Software Design, Modelling and Analysis in UML

Lecture 07: A Type System for Visibility

2013-11-18

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Contents & Goals

Last Lecture:
- Representing class diagrams as (extended) signatures — for the moment without associations (see Lecture 08).
- 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: From Class Boxes to Extended Signatures
Extended Classes

From now on, we assume that each class $C \in \mathcal{C}$ has:

- a finite (possibly empty) set $S_C$ of stereotypes,
- a boolean flag $a \in \mathbb{B}$ indicating whether $C$ is abstract,
- a boolean flag $t \in \mathbb{B}$ indicating whether $C$ is active.

We use $\mathcal{S}$ to denote the set $\bigcup_{C \in \mathcal{C}} S_C$ of stereotypes in $\mathcal{I}$.

(Alternatively, we could add a set $St$ as 5-th component to $\mathcal{I}$ to provides the stereotypes (names of stereotypes) to choose from. But: too unimportant to care.)

**Convention:**

- We write

$$\langle C, S_C, a, t \rangle \in \mathcal{C}$$

when we want to refer to all aspects of $C$.

- If the new aspects are irrelevant (for a given context), we simply write $C \in \mathcal{C}$ i.e. old definitions are still valid.
**Extended Attributes**

- From now on, we assume that each attribute $v \in V$ has (in addition to the type):
  - a **visibility**
    $$\xi \in \{\text{public, private, protected, package}\}$$
    
    $$ ::= + \quad ::= - \quad ::= # \quad ::= \sim$$

- an **initial value** $expr_0$ given as a word from language for initial values, e.g. OCL expresions.
  (If using Java as action language (later) Java expressions would be fine.)

- a finite (possibly empty) set of **properties** $P_v$.
  We define $P_v$ analogously to stereotypes.

**Convention:**
- We write $\langle \tau, \xi, expr_0, P_v \rangle \in V$ when we want to refer to all aspects of $v$.
- Write only $v : \tau$ or $v$ if details are irrelevant.
A class box $n$ induces an (extended) signature class as follows:

$$n: \langle \langle S_1, \ldots, S_k \rangle \rangle$$

$$C \xi_1 v_1; \tau_1 = v_{0,1} \{P_{1,1}, \ldots, P_{1,m_1}\}$$

$$\vdots$$

$$C \xi_\ell v_\ell; \tau_\ell = v_{0,\ell} \{P_{\ell,1}, \ldots, P_{\ell,m_\ell}\}$$

$$C(n) := \langle C, \{S_1, \ldots, S_k\}, a(n), t(n) \rangle$$

$$V(n) := \{\langle v_1 : \tau_1, \xi_1, v_{0,1}, \{P_{1,1}, \ldots, P_{1,m_1}\} \rangle, \ldots, \langle v_\ell : \tau_\ell, \xi_\ell, v_{0,\ell}, \{P_{\ell,1}, \ldots, P_{\ell,m_\ell}\} \rangle\}$$

$$\text{attr}(n) := \{C \mapsto \{v_1, \ldots, v_\ell\}\}$$

where

- "abstract" is determined by the font:
  $$a(n) = \begin{cases} 
  \text{true} & \text{if } n = \boxed{C} \text{ or } n = \boxed{C \{A\}} \\
  \text{false} & \text{otherwise}
  \end{cases}$$

- "active" is determined by the frame:
  $$t(n) = \begin{cases} 
  \text{true} & \text{if } n = \boxed{C} \text{ or } n = \boxed{C} \\
  \text{false} & \text{otherwise}
  \end{cases}$$
Excursus: Type Theory (cf. Thiemann, 2008)
Recall: In lecture 03, we introduced OCL expressions with **types**, for instance:

\[
expr ::= \quad w : \tau \quad \ldots \text{logical variable } w \\
| \text{true} | \text{false} : \text{Bool} \quad \ldots \text{constants} \\
| 0 \mid -1 \mid 1 \mid \ldots : \text{Int} \quad \ldots \text{constants} \\
| expr_1 + expr_2 : \text{Int} \times \text{Int} \to \text{Int} \quad \ldots \text{operation} \\
| \text{size}(expr_1) : \text{Set}(\tau) \to \text{Int} \\
| \text{not } expr : \text{Bool} \to \text{Bool}
\]

