cd ::= class C extends D { $\overline{\text{fd}}$; kd $\overline{\text{md}}$ } class
fd ::= q C f field
kd ::= q C (t C g, t C f) { super(g); this.f = f; } constructor
md ::= t C m (t C this, t C x) { $\overline{\text{t C y}}$ $\overline{\text{s}}$; return z } instance method
e ::= x | x.f expression
s ::= x = e | x.f = y | x = y.m(z) | super(g) | this(g) | x = new C() | s;s statement
t ::= k q qualifier type
k ::= initialized | underinitialized | unknowninitialized | fbcbottom
initializatioin qualifier

q ::= readonly | mutable | polymutable | substitutablepolymutable |
receiverdependantmutable | immutable | bottom
immutability qualifier

Qualifier Hierarchy

```
                         readonly
      /      /               |                 \                  \
mutable  polymutable  substitutablepolymutable  receiverdependantmutable immutable
  (default)            (polymorphic qual in CF)
      \      \               |                 /                  /
                           bottom
```

```
           unknowninitialized
             /            \
    underinitialized    initialized
                         (default)

             \            /
               fbcbottom
```

Figure 1 Combination of qualifiers. Two qualifier hierarchies are orthogonal. If an object is under initialization, its immutability guarantee is not satisfied. So even immutable and receiverdependantmutable objects can also be modifed when under initialization.

Subtype relations

$k_1\ q_1 <: k_2\ q_2 \iff k_1 <: k_2 \wedge q_1 <: q_2$

Helper Functions

$$\frac{q\ C\ f}{fType(f) = q}$$

Note:
1) No initialization modifier on field declarations. In actual implementation, to have circular initialization, *@NotOnlyInitialized* can be used on field declaration. However, it doesn't belong to initialization qualifier hierarchy.
2) The field is unique within the whole type hierarchy

fields(C) returns all fields directly declared in C.
cBody(kd) returns constructor body of kd.
mBody(md) returns method body of md.

Viewpoint Adaptation Rules

_ ▹ mutable = mutable
_ ▹ readonly = readonly
_ ▹ immutable = immutable
_ ▹ bottom = bottom
_ ▹ polymutable = substitutablepolymutable
q ▹ receiverdependantmutable = q

Note: substitutablepolymutable only exists shortly after viewpoint adaptation is done, but will must be subsituted by another qualifier immediately by QualifierPolymorphism. So, substitutablepolymutable should not appear on left or right side of viewpoint adaptation triangle.

Special Rules

- Forbid polymutable fields; readonly or polymutable constructor return type and readonly instantiation of objects
- In constructor, $q_{this} = q_{ret}$
- Forbid initialization modifier on fields, constructor return type and new statement
- Forbid bottom except on (implicit/explicit) lower bounds and null literal.
- Forbid explicit use of substitutablepolymutable everywhere.

TODO: Should we allow polymutable constructor return type?

Typing Rules

$$\frac{x \in \Gamma}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{l}\Gamma(x) = k_x\, q_x \quad fType(f) = q_f \quad q = q_x \rhd q_f \\ k = \begin{cases} \text{initialized} & \textit{if } k_x = \textit{initialized} \\ \text{unknowninitialized} & \textit{otherwise} \end{cases}\end{array}}{\Gamma \vdash x.f : k\, q} \quad \text{(T-FLD)}$$

Figure 2 Expression typing

$$\frac{\Gamma \vdash e = t_e \quad t_e <: \Gamma(x)}{\Gamma \vdash x = e} \quad \text{(T-VARASS)}$$

$$\frac{\begin{array}{l}\Gamma(x) = k_x\, q_x \quad \Gamma(y) = k_y\, q_y \quad typeof(f) = q_f \\ q_x = \text{mutable} \lor \\ q_f = \text{mutable} \lor \\ q_f = \text{readonly} \lor \\ (k_x = \text{underinitialized} \land q_x = \text{immutable}) \lor \\ (k_x = \text{underinitialized} \land q_x = \text{polyimmutable}) \\ q_y <: q_x . \rhd q_f \\ k_x = \text{underinitialized} \lor k_y = \text{initialized}\end{array}}{\Gamma \vdash x.f = y} \quad \text{(T-FLDASS)}$$

$\Gamma(x) = k_x\ q_x \quad \Gamma(y) = k_y\ q_y \quad \Gamma(\overline{z}) = \overline{k_z}\ \overline{q_z} \quad \text{typeof}(m) = k_{this}\ q_{this},\ \overline{k_p}\ \overline{q_p} \rightarrow k_{ret}\ q_{ret}$

$k_y <: k_{this} \quad \overline{k_z} <: \overline{k_p} \quad k_{ret} <: k_x$

$q_{this\text{-}vp} = q_y \rhd q_{this} \quad \overline{q}_{p\text{-}vp} = q_y \rhd \overline{q_p} \quad q_{ret\text{-}vp} = q_y \rhd q_{ret}$

