Well-formedness
Formalization of all context conditions (*static semantics*)
“Type system++”

When do we have enough?

If nothing bad can happen during the execution of well-formed programs

How is “nothing bad can happen” formalized?
Execution does not get stuck (*type safety*, later)
Overview

- Definite assignment
- Method overriding
- Well-formedness
Definite assignment
Variables must be assigned to before use

Fields are automatically initialized with their default value upon object creation
**In theory**

**Thm** It is in general undecidable if a variable/field has been initialized before use.

**Proof** \( \text{if } (b) \ V := \text{Val } v \ \text{else } \text{unit}; \ \text{Var } V \)

1. \( V \) is initialized before use iff \( b \) does not evaluate to false.

2. Therefore a program analyzer would need to decide whether \( b \) can evaluate to false.

3. This is undecidable for arbitrary \( b \). Take “Turing machine \( k \) with input \( k \) terminates with output 0” (realized in Jinja as a search procedure). There is no program that can decide this for every \( k \).
In practice: variables

**Fact** It is easy to check if a variable has been initialized before use: if in doubt, be conservative.

Examples:

- \( \text{if } (b) \ V := \text{Val } v; \ 	ext{Var } V \)
  - is rejected because \( b \) could be false.
  - What if \( b \) is never false? Then \( \text{if} \) is pointless!

- \( \text{if } (b_1) \ V := \text{Val } v_1; \text{if } (b_2) \ V := \text{Val } v_2; \ 	ext{Var } V \)
  - is rejected because \( b_1 \) and \( b_2 \) could be false.

Rejection is not a problem:

Insert dummy initialization \( V := \text{Val } v \)
right after declaration of \( V \).
**Fact** It is hard to check if a field of an object has been initialized before use.

**Example:** Let $p$ be a formal parameter of class type. Inside the method body

- variable $p$ is initialized,
- but is field $F$ of $p$ initialized??

Therefore Java and Jinja initialize all fields of an object with their default value.
Function $\mathcal{A}$ (1)

$\mathcal{A} :: expr \Rightarrow vname\ set$

$\mathcal{A} e =$ the set of all (global) variables that are necessarily assigned to if $e$ terminates normally

$\mathcal{A} (new\ C) = \{\}$
$\mathcal{A} (Cast\ C\ e) = \mathcal{A} e$
$\mathcal{A} (Val\ v) = \{\}$
$\mathcal{A} (e_1 \ « bop »\ e_2) = \mathcal{A} e_1 \cup \mathcal{A} e_2$
$\mathcal{A} (Var\ V) = \{\}$
$\mathcal{A} (V := e) = \{V\} \cup \mathcal{A} e$
$\mathcal{A} (e.F\{D\}) = \mathcal{A} e$
$\mathcal{A} (e_1.F\{D\} := e_2) = \mathcal{A} e_1 \cup \mathcal{A} e_2$
$\mathcal{A} (e.M(\text{es})) = \mathcal{A} e \cup (\bigcup_{e \in \text{set}\ \text{es}} \mathcal{A} e)$
Function $\mathcal{A} \ (2)$

$$
\begin{align*}
\mathcal{A} \{ V : T ; e \} &= \mathcal{A} \ e - \{ V \} \\
\mathcal{A} \ (e_1 ; e_2) &= \mathcal{A} \ e_1 \cup \mathcal{A} \ e_2 \\
\mathcal{A} \ (if \ (e) \ e_1 \ else \ e_2) &= \mathcal{A} \ e \cup (\mathcal{A} \ e_1 \cap \mathcal{A} \ e_2) \\
\mathcal{A} \ (while \ (b) \ e) &= \mathcal{A} \ b \\
\mathcal{A} \ (throw \ e) &= \{ V \ | \ True \} \\
\mathcal{A} \ (try \ e_1 \ catch(C \ V) \ e_2) &= \mathcal{A} \ e_1 \cap (\mathcal{A} \ e_2 - \{ V \})
\end{align*}
$$

Motivation for $\text{throw}$:

$$
\mathcal{A} \ (if \ (e) \ V := Val \ v \ else \ throw \ e_2) = \{ V \}
$$
Function $\mathcal{D}$ (1)

$\mathcal{D} :: \text{expr} \Rightarrow \text{vname set} \Rightarrow \text{bool}$

$\mathcal{D} \ e \ A =$
if initially all variables in $A$ are initialized
then execution of $e$ does not access an uninitialized variable

