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 $\mathcal{P}_1$ and $\mathcal{P}_2$ only differ in visibility of some attributes, then $\Sigma_{\mathcal{P}_1} = \Sigma_{\mathcal{P}_2}$ for each $\mathcal{P}$.
- **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.
Recall: In lecture 03, we introduced OCL expressions with types, for instance:

\[ expr ::= \begin{array}{ll}
w & : \tau \quad \ldots \text{logical variable } w \\
\text{true | false} & : \text{Bool} \quad \ldots \text{constants} \\
0 | -1 | 1 | \ldots & : \text{Int} \quad \ldots \text{constants} \\
expr_1 + expr_2 & : \text{Int x Int } \rightarrow \text{Int} \quad \ldots \text{operation} \\
\text{size}(expr_1) & : \text{Set}(\tau) \rightarrow \text{Int}
\end{array} \]

Wanted: A procedure to tell well-typed, such as \((w : \text{Bool})\)

not \(w\)

from not well-typed, such as,

\(\text{size}(w)\).

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

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

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

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

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

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

(iii) Well-typedness in type environment \( A \) and context \( D \): (for visibility)
\[ A, D \vdash \text{expr} : \tau \]
\[ \text{self} : \tau_C, C \vdash \text{self}.r.v : \text{Int} \]
Constants and Operations

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

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

- 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`:

  \[ \frac{\vdash expr_1 : \tau_1 \quad \vdash \cdots \quad \vdash expr_n : \tau_n}{\vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{E}) \]

(Note: this rule also covers ‘\(=\)’, ‘isEmpty’, and ‘size’.)
**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

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

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

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

- If `expr` is of type `τ`, then it is of type `τ` in any type environment:

\[
\frac{}{A ⊢ expr : τ} \quad (EnvIntro)
\]

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

\[
\frac{A ⊢ expr_1 : τ_1 \ldots A ⊢ expr_n : τ_n}{A ⊢ \omega(expr_1, \ldots, expr_n) : τ, \quad n ≥ 1, \omega /∈ atr(∉)} \quad (Fun_{1})
\]

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

\[
\frac{w : τ ∈ A}{A ⊢ w : τ} \quad (Var)
\]

Type Environment Example

Example:

- `w + 3, A = w : Int`

\[\begin{align*}
A ⊢ \omega ∈ A & \quad (Var) \\
A ⊢ w : \text{Int} & \quad (EnvIntro) \\
A ⊢ w : \text{Int} & \quad (EnvIntro) \\
A ⊢ 3 : \text{Int} & \quad (EnvIntro) \\
A ⊢ 3 + (w, 3) : \text{Int} & \quad (Fun_{1})
\end{align*}\]
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) \[ \vdash \text{allInstances}_C : \text{Set}(\tau_C) \]

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

  (Attr^0.1) \[ A \vdash expr_1 : \tau_C \]
  \[ A \vdash r_1(expr_1) : \tau_D \]
  \[ r_1 : D_{0,1} \in \text{atr}(C) \]

  (Attr^\ast) \[ A \vdash expr_1 : \tau_C \]
  \[ A \vdash r_2(expr_1) : \text{Set}(\tau_D) \]
  \[ r_2 : D_\ast \in \text{atr}(C) \]

Attributes in Type Environment Example

- `self : \tau_C \vdash self.x : \text{Int}`

- `self : \tau_C \vdash self.x \times \tau \vdash self.y : \text{Int}`

- `self : \tau_D \vdash self.x : \text{Int}`
Iterate

- If `expr` is an **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:

  \[
  A \vdash \text{expr}_1 : \text{set}(\tau_1) \quad A^i \vdash \text{expr}_2 : \tau_1 \\
  A \vdash \text{expr}_1 \rightarrow \text{iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = \text{expr}_2 \mid \text{expr}_3) : \tau_2
  \]

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

- Context `C` in:

\[
\text{set}_A \rightarrow \text{iterate}_C(y : \text{set} \rightarrow \text{D}, \; x : \text{kat} = \text{set}_A(y \upharpoonright \text{D}) ; x : \text{kat})
\]

- Both evaluate `expr` in the active scope (A) instead of \( A' \) as \( \text{expr}_2 \) needs to be evaluated even with empty base set (as given by `expr_1`).
Iterate Example

\[
\begin{array}{c}
(\text{AllInst}) \quad \Gamma \vdash \text{allInstances}_C : \text{Set}(\tau_C) \\
(\text{Attr}) \quad A \vdash \text{expr}_1 : \tau_C \\
\hline
(\text{Iter}) \quad A \vdash \text{expr}_1 : \text{Set}(\tau_1) \quad A' \vdash \text{expr}_2 : \tau_2 \\
\hline
A \vdash \text{expr}_1 \rightarrow \text{iterate}(w_1 : \tau_1; w_2 : \tau_2 = \text{expr}_2 | \text{expr}_3) : \tau_2
\end{array}
\]

