Well-formedness

Aim

Formalization of all context conditions (*static semantics*)

"Types and more"

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

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 e; $\{V{:}T; \texttt{Var}\ V\}$

1. V is not initialized before use iff e terminates (normally).
2. Therefore a program analyzer would need to decide whether e terminates.
3. But it is in general undecidable if a given Turing machine/program/expression e terminates.

In practice: variables

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

Examples:

- $\mathtt{if}\ (b)\ V := \mathtt{Val}\ v\ \mathtt{else}\ \mathit{unit};\ \mathtt{Var}\ V$
  is rejected because *b* could be false.
  What if *b* is never false? Then `if` is pointless!
- $\mathtt{throw}\ e;\ \{V{:}T;\ \mathtt{Var}\ V\}$
  is rejected.

Rejection is not a problem:

Insert dummy initialization $V := \mathit{Val}\ v$
right after declaration of *V*.

In practice: fields

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??

Java and Jinja initialize all fields of an object with their default value.

$\mathcal{A}$

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

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

$$
\begin{aligned}
\mathcal{A}\ (\texttt{new}\ C) &= \emptyset \\
\mathcal{A}\ (\texttt{Cast}\ C\ e) &= \mathcal{A}\ e \\
\mathcal{A}\ (\texttt{Val}\ v) &= \emptyset \\
\mathcal{A}\ (e_1 \ll bop \gg e_2) &= \mathcal{A}\ e_1 \cup \mathcal{A}\ e_2 \\
\mathcal{A}\ (\texttt{Var}\ V) &= \emptyset \\
\mathcal{A}\ (V := e) &= \{V\} \cup \mathcal{A}\ e \\
\mathcal{A}\ (e.F\{D\}) &= \mathcal{A}\ e \\
\mathcal{A}\ (e_1.F\{D\} := e_2) &= \mathcal{A}\ e_1 \cup \mathcal{A}\ e_2 \\
\mathcal{A}\ (e.M(es)) &= \mathcal{A}\ e \cup (\textstyle\bigcup_{e \in set\ es} \mathcal{A}\ e)
\end{aligned}
$$

$\mathcal{A}$

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

Motivation for *throw*:

$$\mathcal{A}\ (\texttt{if}\ (e)\ V \texttt{:=}\ \texttt{Val}\ v\ \texttt{else}\ \texttt{throw}\ e_2) = \mathcal{A}\ e \cup \{V\}$$

$\mathcal{D}$

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

$\mathcal{D}\ e\ A$ =

if initially all variables in A are initialized

then execution of e does not access an uninitialized variable

$$
\begin{aligned}
&\mathcal{D}\ (\texttt{new}\ C)\ A &&= True\\
&\mathcal{D}\ (\texttt{Cast}\ C\ e)\ A &&= \mathcal{D}\ e\ A\\
&\mathcal{D}\ (\texttt{Val}\ v)\ A &&= True\\
&\mathcal{D}\ (e_1 \text{ «}bop\text{» } e_2)\ A &&= (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1))\\
&\mathcal{D}\ (\texttt{Var}\ V)\ A &&= (V \in A)\\
&\mathcal{D}\ (V := e)\ A &&= \mathcal{D}\ e\ A\\
&\mathcal{D}\ (e.F\{D\})\ A &&= \mathcal{D}\ e\ A\\
&\mathcal{D}\ (e_1.F\{D\} := e_2)\ A &&= (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1))
\end{aligned}
$$