$\mathcal{D} \ (\text{new } C) \ A = \text{True}$
$\mathcal{D} \ (\text{Cast } C \ e) \ A = \mathcal{D} \ e \ A$
$\mathcal{D} \ (\text{Val } v) \ A = \text{True}$
$\mathcal{D} \ (e_1 \ « \text{bop} » \ e_2) \ A = (\mathcal{D} \ e_1 \ A \land \mathcal{D} \ e_2 \ (A \cup A \ e_1))$
$\mathcal{D} \ (\text{Var } V) \ A = (V \in A)$
$\mathcal{D} \ (V ::= e) \ A = \mathcal{D} \ e \ A$
$\mathcal{D} \ (e \cdot F\{D\}) \ A = \mathcal{D} \ e \ A$
$\mathcal{D} \ (e_1 \cdot F\{D\}:={e_2}) \ A = (\mathcal{D} \ e_1 \ A \land \mathcal{D} \ e_2 \ (A \cup A \ e_1))$
Function $\mathcal{D}$ (2)

\[\mathcal{D} \{ V: T; e \} A = \mathcal{D} e (A \setminus \{ V \})\]

\[\mathcal{D} \langle e_1; e_2 \rangle A = (\mathcal{D} e_1 A \land \mathcal{D} e_2 (A \cup A e_1))\]

\[\mathcal{D} \langle \text{if} \langle e \rangle e_1 \text{ else } e_2 \rangle A =\]

\[\quad (\mathcal{D} e A \land \mathcal{D} e_1 (A \cup A e) \land \mathcal{D} e_2 (A \cup A e))\]

\[\mathcal{D} \langle \text{while} \langle b \rangle e \rangle A = (\mathcal{D} b A \land \mathcal{D} e (A \cup A b))\]

\[\mathcal{D} \langle \text{throw} \ e \rangle A = \mathcal{D} e A\]

\[\mathcal{D} \langle \text{try} \ e_1 \text{ catch} \langle C V \rangle e_2 \rangle A = (\mathcal{D} e_1 A \land \mathcal{D} e_2 (A \cup \{ V \}))\]
Each method body \((Vs,e)\) must fulfill the condition

\[\mathcal{D}e (\{\text{this}\} \cup \text{set Vs})\]
Correctness

**Thm** If $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$ then $\mathcal{A} e \subseteq \text{dom } l'$

**Proof** by rule induction over $\Rightarrow$

Correctness of $\mathcal{D}$: part of type safety proof (later)
A small imprecision

Let \( e = \text{if (Var } B) \{V:T; \text{throw } \_\} \text{ else } V := \text{true} \)

\[ A \{ V:T; \text{throw } \_\} = A (\text{throw } \_) - \{ V \} \]

\[ A e = A(\text{Var } B) \cup (A \{ V:T; \text{throw } \_\} \cap A(V := \text{true})) \]

\[ = \{ \} \]

\[ D(e; V := \text{Var } V) \{ B \} = \]

\[ (D e \{ B \} \land D(V := \text{Var } V) (\{ B \} \cup \{ \})) = \]

\[ (D e \{ B \} \land D(\text{Var } V) \{ B \}) = \]

\[ (D e \{ B \} \land V \in \{ B \}) = \text{False} \]

\[ A e = \{ \} \text{ is too pessimistic but not incorrect} \]
The problem

\[ A \{ V : T ; \text{throw } _\_ \} \]
should be \{ V \mid True\} and not \{ V \mid True\} – \{ V \}.

Possible solutions:

- Redefine \( A \) and \( D \). Complicated. See Jinja paper.
- Do not allow nested declaration of the same \( V \). Java!
  Then it does not matter if \( V \in A \{ V : T ; _\_ \} \) or not.

For this course: we leave things as they are and put up with the small imprecision.
Method overriding
class C extends B
{...
  method M(p:T):R = ...
  ...
}

class D extends C
{...
  method M(p:T'):R' = ...
  ...
}

M in D \textit{overrides} (overwrites, redefines) M in C
The question

How should $T$ and $T'$ be related?
How should $R$ and $R'$ be related?

Guiding principle: Type Safety

If $e$ is statically (at compile time) of class $C$
then the evaluation of $e$ should yield a subclass of $C$

Then $e.M(\ldots)$ can never fail at runtime as in SmallTalk:

Method not understood
Covariance in the result type

\[ R' \leq R \]

New result type must be subtype of old result type

Otherwise:

```java
class R' { }
class R extends R' { method M2() = ... }

{V:C; V := new D; V.M(...).M2()}
```

is type correct but semantics gets stuck.
Contravariance in the argument type

\[ T \leq T' \]

New argument type must be *supertype* of old argument type

Why?