$q_{this\text{-}vp} = substitutablepolymutable \lor \overline{q}_{p\text{-}vp} = substitutablepolymutable \lor q_{ret\text{-}vp} = substitutablepolymutable \Rightarrow$ s exists

$$q_y <: \begin{cases} q_{this\text{-}vp} & if\ \ q_{this\text{-}vp} \neq substitutablepolymutable \\ s & else \end{cases}$$

$$\overline{q_z} <: \begin{cases} \overline{q}_{p\text{-}vp} & if\ \ \overline{q}_{p\text{-}vp} \neq substitutablepolymutable \\ s & else \end{cases}$$

$$q_x :> \begin{cases} q_{ret\text{-}vp} & if\ \ q_{ret\text{-}vp} \neq substitutablepolymutable \\ s & else \end{cases}$$

$$\Gamma \vdash x = y.[s]m(\overline{z}) \qquad \text{(T-CALL)}$$

Note: inference of s is another subproblem. It is disussed in the last two pages.

kd in C $\quad$ C <: D $\quad \text{typeof}(D) = \overline{k}_{p\text{-}D}\ \overline{q}_{p\text{-}D} \rightarrow q_{ret\text{-}D} \quad \text{typeof}(kd) = \overline{\_\ \_} \rightarrow q_{ret\text{-}C}$

$$q_{ret\text{-}C} = \begin{cases} \_ & if\ \ q_{ret\text{-}D} = \text{receiverdependantmutable} \\ \text{immutable} & if\ \ q_{ret\text{-}D} = \text{immutable} \\ \text{mutable} & if\ \ q_{ret\text{-}D} = \text{mutable} \end{cases}$$

$\Gamma(z) = k_z\ q_z \quad \overline{k_z} <: \overline{k}_{p\text{-}D} \quad \overline{q_z} <: q_{ret\text{-}C} \rhd \overline{q}_{p\text{-}D}$

$$\Gamma \vdash super(\overline{z}) \text{ in kd} \qquad \text{(T-SUPER)}$$

* Previously, when $q_{ret\text{-}D}$ = mutable, $q_{ret\text{-}C}$ can still be immutable. Because at that time, immutable constructors only have immutable or polyimmutable(does not exist anymore), thus any mutable objet created locally cannot escape and be captured by outside objects; Neither outside mutable objects will be captured by the

receiverdependantmutable field when invoking mutable super constructor in immutable constructor. But now, immutable and receiverdependantmutable constructors don't have such restrictions(mutable parameters are allowed in both cases) any more, so outside mutable objects can be captured by receiverdependantmutable field. If we allow calling mutable super() in immutable subclass constructor, when we use $this_{sub}$.rdmf to access the field, the result is not guarantee to be immutable(may be the mutable object assigned in super mutable constructor). Therefore, we don't allow this kinds of flexibility and require subclass and superclass constructors should have the exact same qualifier if $q_{ret\text{-}D} \neq$ receiverdependantmutable

(T-THIS) (omitted)

* *Note:* In real Java code, one class can have multiple overloaded consturctors. One constructor can invoke the other by "this(..., ...)". The type rule T-THIS is very much the same as T-SUPER except that the constructor invoked by "this(..., ...)" comes from the same class.

$$\frac{\begin{array}{l}\Gamma(x) = k_x\ q_x \quad \Gamma(\overline{y}) = \overline{k_y}\ \overline{q_y} \quad typeof(C) = \overline{k_p}\ \overline{q_p} \rightarrow q_{ret} \\ \overline{q_y} <: q \triangleright \overline{q_P} \quad q <: q \triangleright q_{ret} \quad q \neq readonly \\ \overline{k_y} <: \overline{k_p} \\ q <: q_x \quad k <: k_x \qquad k = \begin{cases} initialized & \textit{if } \overline{k_p} = \textit{initialized} \\ underinitialized & \textit{otherwise} \end{cases}\end{array}}{\Gamma \vdash x = new\ q\ C(\overline{y})} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash s_1; s_2} \quad \text{(T-SEQ)}$$

Figure 3 Statement typing

Well-formdness Rules

$$\frac{fType(fd) \neq \text{polymutable} \qquad C <: D \qquad fd \notin fields(D)}{\vdash_C \ fd \text{ is OK}} \quad \text{(WF-FLD)}$$

$$\frac{}{\vdash_{object} \ kd \text{ is OK}} \quad \text{(WF-CONS-OBJECT)}$$

$$\frac{\begin{array}{l} cBody(kd) = super(g); this.f = f \qquad typeof(kd) = \overline{k_p}\ \overline{q_p} \rightarrow q_{ret} \\ q_{ret} \neq \text{readonly} \wedge q_{ret} \neq \text{polymutable} \\ \Gamma = (this : \text{underinitialized } q_{ret},\ \overline{p} : \overline{k_p}\ \overline{q_p},\ \overline{y} : \overline{k_{local}}\ \overline{q_{local}}) \\ \Gamma \vdash super(\overline{y}) \text{ in } kd \qquad \Gamma \vdash this.f = f \end{array}}{\vdash_C \ kd \text{ is OK}} \quad \text{(WF-CONS)}$$

Note: $\vdash_{C\ kd}$ reads “constructor kd in class C is well-formed”.