$\mathcal{D}$

$$
\begin{array}{ll}
\mathcal{D}\ \{V\!:\!T;\ e\}\ A & = \mathcal{D}\ e\ (A - \{V\}) \\
\mathcal{D}\ (e_1\texttt{;}\ e_2)\ A & = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e_1)) \\
\mathcal{D}\ (\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2)\ A & = \\
\quad (\mathcal{D}\ e\ A \wedge \mathcal{D}\ e_1\ (A \cup \mathcal{A}\ e) \wedge \mathcal{D}\ e_2\ (A \cup \mathcal{A}\ e)) & \\
\mathcal{D}\ (\texttt{while}\ (b)\ e)\ A & = (\mathcal{D}\ b\ A \wedge \mathcal{D}\ e\ (A \cup \mathcal{A}\ b)) \\
\mathcal{D}\ (\texttt{throw}\ e)\ A & = \mathcal{D}\ e\ A \\
\mathcal{D}\ (\texttt{try}\ e_1\ \texttt{catch}\ (C\ V)\ e_2)\ A & = (\mathcal{D}\ e_1\ A \wedge \mathcal{D}\ e_2\ (A \cup \{V\}))
\end{array}
$$

Well-formedness (sneak preview)

Each method body (Vs, e) must fulfill the condition

$$\mathcal{D}\ e\ (\{this\} \cup set\ Vs)$$

Correctness

Thm If $P \vdash \langle e,(h, l)\rangle \Rightarrow \langle e',(h', l')\rangle$ then $\mathcal{A}\ e \ \subseteq \mathit{dom}\ l'$
Proof by rule induction over $\Rightarrow$

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

A small imprecision

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

$\mathcal{A}\ \{V{:}T;\ \texttt{throw}\ \_\} = \mathcal{A}\ (\texttt{throw}\ \_) - \{V\}$

$\mathcal{A}\ e = \mathcal{A}(\mathit{Var}\ B) \cup (\mathcal{A}\ \{V{:}T;\ \texttt{throw}\ \_\} \cap \mathcal{A}(V := \mathit{true}))$
$= \{\}$

$\mathcal{D}\ (e;\ \texttt{Var}\ V)\ \{B\} =$
$(\mathcal{D}\ e\ \{B\} \wedge \mathcal{D}(\mathit{Var}\ V)\ (\{B\} \cup \{\})) =$
$(\mathcal{D}\ e\ \{B\} \wedge V \in \{B\}) = \mathit{False}$

$\mathcal{A}\ e = \{\}$ is too pessimistic but not incorrect

The problem

$\mathcal{A}\,\{V{:}T;\,\texttt{throw}\,\_\}$
should be $\{V \mid \mathit{True}\}$ and not $\{V \mid \mathit{True}\} - \{V\}$.

Posible solutions:

- Redefine $\mathcal{A}$ and $\mathcal{D}$.
- Do not allow nested declaration of the same V. Java!
  Then it does not matter if $V \in \mathcal{A}\,\{V{:}T;\,\_\}$ or not.

We leave things as they are and put up with the small imprecision.

