Contents & Goals

Last Lecture:

- Representing class diagrams as (extended) signatures — for the moment without associations (see Lectures 07 and 08).
- **Insight:** visibility doesn’t contribute to semantics in the sense that if $\mathcal{I}_1$ and $\mathcal{I}_2$ only differ in visibility of some attributes, then $\Sigma_{\mathcal{D}_1} = \Sigma_{\mathcal{D}_2}$ for each $\mathcal{D}$.
- **And:** in Lecture 03, implicit assumption of well-typedness of OCL expressions.

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - Is this OCL expression well-typed or not? Why?
  - How/in what form did we define well-definedness?
  - What is visibility good for?

- **Content:**
  - Recall: type theory/static type systems.
  - Well-typedness for OCL expression.
  - Visibility as a matter of well-typedness.
Excursus: Type Theory (cf. Thiemann, 2008)
Type Theory

Recall: In lecture 03, we introduced OCL expressions with types, for instance:

\[ expr ::= w : \tau \quad \ldots \text{logical variable } w \]
\[ | \text{true} | \text{false} : \text{Bool} \quad \ldots \text{constants} \]
\[ | 0 | -1 | 1 | \ldots : \text{Int} \quad \ldots \text{constants} \]
\[ | \text{expr}_1 + \text{expr}_2 : \text{Int} \times \text{Int} \to \text{Int} \quad \ldots \text{operation} \]
\[ | \text{size}(\text{expr}_1) : \text{Set}(\tau) \to \text{Int} \]

Wanted: A procedure to tell well-typed, such as \((w : \text{Bool})\)
not \(w\)
from not well-typed, such as,
size(\(w\)).

Approach: Derivation System, that is, a finite set of derivation rules.
We then say \(expr \text{ is well-typed}\) if and only if we can derive

\[ A, C \vdash expr : \tau \quad \text{(read: \text{“expression } expr \text{ has type } \tau \text{”})} \]

for some OCL type \(\tau\), i.e. \(\tau \in T_B \cup T_{\mathcal{C}} \cup \{\text{Set}(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathcal{C}}\}\), \(C \in \mathcal{C}\).
A Type System for OCL
A Type System for OCL

We will give a finite set of type rules (a type system) of the form

\[(\text{"name"}) \quad \text{"premises"} \quad \text{"conclusion"} \quad \text{"side condition"}\]

These rules will establish well-typedness statements (type sentences) of three different "qualities":

(i) Universal well-typedness:

\[\vdash \text{expr} : \tau\]
\[\vdash 1 + 2 : \text{Int}\]

(ii) Well-typedness in a type environment \(A\): (for logical variables)

\[\vdash_A \text{expr} : \tau\]
\[\vdash \text{self} : \tau_C \vdash \text{self}.v : \text{Int}\]

(iii) Well-typedness in type environment \(A\) and context \(D\): (for visibility)

\[\vdash_{A,D} \text{expr} : \tau\]
\[\vdash \text{self} : \tau_C, C \vdash \text{self}.r.v : \text{Int}\]
If \( expr \) is a boolean constant, then \( expr \) is of type \( \text{Bool} \):

\[
\text{(BOOL)} \quad \vdash B : \text{Bool}, \quad B \in \{\text{true, false}\}
\]
Constants and Operations

- If \( expr \) is a **boolean constant**, then \( expr \) is of type \( \text{Bool} \):
  \[
  \frac{\text{(BOOL)}}{\vdash B : \text{Bool}, \quad B \in \{ \text{true}, \text{false} \}}
  \]

- If \( expr \) is an **integer constant**, then \( expr \) is of type \( \text{Int} \):
  \[
  \frac{\text{(INT)}}{\vdash N : \text{Int}, \quad N \in \{ 0, 1, -1, \ldots \}}
  \]