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

\[\text{not } w\]

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. \(\tau \in T_B \cup T_C \cup \{\text{Set}(\tau_0) \mid \tau_0 \in T_B \cup T_C\}\), \(C \in \mathcal{C}\).
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

```
("name")  "premises"  "conclusion"  "side condition"
```

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

(i) Universal well-typedness:

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

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

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

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

\[ A, B \vdash expr : \tau \]
\[ self : \tau_C, C \vdash self.r.v : \text{Int} \]
### Constants and Operations

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

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

- 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 $\tau$:
  \[(Fun_0) \quad \vdash expr_1 : \tau_1 \ldots \vdash expr_n : \tau_n \quad \vdash \omega(expr_1, \ldots, expr_n) : \tau, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin atr(\mathcal{C})\]

(Note: this rule also covers ‘$\equiv_{\tau}$’, ‘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\_0) \[ \vdash expr_1 : \tau_1 \ldots \vdash expr_n : \tau_n, \quad \omega : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{C}) \]

Example:

- **not true**

  \[
  \frac{(BOOL) \quad \vdash \text{true} : \text{Bool}}{(Fun_0) \quad \vdash \text{not} \, \text{true} : \text{Bool}} \]

  \[
  \frac{(Fun_0) \quad \vdash \text{true} : \text{Int} \quad \vdash 3 : \text{Int}}{(NT) \quad \vdash \text{true} + 3 : \text{Int}} \]

- **true + 3**

  ① get stuck - we cannot derive this from the rules

  ② true + 3 is not well-typed
• **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 \( T = T_B \cup T_C \cup \{ \text{Set}(\tau_0) \mid \tau_0 \in T_B \cup T_C \} \).
Environment Introduction and Logical Variables

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

\[
\frac{}{A \vdash \text{(EnvIntro)} \vdash expr : \tau \quad A} \ \vdash \ \frac{}{A \vdash expr : \tau}
\]

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

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

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

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

\[
\begin{align*}
(EnvIntro) & \quad \frac{}{A \vdash \text{expr} : \tau} \\
(Fun_1) & \quad \frac{A \vdash \text{expr}_1 : \tau_1 \ldots \ A \vdash \text{expr}_n : \tau_n}{A \vdash \omega(\text{expr}_1,\ldots,\text{expr}_n) : \tau}, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \quad n \geq 1, \omega \notin \text{atr}(\mathcal{E}) \\
(Var) & \quad \frac{\text{w} : \tau \in A}{A \vdash \text{w} : \tau}
\end{align*}
\]

Example:

- \( w + 3, \ A = w : \text{Int} \)

\[
\begin{align*}
\text{w : Int} & \in A \\
A & \vdash \text{w : Int} \\
\vdash 3 : \text{Int} \\
A & \vdash 3 : \text{Int} \\
\vdash \text{w} + 3 : \text{Int} \\
A & \vdash \text{w} + 3 : \text{Int} \\
\vdash \text{w} : \text{Int} \\
\vdash (\text{w}, \text{z}) : \text{Int} \times \text{Int} \\
\vdash \text{w} + 3 : \text{Int} \\
\vdash \text{w} + 3 : \text{Int} \quad \text{prefix normal form}
\end{align*}
\]

\( \emptyset \) w+3 is well-typed under A
All Instances and Attributes in Type Environment

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

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

- 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 \( \tau_C \):

\[ (\text{Attr}_0) \quad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}, \quad v : \tau \in atr(C), \tau \in \mathcal{T} \]

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

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

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

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

\[(Attr_0^{*})\]
\[
\frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : \text{Set}(\tau_D)}, \quad r_2 : D^{*} \in atr(C)
\]

1. \textit{self} : \tau_C \vdash \textit{self} . y : \text{Int}

2. \textit{self} : \tau_C \vdash \textit{self} . x : \text{Int} \quad \text{well-typed by (Attr_0), (Var)}

3. \textit{self} : \tau_C \vdash \textit{self} . r : \tau_D \quad \text{well-typed (Attr_0^{0,1}), (Var)}