Method overriding

The situation

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

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

`M` in `D` *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

$$\mathtt{R'} \leq \mathtt{R}$$

New result type must be *subtype* of old result type

Otherwise:

```
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

$$\mathtt{T} \leq \mathtt{T'}$$

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

Why?

1. Assume `e.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. $\Longrightarrow \mathtt{T} \leq \mathtt{T'}$

Example

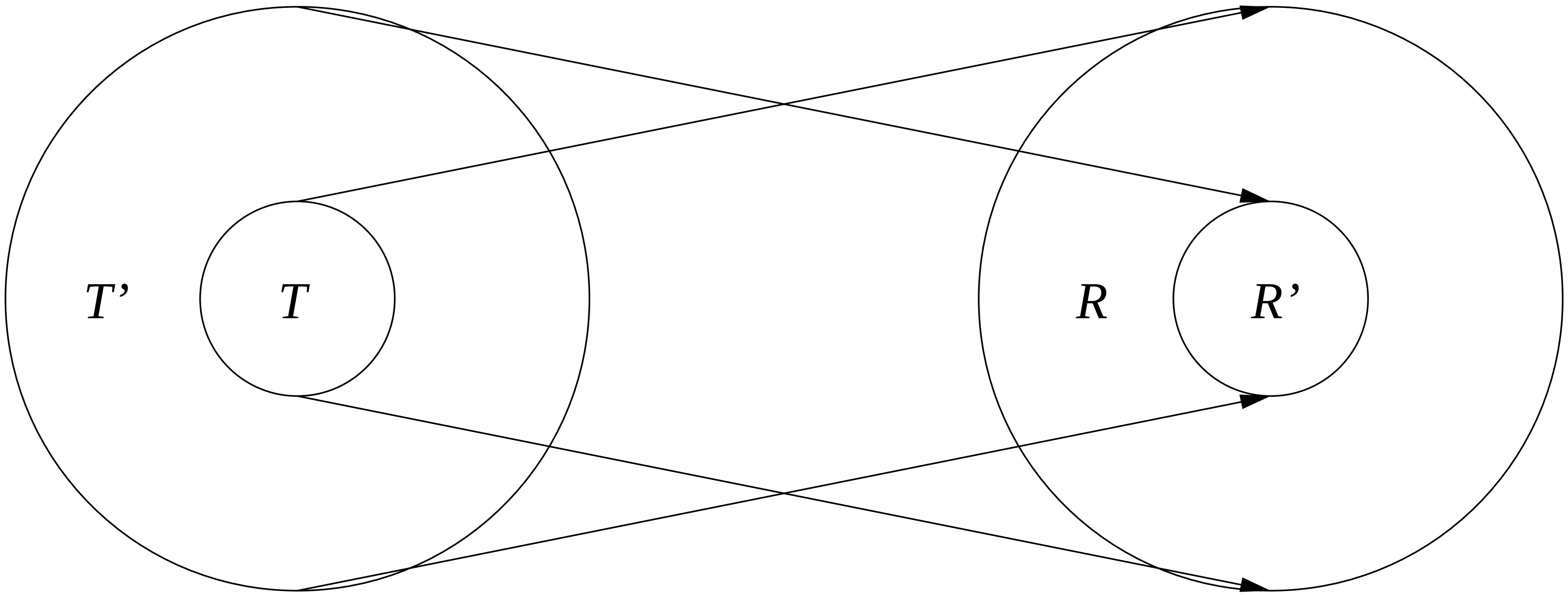
```
class C extends B
{ method M(p:T):R = ... }

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:

$$\frac{T \subseteq T' \quad R' \subseteq R}{T' \to R' \subseteq T \to R}$$

Terminology

$\_ \rightarrow \_$ is	*contravariant*	in the	first	argument
$\_ \rightarrow \_$ is	*covariant*	in the	second	argument

Alternatively: *antimonotone*, *monotone*

Method overriding in Jinja, Java and Eiffel

Jinja Contravariant in parameters, covariant in result

Classic Java Invariant:
If `T=T'` then `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).

Java 1.5 Invariant in parameters, covariant in result

Eiffel Covariant in parameters and result (not type safe)

An example

```
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

wf-J-prog P

- The system classes are declared:
  $\{$*Object*, *NullPointer*, *ClassCast*, *OutOfMemory*$\}$
  $\subseteq$ *set*(*map fst P*)
- No class is declared twice:
  distinct (*map fst P*)
- Every class declaration $(C, D, fs, ms) \in$ *set P* is well-formed.

Well-formed class declaration $(\mathbf{C},\mathbf{D},\mathbf{fs},\mathbf{ms})$

- 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 $C \neq \mathit{Object}$ then
  - D is a class in P
  - D is not a subclass of C: $\neg\ P \vdash D \preceq^* C$
  - *Overriding:* if $(M, Ts, T, mb) \in \mathit{set}\ \mathit{ms}$ 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 $(\mathbf{M}, \mathbf{Ts}, \mathbf{T}, \mathbf{Vs}, \mathbf{e})$

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

Well-formed field declaration (F, T)

The type is valid: *is-type P T*