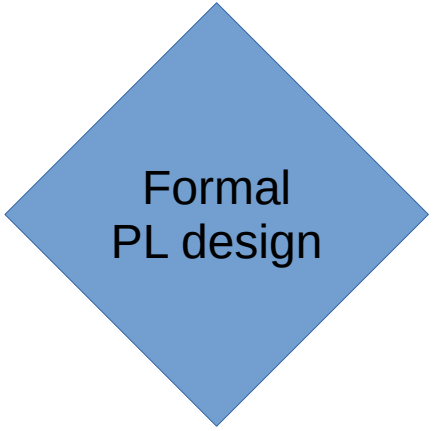
SWEN423 - Featherweight Java



Lect4: Type system

Meaning of the reduction arrow

- If an element of the form

```
e --> e'
```

  is contained in the set of defined reduction arrows, then e reduces in a single step in e'. A complete computation is just the concatenation of all the reduction steps.

Transitive and reflexive closure

$$\frac{e_1 \longrightarrow e_2}{e_1 \longrightarrow^* e_2}\ (\text{base})$$

$$\frac{}{e_1 \longrightarrow^* e_1}\ (\text{refl})$$

$$\frac{e_1 \longrightarrow^* e_2 \qquad e_2 \longrightarrow^* e_3}{e_1 \longrightarrow^* e_3}\ (\text{trans})$$

Every arrow implicitly support the transitive closure, modeling the full execution instead of just a single reduction step

Values, Redexes and Stuck

- Given a reduction arrow, a **normal form** is a term that can not be reduced any further.
- A **value** is a kind of normal form identified as an expected final result.
- Any normal form that is not a value is called **Stuck**
- For example, calling a method that does not exists. The goal of **type systems** is to rule-out terms whose execution may go stuck.

Notations and conventions

- Some use 's' to denote repetition, as in

  ' Fs ' is implicitly defined as ' Fs ::= $F_1..F_n$ '

- Some (more common) use the overbar, as in

  ' $\overline{F}$ ' is implicitly defined as ' $\overline{F}$ ::= $F_1..F_n$ '

- However, there are other interpretations for the overbar:

  ' $\overline{F}$ ' is implicitly replaced in place with ' $F_1..F_n$ '

  In this interpretation the index 'n' can be used in the context of the overbar, and multiple overbars must have the same length. To avoid those ambiguities, we are avoiding the overbar.

Notations and conventions

- The form e[x=e'] is the most common and modern notation for variable substitution, but there is not a real agreement. In various works I have seen many of the following variants to express the same concept:
  e[x=e'] e{x=e'} [x=e']e {x=e'}e
  e[x\e'] e[x/e'] e[x<-e'] e[x->e']
  e[e'\x] e[e'/x] $e[^{e'}/_{x}]$ $e[^{x}/_{e'}]$

Notations and conventions

- Different researchers / research groups use the metarule language in sightly different ways:
- Some insist of writing the side conditions “at the side”, some mix them together with the preconditions, some just avoid the line and just write

'consequence'
where: 'sideconditions', 'premised'

FULL FJ reduction, fully compact

$$\frac{e_1 \longrightarrow e_2}{\mathcal{E}v[e_1] \longrightarrow \mathcal{E}v[e_2]}\ (\text{ctx})$$

$$\frac{\text{class C \_\{ } C_1\ x_1; .. C_n\ x_n;\ \text{K Ms } \}\ \text{in cds}}{\text{new C}(v_1..v_n).x_i \longrightarrow v_i}\ (\text{f-access})$$

$$\frac{\text{class C \_\{ \_ } C_0\ \text{m}(C_1\ x_1..C_n\ x_n)\{\text{return e;}\}\ \text{\_ } \}\ \text{in cds}}{\text{new C(vs).m}(v_1..v_n) \longrightarrow e[\text{this=new C(vs)}, x_1=v_1..x_n=v_n]}\ (\text{m-call})$$

```
e  ::= x | e.m(es) | new C(es) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(C1 x1 .. Cn xn)

v  ::= new C(vs)

Ev ::= □ | Ev.m(es) | v.m(vs,Ev,es)| new C(vs,Ev,es) | Ev.f
```

```
#Define  e0[x=e1] = e2
  x[x=e] = e
  x[x'=e] = x where: x != x'
  e0.m(e1..en)[x=e] = e0[x=e].m(e1[x=e]..en[x=e])
  new C(e1..en)[x=e] = new C(e1[x=e]..en[x=e])
  e0.f[x=e] = e0[x=e].f
```