- If \( expr \) is the application of **operation** \( \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \) to expressions \( expr_1, \ldots, expr_n \) which are of type \( \tau_1, \ldots, \tau_n \), then \( expr \) is of type \( \tau \):
  \[
  \frac{\text{(Fun}_0\text{)}}{\vdash expr_1 : \tau_1 \ldots \vdash expr_n : \tau_n \quad \vdash \omega(expr_1, \ldots, expr_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr(C)}}
  \]
  (Note: this rule also covers ‘\( =_{\tau} \)’, ‘isEmpty’, and ‘size’.)
Constants and Operations Example

(BOOL) \[ \vdash B : \text{Bool}, \quad B \in \{\text{true, false}\} \]

(INT) \[ \vdash N : \text{Int}, \quad N \in \{0, 1, -1, \ldots\} \]

(Fun) \[ \vdash \omega(\text{expr}_1, \ldots, \text{expr}_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{C}) \]

Example:

- \( \text{not(true)} \)
- \( \text{not("hello")} \)
- \( \text{true} + 3 \)

\[ \vdash \text{true} : \text{Int} \]

\[ \vdash \text{true} + 3 : \text{Int} \]

Thus \( \text{not(true)} \) is well-typed.

GOT STUCK, NO RICE TO SHOW

Thus \( \text{true} + 3 \) IS NOT WELL-TYPED
**Type Environment**

- **Problem:** Whether

  \[ w + 3 \]

  is well-typed or not depends on the type of logical variable \( w \in W \).

- **Approach:** **Type Environments**

  **Definition.** A type environment is a (possibly empty) finite sequence of type declarations. The set of type environments for a given set \( W \) of logical variables and types \( T \) is defined by the grammar

  \[
  A ::= \emptyset \mid A, w : \tau
  \]

  where \( w \in W, \tau \in T \).

  **Clear:** We use this definition for the set of OCL logical variables \( W \) and the types \( T = T_B \cup T_\mathcal{E} \cup \{ Set(\tau_0) \mid \tau_0 \in T_B \cup T_\mathcal{E} \} \).
Environment Introduction and Logical Variables

- If $expr$ is of type $\tau$, then it is of type $\tau$ in any type environment:

\[
\frac{}{A \vdash expr : \tau}
\]

\[
(EnvIntro)
\]

- Care for logical variables in sub-expressions of operator application:

\[
(\text{Fun}_1) \quad \frac{A \vdash expr_1 : \tau_1 \ldots A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{attr}(C)
\]

- If $expr$ is a logical variable such that $w : \tau$ occurs in $A$, then we say $w$ is of type $\tau$,

\[
(Var) \quad \frac{w : \tau \in A}{A \vdash w : \tau}
\]
### Type Environment Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Hypothesis</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(EnvIntro)</td>
<td>( A \vdash \text{expr} : \tau )</td>
<td>( A \vdash \text{expr} : \tau )</td>
</tr>
<tr>
<td>(Fun₁)</td>
<td>( A \vdash \text{expr}_1 : \tau_1 \ldots A \vdash \text{expr}_n : \tau_n )</td>
<td>( A \vdash \omega(\text{expr}_1, \ldots, \text{expr}_n) : \tau ), ( \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau ), ( n \geq 1 ), ( \omega \notin \text{atr}(\mathcal{E}) )</td>
</tr>
<tr>
<td>(Var)</td>
<td>( w : \tau \in A )</td>
<td>( A \vdash w : \tau )</td>
</tr>
</tbody>
</table>

### Example:
- \( w + 3, A = w : \text{Int} \)
All Instances and Attributes in Type Environment

- If $expr$ refers to all instances of class $C$, then it is of type $Set(\tau_C)$,

\[
(AllInst) \quad \vdash allInstances_C : Set(\tau_C)
\]