$$\frac{\begin{array}{l} mBody(md) = \overline{s}; return\ z \qquad typeof(md) = k_{this}\ q_{this},\ \overline{k_p}\ \overline{q_p} \rightarrow t_{ret} \\ \Gamma = (this : k_{this}\ q_{this},\ \overline{p} : \overline{k_p}\ \overline{q_p},\ \overline{y} : \overline{k_{local}}\ \overline{q_{local}}) \qquad \Gamma \vdash \overline{s} \qquad \Gamma(z) <: t_{ret} \\ \text{Standard method overriding rule} \end{array}}{\vdash_C \ md \text{ is OK}} \quad \text{(WF-METH)}$$

$$\frac{\vdash_C \overline{f} \text{ is OK} \qquad \vdash_C kd \text{ is OK} \qquad \vdash_C \overline{md} \text{ is OK}}{\vdash C \text{ is OK}} \quad \text{(WF-CLASS)}$$

Figure 4 Well-formdness typing

Extension to real Java

In real Java, there are static fields, static methods, initialization blocks.

Helper Method

usedQualifiers($\overline{s}$) returns all immutability qualifiers used in $\overline{s}$ recursively

cd ::= class C extends D { $\overline{sfd}$ $\overline{fd}$; $\overline{sib}$ $\overline{ib}$ kd $\overline{smd}$ $\overline{md}$ } class
sfd ::= static q C sf static field
smd ::= static t C sm (t C x) { $\overline{t\ C\ y}$ $\overline{s}$; return z; } static method
sib ::= static{$\overline{s}$;} static initialization block
ib ::= {$\overline{s}$;} initialization block

$$\frac{\text{fType(sfd)} \neq \text{polymutable} \wedge \text{fType(sfd)} \neq \text{receiverdependantmutable}}{\vdash \text{sfd is OK}} \quad \text{(WF-STATIC-FLD)}$$

$$\frac{\begin{array}{l} \text{mBody(smd)} = \overline{s}\text{;return z} \qquad \text{typeof(smd)} = \overline{k_p}\ \overline{q_p} \rightarrow t_{ret} \\ \Gamma = (\overline{p} : \overline{k_p}\ \overline{q_p},\ \overline{y} : \overline{k_{local}}\ \overline{q_{local}}) \qquad \Gamma \vdash \overline{s} \qquad \Gamma(z) <: t_{ret} \\ \overline{q_p} \neq \text{receiverdependantmutable} \wedge q_{ret} \neq \text{receiverdependantmutable} \\ \text{receiverdependantmutable} \notin \text{usedQualifiers}(\overline{s}\text{;return z}) \end{array}}{\vdash \text{smd is OK}}$$

(WF-STATIC-METH)

$$\frac{\Gamma \vdash \overline{s} \qquad \text{receiverdependantmutable} \notin \text{usedQualifiers}(\overline{s})}{\vdash \text{sib is OK}} \quad \text{(WF-STATIC-BLK)}$$

$$\frac{\Gamma \vdash \overline{s}}{\vdash_C \text{ ib is OK}} \quad \text{(WF-BLK)}$$

$$\frac{\vdash \overline{\text{sfd}} \text{ is OK} \quad \vdash_C \overline{\text{f}} \text{ is OK} \quad \vdash_C \text{kd is OK} \quad \vdash \overline{\text{smd}} \text{ is OK} \quad \vdash_C \overline{\text{md}} \text{ is OK} \quad \vdash \overline{\text{sib}} \text{ is OK} \quad \vdash_C \overline{\text{ib}} \text{ is OK}}{\vdash \text{C is OK}} \quad \text{(WF-CLASS)}$$

Inference of immutability qualifier for polymutable methods

After viewpoint adapting m() at the invocation site, if $q_{this\text{-}vp}$, $\overline{q}_{p\text{-}vp}$, $q_{ret\text{-}vp}$ are NOT substitutablepolymutable, standard subtyping rules apply:

$q_y <: q_{this\text{-}vp}$ $\qquad$ $\overline{q}_z <: \overline{q}_{p\text{-}vp}$ $\qquad$ $q_{ret\text{-}vp} <: q_x$

But if any of them is substitutablepolymutable, we use a variable **s** to replace corresponding $q_{this\text{-}vp}$, $\overline{q}_{p\text{-}vp}$, $q_{ret\text{-}vp}$ and add it/them to constraints set. After collecting all the constraints, we try to find a solution s that satisfies all the subtype constraints. If there is such a solution, then method invocation typechecks; Otherwise, it doesn't typecheck.

For example, assuming we have a method after viewpoint adaptation with signature:
substitutablepolymutable Object m(substitutablepolymutable A this, substitutablepolymutable Object p);
If we invoke it as:
immutable A a;
readonly Object ro = a.m(new immutable Object());

Constraints are collected in this way:
immutable <: s $\qquad$ immutable <: s $\qquad$ s <: readonly
We'll have a solution:
s = immutable(or readonly), so this method invocation typechecks.

But if we invoke the method as:

mutable Object ro = a.m(new immutable Object());
Then we have constraints:
immutable <: s   immutable <: s   s <: mutable
There is NO solution for s, so the type system rejects this method invocation.