Terminology: “Variable binding”

```
class User{
  Num foo(Point p){return p.getX();}
  }
```

We say that in the expression 'p.getX()'
'p' is a free variable.
In the case of method 'foo', 'p' is bonded to the parameter 'Point p'

In a FJ main expression, we should have no free variables.
They only make sense inside of method bodies,
so that they can refer to the formal parameters.

FJ type system: Gamma "Γ"

Γ ::= $x_1 : C_1 .. x_n : C_n$ 

Keeps track of variables types.
For example, the following code:

```
class User{
  Num foo(Point p){return p.getX();}
  }
```

'p.getX()' will be typed with Γ= this:User, p:Point

We can write "Γ(x)" to extract the C associated to x.

FJ type judgements are of the form: //just another form of syntax

Γ |- e : C

That reads as: **in the environment Gamma, the expression e has type C.**

```
e  ::= x | e.m(es) | new C(es) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(T1 x1 .. Tn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(T1 x1 .. Tn xn)

v  ::= new C(vs)

Ev ::= □ | Ev.m(es) | v.m(vs,Ev,es)| new C(vs,Ev,es) | Ev.f

Γ  ::= x1 : C1 .. xn : Cn
```

```
_______
#Define  e0[x=e1] = e2
  x[x=e] = e
  x[x'=e] = x where: x != x'
  e0.m(e1..en)[x=e] = e0[x=e].m(e1[x=e]..en[x=e])
  new C(e1..en)[x=e] = new C(e1[x=e]..en[x=e])
  e0.f[x=e] = e0[x=e].f
```

FJ expression type system:

$$\text{(x-t)}\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\text{(m-t)}\ \frac{\Gamma \vdash e_0 : C_0 \ldots \Gamma \vdash e_n : C_n \quad \_\ C_0\ \_\ \{\_\ C\ m(C_1\ x_1 .. C_n\ x_n)\_\ \_\}\ \text{in cds}}{\Gamma \vdash e_0.m(e_1 .. e_n) : C}$$

$$\text{(n-t)}\ \frac{\Gamma \vdash e_1 : C_1 \ldots \Gamma \vdash e_n : C_n \quad \text{class C \_ \{ } C_1\ x_1; .. C_n\ x_n; \text{ K Ms \} in cds}}{\Gamma \vdash \text{new C}(e_1 .. e_n) : C}$$

$$\text{(f-t)}\ \frac{\Gamma \vdash e : C \quad \text{class C \_ \{ } C_1\ x_1; .. C_n\ x_n; \text{ K Ms \} in cds}}{\Gamma \vdash e.x_i : C_i}$$

```
e  ::= x | e.m(es) | new C(es) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(T1 x1 .. Tn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(T1 x1 .. Tn xn)

v  ::= new C(vs)

Ev ::= □ | Ev.m(es) | v.m(vs,Ev,es)| new C(vs,Ev,es) | Ev.f

Γ  ::= x1 : C1 .. xn : Cn
```

$$\text{(sub-trans)}\ \frac{C_1 \le C_2 \quad C_2 \le C_3}{C_1 \le C_3}$$

$$\text{(sub-direct)}\ \frac{i \text{ in } 0..n \quad \_\ C_0\ \_\ C_1..C_n\ \{\_\}\ \text{in cds}}{C_0 \le C_i}$$

$$\text{(sub)}\ \frac{C \le C' \quad \Gamma \vdash e : C}{\Gamma \vdash e : C'}$$

Meaning of the Type Judgment?

- A well typed expression can be reduced without going stuck. The metarules are defining a set of type judgments, in the same way the reduction rules are defining the set of valid reduction arrows.
- If an element of the form

  ```
  Γ |- e : C
  ```

  is contained in the set of the defined typed judgments, then e is well typed in such Gamma, and we can prove that is not going stuck.