- If $expr$ is an attribute access of an attribute of type $\tau$ for an object of $C$ as denoted by $expr_1$, then the premise is that $expr_1$ is of type $\tau_C$:

\[
(Attr_0) \quad A \vdash expr_1 : \tau_C, \quad \vdash v(expr_1) : \tau, \quad v : \tau \in atr(C), \tau \in \mathcal{T}
\]

\[
(Attr_{0,1}) \quad A \vdash expr_1 : \tau_C, \quad \vdash r_1(expr_1) : \tau_D, \quad r_1 : D_{0,1} \in atr(C)
\]

\[
(Attr^{*}_{0}) \quad A \vdash expr_1 : \tau_C, \quad \vdash r_2(expr_1) : Set(\tau_D), \quad r_2 : D^{*} \in atr(C)
\]
Attributes in Type Environment Example

\[(Attr_0)\]
\[
A \vdash expr_1 : \tau_C,\quad v : \tau \in \text{atr}(C), \tau \in \mathcal{T}
\]

\[(Attr_0^{0,1})\]
\[
A \vdash expr_1 : \tau_C,\quad r_1 : D_{0,1} \in \text{atr}(C)
\]

\[(Attr_0^\ast)\]
\[
A \vdash expr_1 : \tau_C,\quad r_2 : D_\ast \in \text{atr}(C)
\]

\[
\begin{array}{|c|}
\hline
C \\
\hline
\hline
x : \text{Int} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
D \\
\hline
\hline
y : \text{Int} \\
\hline
\end{array}
\]

- \(\text{self} : \tau_C \vdash \text{self}.x \checkmark : \text{Int}\)

- \(\text{self} : \tau_C \vdash \text{self}.r.x \times \text{syntax error!}\)

- \(\text{self} : \tau_C \vdash \text{self}.r.y : \text{Int}\)

- \(\text{self} : \tau_D \vdash \text{self}.x : \text{Int}\)
If \( expr \) is an \textbf{iterate expression}, then

- the iterator variable has to be type consistent with the base set, and
- initial and update expressions have to be consistent with the result variable:

\[
\frac{A \vdash \text{expr}_1 : \text{Set}(\tau_1) \quad A \vdash \text{expr}_2 : \tau_2 \quad A' \vdash \text{expr}_3 : \tau_2}{A \vdash \text{expr}_1 \text{->iterate}(w_1 : \tau_1; w_2 : \tau_2 = \text{expr}_2 | \text{expr}_3) : \tau_2}
\]

where \( A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2) \).
Better evaluate `expS2` in the outer scope (A) instead of A' as `expS2` needs to be evaluated even with empty base set (as given by `expV1`).
Iterate Example

((AllInst) \[\frac{}{\vdash \text{allInstances}_C : \text{Set}(\tau_C)}\])

((Attr) \[\frac{A \vdash \text{expr}_1 : \tau_C \quad A \vdash \nu(\text{expr}_1) : \tau}{\vdash \nu(\text{expr}_1)}\])

((Iter) \[\frac{A \vdash \text{expr}_1 : \text{Set}(\tau_1) \quad A' \vdash \text{expr}_2 : \tau_2 \quad A' \vdash \text{expr}_3 : \tau_2}{A \vdash \text{expr}_1 \rightarrow \text{iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = \text{expr}_2 \mid \text{expr}_3) : \tau_2} \]

where \(A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2)\).

Example: \(\mathcal{S} = (\{\text{Int}\}, \{C\}, \{x : \text{Int}\}, \{C \mapsto \{x\}\})\)

allInstances\(_C \rightarrow \text{iterate}(self : C ; w : \text{Bool} = \text{true} \mid w \land self . x = 0)\)

allInstances\(_C \rightarrow \text{forAll}(self : C \mid self . x = 0)\)

context \text{self} : C \ inv : \text{self} . x = 0

context \text{C} \ inv : x = 0
First Recapitulation

- **I only** defined for well-typed expressions.

- **What can hinder** something, which looks like a well-typed OCL expression, from being a well-typed OCL expression...?

\[ \mathcal{L} = (\{\text{Int}\}, \{C, D\}, \{x : \text{Int}, n : D_{0,1}\}, \{C \mapsto \{n\}, \{D \mapsto \{x\}\}) \]

- **Plain syntax error:**
  
  context \( C : \text{false} \)

- **Subtle syntax error:**
  
  context \( C \ inv : y = 0 \)

- **Type error:**
  
  context \( self : C \ inv : self . n = self . n . x \)
Casting in the Type System
One Possible Extension: Implicit Casts

• We **may wish** to have

\[ \vdash 1 \text{ and } false : Bool \]  

(\(*\))

**In other words:** We may wish that the type system allows to use 0, 1 : Int instead of true and false without breaking well-typedness.