1. Assume \( e \cdot M ( e' ) \) is statically well-typed.
2. Assume \( e \) has static type \( C \).
3. Then the static type of \( e' \) is \( \leq T \).
4. Now assume the dynamic type of \( e \) is \( D \).
5. Thus the dynamic type of \( e' \) must be \( \leq T' \).
6. But we can only guarantee \( e' \) has type \( T \).
7. \( \implies T \leq T' \)
Example

class C extends B
{  method M(p:T):R = ...  }

class D extends C
{  method M(p:T'):R = p.M2()  }

class T {  }

class T' extends T {  method M2():R = ...  }

Problem?

{V:C; V := new D; V.M(new T)}

is type correct but semantics gets stuck.
The set theoretic perspective

For total functions:

\[
\begin{align*}
T & \subseteq T' & R' & \subseteq R \\
T' & \rightarrow R' & \subseteq T & \rightarrow R
\end{align*}
\]
**Terminology**

\[ \_ \rightarrow \_ \text{ is } \textit{contravariant} \text{ in the first argument} \]

\[ \_ \rightarrow \_ \text{ is } \textit{covariant} \text{ in the second argument} \]

Alternatively: \textit{antimonotone}, \textit{monotone}
**Method overriding in Jinja, Java and Eiffel**

**Jinja**  Contravariant in parameters, covariant in result

**Java**  Invariant:

\[
T = T' \quad \text{then} \quad R = R'
\]

else overloading, not overriding.

Overloading: multiple methods with the same name but different parameter types are visible at the same time. Use static type of actual parameters in method selection (tricky).

**Eiffel**  Covariant in parameters and result (not type safe)
class Point
{field x: Integer
  method eq(p:Point):Boolean = (this.x = p.x)
}

class ColPoint extends Point
{field col: Integer
  method eq(cp:ColPoint):Boolean =
    (if (this.x = cp.x) (this.col = cp.col)
      else false)
}
Java versus Eiffel

```
{p1:ColPoint; p1 := new ColPoint;
 p1.x := 5; p1.col := 0;
 {p2:Point; p2 := new ColPoint;
  p2.x := 5; p2.col := 1;
  p1.eq(p2) }}
```
evaluates to

- in Java: `true`
- in Eiffel: `false`
- in Jinja: `eq in ColPoint not wellformed!`

How can we program `ColPoint` in Jinja?
ColPoint in Jinja

class ColPoint extends Point
{
    field col: Integer
    method eqCP(cp: ColPoint): Boolean =
        (if (this.x = cp.x) (this.col = cp.col)
         else false)
}

Similar to Java: make argument type part of method name. In Java this happens implicitly via overloading.
Well-formedness
Well-formed program $P$

$\text{wf-J-prog } P$

- The system classes are declared:
  $$\{ \text{Object, } \text{NullPointer, ClassCast, OutOfMemory} \} \subseteq \text{set}(\text{map fst } P)$$

- No class is declared twice:
  $$\text{distinct } (\text{map fst } P)$$

- Every class declaration $$(C, D, fs, ms) \in \text{set } P$$ is well-formed.
Well-formed class declaration \((C, D, fs, ms)\)

- All field declarations in \(fs\) are well-formed
- No field in \(fs\) is declared twice
- All method declarations in \(ms\) are well-formed
- No method in \(ms\) is declared twice
- If \(C \neq Object\) then
  - \(D\) is a class in \(P\)
  - \(D\) is not a subclass of \(C\): \(\neg P \vdash D \not\subseteq^* C\)
  - **Overriding:** if \((M, Ts, T, mb) \in set ms\) and \(P \vdash D\) sees \(M\): \(Ts' \rightarrow T' = mb'\) in \(D'\)
    then \(P \vdash T \leq T'\) and \(P \vdash Ts' [\leq] Ts\)
    Method overriding is **covariant** in the result type and **contravariant** in the argument types
**Well-formed method declaration** \((M, Ts, T, Vs, e)\)

- All parameter types are valid: \(\forall T \in \text{set Ts. is-type } P T\)
- The result type is valid: \(\text{is-type } P T\)
- There are as many parameter types as names: \(|Ts| = |Vs|\)
- All parameter names are distinct: \(\text{distinct } Vs\)
- \textit{this} is not a parameter name: \(\text{this} \notin \text{set } Vs\)
- The method body is well-typed:
  \[
  \exists T'. P, [\textit{this} \mapsto \text{Class } C, Vs \mapsto Ts] \vdash e :: T' \land P \vdash T' \leq T
  \]
- All local variables are assigned to before use:
  \(\mathcal{D} e (\{\textit{this}\} \cup \text{set } Vs)\)
Well-formed field declaration \((F, T)\)

The type is valid: \(\text{is-type } P\ T\)