FJ class type system:

$$\frac{C;Cs \vdash M_1: ok \;..\; C;Cs \vdash M_n: ok \qquad dom(C') \subseteq dom(C)\ \text{forall } C' \text{ in } Cs}{\text{(cd)}\quad \text{class C implements Cs\{ Fs K } M_1..M_n\} : ok}$$

$$\frac{override Ok(C',MH_1)\ \text{forall } C' \text{ in } Cs \;..\; override Ok(C',MH_n)\ \text{forall } C' \text{ in } Cs}{\text{(id)}\quad \text{interface C extends Cs\{}MH_1;..MH_n;\} : ok}$$

$$\frac{override Ok(C',C_0\ m(C_1\ x_1..C_n\ x_n))\ \text{forall } C' \text{ in } Cs \qquad this:C, x_1:C_1..x_n:C_n \vdash e : C_0}{\text{(meth)}\quad C;Cs \vdash C_0\ m(C_1\ x_1..C_n\ x_n)\{\text{return e;}\} : ok}$$

.

```
e  ::= x | e.m(es) | new C(es) | e.f

cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }

K  ::= C(T1 x1 .. Tn xn){this.x1=x1;..this.xn=xn;}

F  ::= C f;        M  ::= MH{return e;}

MH ::= C m(T1 x1 .. Tn xn)

v  ::= new C(vs)

Ev ::= □ | Ev.m(es) | v.m(vs,Ev,es)| new C(vs,Ev,es) | Ev.f

Γ  ::= x1 : C1 .. xn : Cn
```

```
_______
#Define overrideOk(C,MH)
  overrideOk(C,C0 m(C1 x1..Cn xn))
    interface C _{MHs} in cds
    C0 m(C1 x1..Cn xn) in MHs
    or m not in dom(C)

_______
#Define dom(C)
  dom(C) = m1..mn
   class C _{Fs K M1..Mn} in cds
    Mi= _ mi(_){_}
  dom(C) = m1..mn
    interface C _{MH1..MHn} in cds
    MHi= _ mi(_)
```

Something missing?

- Well Formedness: a filter over the grammar definition, removing all the “obvious mistakes” early:
  - All classes and interfaces are uniquely named.
  - All methods in a given class are uniquely named.
  - All fields in a given class are uniquely named.
  - All parameters in a given method are uniquely named and not called this.
  - Classes can only implement interfaces.
  - Interfaces can only extend other interfaces.

$$\frac{e_1 \longrightarrow e_2}{\mathcal{E}v[e_1] \longrightarrow \mathcal{E}v[e_2]}\ (ctx) \qquad \frac{\texttt{class C \_\{ } C_1\ x_1;..C_n\ x_n\texttt{; K Ms \} in cds}}{\texttt{new C(}v_1..v_n\texttt{).}x_i \longrightarrow v_i}\ (f\text{-}access)$$

$$\frac{\texttt{class C \_\{ \_ } C_0\ \texttt{m(}C_1\ x_1..C_n\ x_n\texttt{)\{return e;\} \_ \} in cds}}{\texttt{new C(vs).m(}v_1..v_n\texttt{)} \longrightarrow e[\texttt{this=new C(vs)},x_1=v_1..x_n=v_n]}\ (m\text{-}call)$$

```
e  ::= x | e.m(es) | new C(es) | e.f
cd ::= class C implements Cs{ Fs K Ms }
     | interface C extends Cs{ MH1; .. MHk; }
K  ::= C(C1 x1 .. Cn xn){this.x1=x1;..this.xn=xn;}
F  ::= C f;        M  ::= MH{return e;}
MH ::= C m(C1 x1 .. Cn xn)
v  ::= new C(vs)
Ev ::= □ | Ev.m(es) | v.m(vs,Ev,es)| new C(vs,Ev,es) | Ev.f
Γ  ::= x1 : C1 .. xn : Cn
```