• Then just have a rule:

\[
\begin{align*}
(Cast) & \quad \frac{A \vdash expr : Int}{A \vdash expr : Bool}
\end{align*}
\]

• With (Cast) (and (Int), and (Bool), and (Fun\(_0\))), we can derive the sentence (\(*\)), thus conclude well-typedness.

• **But:** that’s only half of the story — the definition of the interpretation function \( I \) that we have is not prepared, it doesn’t tell us what (\(*\)) means...
So, why isn’t there an interpretation for (1 and false)?

- First of all, we have (syntax)

\[ expr_1 \text{ and } expr_2 : \text{Bool} \times \text{Bool} \to \text{Bool} \]

- Thus,

\[ I(\text{and}) : I(\text{Bool}) \times I(\text{Bool}) \to I(\text{Bool}) \]

where \( I(\text{Bool}) = \{\text{true, false}\} \cup \{\bot_{\text{Bool}}\} \).

- By definition,

\[ I[1 \text{ and } \text{false}](\sigma, \beta) = I(\text{and})(I[1](\sigma, \beta), I[\text{false}](\sigma, \beta)) , \]

and there we’re stuck.
Implicit Casts: Quickfix

- Explicitly define

\[ I[\text{and}(expr_1, expr_2)](\sigma, \beta) := \begin{cases} b_1 \land b_2, & \text{if } b_1 \neq \bot_{\text{Bool}} \neq b_2 \\ \bot_{\text{Bool}}, & \text{otherwise} \end{cases} \]

where

- \( b_1 := \text{toBool}(I[expr_1](\sigma, \beta)) \),
- \( b_2 := \text{toBool}(I[expr_2](\sigma, \beta)) \),

and where

\[ \text{toBool} : I(\text{Int}) \cup I(\text{Bool}) \to I(\text{Bool}) \]

\[ x \mapsto \begin{cases} \text{true}, & \text{if } x \in \{ \text{true} \} \cup I(\text{Int}) \setminus \{0, \bot_{\text{Int}} \} \\ \text{false}, & \text{if } x \in \{ \text{false}, 0 \} \\ \bot_{\text{Bool}}, & \text{otherwise} \end{cases} \]
There are wishes for the type-system which require changes in both, the definition of $I$ and the type system. In most cases not difficult, but tedious.

*Note:* the extension is still a basic type system.

*Note:* OCL has a far more elaborate type system which in particular addresses the relation between $Bool$ and $Int$ (cf. [OMG, 2006]).
Visibility in the Type System
Visibility — The Intuition

Let's study an Example:

\[ \mathcal{L} = (\{\text{Int}\}, \{C, D\}, \{n : D_{0,1}, m : D_{0,1}, \langle x : \text{Int}, \xi, \text{expr}_0, \emptyset \rangle\}, \{C \mapsto \{n\}, D \mapsto \{x, m\}\} \]

and

Assume \( w_1 : \tau_C \) and \( w_2 : \tau_D \) are logical variables. Which of the following syntactically correct (?) OCL expressions shall we consider to be well-typed?

<table>
<thead>
<tr>
<th>( \xi ) of ( x ):</th>
<th>public</th>
<th>private</th>
<th>protected</th>
<th>package</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w_1 \cdot n \cdot x = 0 )</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
<td>later</td>
</tr>
<tr>
<td></td>
<td>✗</td>
<td>✓</td>
<td>?</td>
<td>later</td>
</tr>
<tr>
<td>( w_2 \cdot m \cdot x = 0 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>later</td>
</tr>
<tr>
<td></td>
<td>✗</td>
<td>✗</td>
<td>?</td>
<td>not</td>
</tr>
</tbody>
</table>
**Example:** A problem?