4. \textit{self} : \tau_C \vdash \textit{self} . r . x : \text{Int} \quad \text{not well-typed, yet stuck after applying (Attr_0^{0,1})}

5. \textit{self} : \tau_C \vdash \textit{self} . f . y : \text{Int} \quad \text{well-typed by (Attr_0^{0,1}), (Attr_0), (Var)}
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:

\[
\frac{A \vdash expr_1 : \text{Set}(\tau_1) \quad A' \vdash expr_2 : \tau_2 \quad A' \vdash expr_3 : \tau_2}{A \vdash expr_1 \rightarrow \text{iterate}(w_1 : \tau_1 \ ; \ w_2 : \tau_2 = expr_2 \ | \ expr_3) : \tau_2}
\]

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

override typing of $w_1$ and $w_2$ in $A$ ("$w_1 : \tau_1, w_2 : \tau_2$ hide outer scope")

\[\text{outer scope}\]
\[\text{inner scope}\]
Iterate Example

\[
\begin{array}{c}
(AllInst) \quad \frac{}{\vdash \text{allInstances}_C : \text{Set}(\tau_C)} \\
\quad \quad (Attr) \quad \frac{A \vdash \text{expr}_1 : \tau_C}{A \vdash v(\text{expr}_1) : \tau}
\end{array}
\]

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

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

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

\[
\begin{array}{c}
\frac{A \vdash \text{allInstances}_C : \text{Set}(\tau_C)}{A \vdash \text{iterate}(\text{self} : C \; ; \; w : \text{Bool} = \text{true} \; | \; (\text{self}(x).0 : \text{Bool})) : \text{Set}(\tau_C)}
\end{array}
\]

\[
\begin{array}{c}
\frac{A' \vdash \text{self} : \tau_C}{A' \vdash \text{self}(x): \text{Int} \quad A' \vdash 0 : \text{Int}}
\end{array}
\]

\[
\begin{array}{c}
A \vdash \text{true} : \text{Bool}
\end{array}
\]

\[
\begin{array}{c}
\frac{\text{self} : \tau_C \in A'}{A' \vdash \text{true} : \text{Bool}}
\end{array}
\]

\[
\begin{array}{c}
A \vdash \text{allInstances}_C \rightarrow \text{iterate}(\text{self} : C \; ; \; w : \text{Bool} = \text{true} \; | \; \text{self}(x).0 : \text{Bool}) \quad \text{context} \; \text{self} : C \; \text{inv} \; \text{self} : x = 0
\end{array}
\]

\[
\frac{A \vdash \text{context} \; C \; \text{inv} : x = 0 \quad \text{and}(v, \text{inv}) \quad A' \vdash \text{self}(x) : \text{Int} \quad A' \vdash 0 : \text{Int}}{A \vdash (\text{self}(x) . 0) : \text{Bool} \quad A' \vdash \text{self}(x) . 0 : \text{Bool}}
\]

\[
\frac{}{A \vdash \text{allInstances}_C \rightarrow \text{iterate}(\text{self} : C \; ; \; w : \text{Bool} = \text{true} \; | \; \text{self}(x).0 : \text{Bool}) \quad \text{context} \; \text{self} : C \; \text{inv} : x = 0 \quad \text{and}(v, \text{inv}) \quad A' \vdash \text{self}(x) : \text{Int} \quad A' \vdash 0 : \text{Int}}{A \vdash \text{allInstances}_C \rightarrow \text{iterate}(\text{self} : C \; ; \; w : \text{Bool} = \text{true} \; | \; \text{self}(x).0 : \text{Bool}) \quad \text{context} \; \text{self} : C \; \text{inv} : x = 0 \quad \text{and}(v, \text{inv}) \quad A' \vdash \text{self}(x) : \text{Int} \quad A' \vdash 0 : \text{Int} \quad \text{well-typed}}
\]
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 \(C\) : false

- **Subtle syntax error (depends on signature)**:

  context \(C\) inv : \(y = 0\)  

- **Type error**:

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

- We **may wish** to have

\[ \vdash 1 \text{ and } \text{false} : \text{Bool} \quad (\ast) \]

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

- Then just have a rule:

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

- With (Cast) (and (Int), and (Bool), and (Fun_0)), we can derive the sentence \((\ast)\), 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 \((\ast)\) means...
Implicit Casts Cont’d

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

• First of all, we have (syntax)

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

• Thus,

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

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

• 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

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

where

• \(b_1 := toBool(I[expr_1](\sigma, \beta))\),

• \(b_2 := toBool(I[expr_2](\sigma, \beta))\),

and where

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

\[
x \mapsto \begin{cases} 
  \text{true}, & \text{if } x \in \{\text{true}\} \cup I(\text{Int}) \setminus \{0, \bot_{\text{Int}}\} \\
  \text{false}, & \text{if } x \in \{\text{false}, 0\} \\
  \bot_{\text{Bool}}, & \text{otherwise}
\end{cases}
\]
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 \( \text{Bool} \) and \( \text{Int} \) (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>✓️ later</td>
<td>not</td>
</tr>
<tr>
<td>( w_2 \cdot m \cdot x = 0 )</td>
<td>✔️</td>
<td>✗</td>
<td>✗ later</td>
<td>not</td>
</tr>
<tr>
<td>( x(m(w_2)) = 0 )</td>
<td>✗</td>
<td>✗</td>
<td>✗ later</td>
<td>not</td>
</tr>
</tbody>
</table>
**Context**

- **Example:** A problem?

\[ y = ( \{ w, z_c, d_3, \varphi : D_0, v : h m f, \beta \subseteq \varphi_3, \Delta \subseteq \varphi_3 \} ) \]

- 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 \( B \in C \):

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

- In particular: prepare 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:
  $$\frac{A \vdash expr : \tau}{A, B \vdash expr : \tau}$$

- Accessing attribute $v$ of a $C$-object via logical variable $w$ is well-typed if
  - $v$ is public, or
  - $w$ is of type $\tau_B$
  $$\frac{A \vdash w : \tau_B}{A, B \vdash v(w) : \tau} \quad \langle v : \tau, \xi, expr_0, P_\varphi \rangle \in atr(B)$$

- Accessing attribute $v$ of a $C$-object of via expression $expr_1$ is well-typed in context $B$ if
  - $v$ is public, or $expr_1$ denotes an object of class $B$:
  $$\frac{A, B \vdash expr_1 : \tau_C}{A, B \vdash v(expr_1) : \tau} \quad \langle v : \tau, \xi, expr_0, P_\varphi \rangle \in atr(C), \xi = +, \text{ or } C = B$$

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

- Operator Application:

\[
(Fun_2) \quad \frac{A, B \vdash expr_1 : \tau_1 \ldots A, B \vdash expr_n : \tau_n}{A, B \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \quad n \geq 1, \omega \notin atr(\mathcal{C})
\]

- Iterate:

\[
(Iter_1) \quad \frac{A, B \vdash expr_1 : Set(\tau_1) \quad A', B \vdash expr_2 : \tau_2 \quad A', B \vdash expr_3 : \tau_2}{A, B \vdash expr_1 \rightarrow \text{iterate}(w_1 : \tau_1 ; w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}
\]

where \( A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2) \).
Attribute Access in Context Example

\[(\text{ContextIntro})\] \[\frac{A \vdash expr : \tau}{A \otimes \vdash expr : \tau}\]

\[(\text{Attr}_1)\] \[\frac{A, B \vdash expr_1 : \tau_C}{A, B \vdash v(expr_1) : \tau}, \quad \langle v : \tau, \xi, expr_0, P_\varnothing \rangle \in atr(C), \; \xi = +, \text{ or } \xi = - \text{ and } C = B\]

Example:

\[
\text{self} : \tau_C \quad \vdash \text{self} \cdot r \cdot v > 0
\]
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 \( D \))
  
  \[
  \text{context } C \text{ inv } : n.x > 0 ?
  \]
  
  - 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 $\mathcal{CD}$
\[
\begin{array}{c}
\text{induces} \\
\text{extended (!) signature } \mathcal{I}(\mathcal{CD}) \\
\text{gives rise to}
\end{array}
\]

Basic Type System

- We extended the type system for
  - casts (requires change of $I$) and see earlier slides
  - 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