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

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

\[
\begin{align*}
\text{allInstances}_C & \rightarrow \text{iterate}(self : C; w : \text{Bool} = \text{true} | w \land self . x = 0) \\
\text{allInstances}_C & \rightarrow \forall (self : C | self . x = 0) \\
\text{context self : } C \text{ inv : self . } x & = 0 \\
\text{context } C \text{ inv : } x & = 0
\end{align*}
\]

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...?

\[
X = (\{\text{Int}\}, \{C, D\}, \{x : \text{Int}, n : D_0, 1\}, \{C \mapsto \{n\}, \{D \mapsto \{x\}\})
\]

- Plain syntax error:

  context \( C : \text{false} \)

- Subtle syntax error:

  context \( C \text{ inv : } y = 0 \)

- Type error:

  context \( self : C \text{ inv : self . } n = self . n . x \)
One Possible Extension: Implicit Casts

• We may wish to have

$$\vdash \text{true} \text{ and } \text{false} : \text{Bool}$$

(*)

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:

$$(\text{Cast}) \quad A \vdash \text{expr} : \text{Int} \quad \frac{}{A \vdash \text{expr} : \text{Bool}}$$

• 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...
Implicit Casts Cont’d

So, why isn’t there an interpretation for \((1 \text{ and } \text{false})\)?

- First of all, we have (syntax)

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

- Thus,

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

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

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

\[
x \mapsto \begin{cases} 
  \text{true}, & \text{if } x \in \text{true} \cup I(\text{Int}) \setminus \{0, \bot_{\text{Int}}\} \\
  \text{false}, & \text{if } x \in I(\text{false}, 0) \\
  \bot_{\text{Bool}}, & \text{otherwise}
\end{cases}
\]
**Bottomline**

- 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:

\[ \mathcal{S} = \{ \{ \text{Int} \}, \{ C, D \}, \{ n : D_{0.1}, m : D_{0.1}, (x : \text{Int}, \xi, \text{expr}_0, 0) \}, \{ C \mapsto \{ n \}, D \mapsto \{ x, m \} \} \]

\[
\begin{array}{c}
C \quad \xrightarrow{\xi} \quad D \\
\xi x : \text{Int} = \text{expr}_0 \\
\end{array}
\]

and

\[
\begin{array}{c}
C \quad \xrightarrow{n} \quad D \\
\xi x : \text{Int} = \text{expr}_0 \\
\end{array}
\]

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 . n . x = 0 )</td>
<td>✔</td>
<td>✔</td>
<td>later</td>
<td>not</td>
</tr>
<tr>
<td>( w_2 . m . x = 0 )</td>
<td>✔</td>
<td>✔</td>
<td>later</td>
<td>not</td>
</tr>
</tbody>
</table>

Context

- **Example:** A problem?

\[
\begin{array}{c}
C \quad \xrightarrow{r} \quad D \\
\tau D \vdash self : \tau D \\
\end{array}
\]

- 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 \( D \in \mathcal{C} \):

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

- 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:

$$\frac{A \vdash \text{expr} : \tau}{A, D \vdash \text{expr} : \tau} \quad \text{(\textit{ContextIntro})}$$

- Accessing an attribute $v$ of an object of class $C$ is well-typed
  - if $v$ is public, or
  - if the expression $\text{expr}_1$ denotes an object of class $C$:

$$\frac{A, D \vdash \text{expr}_1 : \tau}{A, D \vdash v(\text{expr}_1) : \tau}, \quad \langle v : \tau, \xi, \text{expr}_0, P_0 \rangle \in \text{atr}(C), \quad \xi = +, \text{ or } \xi = - \text{ and } C = D \quad \text{(\textit{Attr}_1)}$$

- Accessing $C_{0,1}$- or $C_*$-typed attributes: similar.

\[
\begin{align*}
\text{Example:} & \\
A, C \vdash r(\text{self}) : D \\
A, C \vdash v(r(\text{self})) : \text{Int} \\
A, C \vdash \text{self} \cdot r \cdot v > 0
\end{align*}
\]
**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 $D$)

  \[
  \text{context } C \text{ inv : } n x > 0 \?
  \]

\[get\times(c)\]
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 $\mathcal{C} \mathcal{D}$

\{ induces \\
extended (\!) signature $\mathcal{P}(\mathcal{C} \mathcal{D})$ \\
gives rise to \\
Basic Type System

- We extended the type system for
  
  \begin{itemize}
  \item \textbf{casts} (requires change of $I$) and
  \item \textbf{visibility} (no change of $I$).
  \item \textbf{Later: navigability} of associations.
  \end{itemize}

\textbf{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