![Diagram](image)

\[
\text{self} : \tau_D \vdash \text{self} . r . v > 0
\]

\[
\text{self} : \tau_C \nvdash \text{self} . r . v > 0
\]

That is, whether an expression involving attributes with visibility is well-typed **depends** on the class of objects for which it is evaluated.

**Therefore:** well-typedness in type environment \( A \) and **context** \( D \in C \):

\[
A, D \vdash expr : \tau
\]

In a sense, already preparing to treat “protected” later (when doing inheritance).
Attribute Access in Context

- If \( expr \) is of type \( \tau \) in a type environment, then it is in any context:

\[
\frac{A \vdash expr : \tau}{A, D \vdash expr : \tau}
\]

\((\text{ContextIntro})\)

- Accessing an attribute \( v \) of an object of class \( C \) is well-typed:
  - if \( v \) is public, or
  - if the expression \( expr_1 \) denotes an object of class \( C \):

\[
\frac{A, D \vdash expr_1 : \tau_C}{A, D \vdash v(expr_1) : \tau}, \quad \langle v : \tau, \xi, expr_0, P_\xi \rangle \in \text{atr}(C), \quad \xi = +, \text{ or } \xi = - \text{ and } C = D
\]

\((\text{Attr}_1)\)

- Accessing \( C_{0,1} \)- or \( C^* \)-typed attributes: similar.
Attribute Access in Context Example

\[
\frac{A \vdash \text{expr} : \tau}{A, D \vdash \text{expr} : \tau}
\]

\[
\frac{A, D \vdash \text{expr}_1 : \tau_C}{A, D \vdash v(\text{expr}_1) : \tau}, \quad \langle v : \tau, \xi, \text{expr}_0, P_\xi \rangle \in \text{atr}(C), \quad \xi = +, \text{ or } \xi = - \text{ and } C = D
\]

Example:

\[
\text{self} : \tau_C, C \vdash \text{self}.r.v > 0
\]

Thus not well typed.
The Semantics of Visibility

- **Observation:**
  - Whether an expression *does* or *does not* respect visibility is a matter of well-typedness **only**.
  
  - We only evaluate (= apply $I$ to) **well-typed** expressions.

  → We **need not** adjust the interpretation function $I$ to support visibility.
What is Visibility Good For?

- Visibility is a property of attributes — is it useful to consider it in OCL?

- In other words: given the picture above, is it useful to state the following invariant (even though $x$ is private in $D$)

  \[
  \text{context } C \text{ inv } : n.x > 0 \ ?
  \]
What is Visibility Good For?

- Visibility is a property of attributes — is it useful to consider it in OCL?

- In other words: given the picture above, is it useful to state the following invariant (even though $x$ is private in $D$)

$$\text{context } C \text{ inv } : n.x > 0 ?$$

- It depends.

  (cf. [OMG, 2006], Sect. 12 and 9.2.2)

- Constraints and pre/post conditions:
  - Visibility is sometimes not taken into account. To state “global” requirements, it may be adequate to have a “global view”, be able to look into all objects.
  - But: visibility supports “narrow interfaces”, “information hiding”, and similar good design practices. To be more robust against changes, try to state requirements only in the terms which are visible to a class.

  **Rule-of-thumb**: if attributes are important to state requirements on design models, leave them public or provide get-methods (later).

- Guards and operation bodies:
  If in doubt, yes (= do take visibility into account).

  Any so-called **action language** typically takes visibility into account.
Recapitulation
Recapitulation

Class Diagrams $\mathcal{CD}$ induces extended (!) signature $\mathcal{I}(\mathcal{CD})$ gives rise to Basic Type System

- We extended the type system for
  - casts (requires change of $I$) and
  - visibility (no change of $I$).
- Later: navigability of associations.

**Good**: well-typedness is decidable for these type-systems. That is, we can have automatic tools that check, whether OCL expressions in a model are well-typed.
References
References