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (x\text{-}t) \qquad \frac{\Gamma \vdash e_0 : C_0\ ..\ \Gamma \vdash e_n : C_n \quad \_\ C_0\ \_\ \{\_\ C\ \texttt{m(}C_1\ x_1..C_n\ x_n\texttt{)\_ \_\}\ in\ cds}}{\Gamma \vdash e_0.\texttt{m(}e_1\ ..\ e_n\texttt{)} : C}\ (m\text{-}t)$$

$$\frac{\Gamma \vdash e_1 : C_1\ ..\ \Gamma \vdash e_n : C_n \quad \texttt{class C \_ \{}C_1\ x_1\texttt{;..}C_n\ x_n\texttt{; K Ms\} in cds}}{\Gamma \vdash \texttt{new C(}e_1\ ..\ e_n\texttt{)} : C}\ (n\text{-}t) \qquad \frac{\Gamma \vdash e : C \quad \texttt{class C \_ \{}C_1\ x_1\texttt{;..}C_n\ x_n\texttt{; K Ms\} in cds}}{\Gamma \vdash e.x_i : C_i}\ (f\text{-}t)$$

$$\overline{\texttt{\#Define}\ \ e_0[x=e_1] = e_2}$$

```
x[x=e] = e
x[x'=e] = x where: x != x'
e0.m(e1..en)[x=e] = e0[x=e].m(e1[x=e]..en[x=e])
new C(e1..en)[x=e] = new C(e1[x=e]..en[x=e])
e0.f[x=e] = e0[x=e].f
```

$$\overline{\texttt{\#Define overrideOk(C,MH)}}$$

```
overrideOk(C,C0 m(C1 x1..Cn xn))
  interface C _{MHs} in cds
  C0 m(C1 x1..Cn xn) in MHs
  or m not in dom(C)
```

$$\frac{\texttt{overrideOk(C',}C_0\ \texttt{m(}C_1\ x_1..C_n\ x_n\texttt{)) forall C' in Cs} \quad \texttt{this:C,}x_1\texttt{:}C_1..x_n\texttt{:}C_n \vdash e : C_0}{\texttt{C;Cs} \vdash C_0\ \texttt{m(}C_1\ x_1..C_n\ x_n\texttt{)\{return e;\} : ok}}\ (meth)$$

$$\frac{\texttt{C;Cs} \vdash M_1\texttt{: ok .. C;Cs} \vdash M_n\texttt{: ok} \quad \texttt{dom(C')} \subseteq \texttt{dom(C) forall C' in Cs}}{\texttt{class C implements Cs\{ Fs K }M_1..M_n\texttt{\}:ok}}\ (cd) \qquad \frac{\texttt{overrideOk(C',}MH_1\texttt{) forall C' in Cs} \quad .. \quad \texttt{overrideOk(C',}MH_n\texttt{) forall C' in Cs}}{\texttt{interface C extends Cs\{}MH_1\texttt{;..}MH_n\texttt{;\} : ok}}\ (id)$$

$$\overline{\texttt{\#Define dom(C)}}$$

```
dom(C) = m1..mn
  class C _{Fs K M1..Mn} in cds
  Mi= _ mi(_){_}
dom(C) = m1..mn
  interface C _{MH1..MHn} in cds
  MHi= _ mi(_)
```

$$\frac{C \le C' \quad \Gamma \vdash e : C}{\Gamma \vdash e : C'}\ (sub) \qquad \frac{i \text{ in } 0..n \quad \_\ C_0\ \_\ C_1..C_n\ \{\_\}\ in\ cds}{C_0 \le C_i}\ (sub\text{-}direct) \qquad \frac{C_1 \le C_2 \quad C_2 \le C_3}{C_1 \le C_3}\ (sub\text{-}trans)$$

Well Formedness:

-All classes and interfaces are uniquely named.
-All methods in a given class are uniquely named.
-All fields in a given class are uniquely named.
-All parameters in a given method are uniquely named and not called this.
-Classes can only implement interfaces.
-Interfaces can only extend other interfaces.

All FJ in a slide

- Compacting the formal definitions of a language means that we can easily “keep them around” when we do any work on the language.
- For example, in the language I'm developing, it is fundamental that I can keep the full language specification in front of me while I work on the implementation

Ass1 deadline tomorrow

- If you have not finished yet, consider hurrying up!
- The assignment works on the web assessment tool, so what should we submit in the assessment system?