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 $C \in \mathscr{C}$ 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 $S_\mathscr{C}$ to denote the set $\bigcup_{C \in \mathscr{C}} S_C$ of stereotypes in $\mathscr{S}$.

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

Convention:

- We write

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

  when we want to refer to all aspects of C.

- If the new aspects are irrelevant (for a given context), we simply write $C \in \mathscr{C}$ 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**

$$\xi \in \{\underbrace{\text{public}}_{:=+}, \underbrace{\text{private}}_{:=-}, \underbrace{\text{protected}}_{:=\#}, \underbrace{\text{package}}_{:=\sim}\}$$

  - 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 $P_{\mathscr{C}}$ analogously to stereotypes.

    V

Convention:

- We write $\langle v : \tau, \xi, expr_0, P_v \rangle \in V$ when we want to refer to all aspects of v.
- Write only $v : \tau$ or v if details are irrelevant.

From Class Boxes to Extended Signatures

A class box n **induces** an (extended) signature class as follows:

n:

$\langle\langle\, S_1, \dots, S_k \,\rangle\rangle$ C
$\xi_1\ v_1 : \tau_1 = v_{0,1}\ \{P_{1,1}, \dots, P_{1,m_1}\}$ $\vdots$ $\xi_\ell\ v_\ell : \tau_\ell = v_{0,\ell}\ \{P_{\ell,1}, \dots, P_{\ell,m_\ell}\}$

$$\rightsquigarrow$$

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

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

$$atr(n) := \{C \mapsto \{v_1, \dots, v_\ell\}\}$$

where

- "abstract" is determined by the font:

$$a(n) = \begin{cases} \textit{true} & \text{, if } n = \boxed{C} \text{ or } n = \boxed{\mathit{C}\ {\scriptstyle \{A\}}} \\ \textit{false} & \text{, otherwise} \end{cases}$$

- "active" is determined by the frame:

$$t(n) = \begin{cases} \textit{true} & \text{, if } n = \boxed{C} \text{ or } n = \boxed{\,\|\ C\ \|\,} \\ \textit{false} & \text{, otherwise} \end{cases}$$

Excursus: Type Theory (cf. Thiemann, 2008)

Type Theory

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

$$
\begin{array}{rlll}
expr ::= & w & : \tau & \text{... logical variable } w \\
& \mid \mathsf{true} \mid \mathsf{false} & : Bool & \text{... constants} \\
& \mid 0 \mid -1 \mid 1 \mid \dots & : Int & \text{... constants} \\
& \mid expr_1 + expr_2 & : Int \times Int \to Int & \text{... operation} \\
& \mid \mathsf{size}(expr_1) & : Set(\tau) \to Int & \\
& \mid \text{not } expr & : Bool \to Bool &
\end{array}
$$

Wanted: A procedure to tell **well-typed**, such as $(w : Bool)$

$$\text{not } w$$

from **not well-typed**, such as,

$$\mathsf{size}(w).$$

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

$$A, C \vdash expr : \tau \qquad\qquad (\textbf{read}: \text{“expression } expr \text{ has type } \tau\text{”})$$

for some OCL type τ, i.e. $\tau \in T_B \cup T_{\mathscr{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathscr{C}}\}$, $C \in \mathscr{C}$.

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

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

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

(i) Universal well-typedness:

$$\vdash \mathit{expr} : \tau$$

$$\vdash 1 + 2 : \mathit{Int}$$

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

$$A \vdash \mathit{expr} : \tau$$

$$\mathit{self} : \tau_C \vdash \mathit{self}.v : \mathit{Int}$$

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

$$A, B \vdash \mathit{expr} : \tau$$

$$\mathit{self} : \tau_C, C \vdash \mathit{self}\ .\ r\ .\ v : \mathit{Int}$$

Constants and Operations

- If $expr$ is a **boolean constant**, then $expr$ is of type $Bool$:

$$(BOOL) \qquad \frac{}{\vdash B : Bool}, \qquad B \in \{\mathit{true}, \mathit{false}\}$$

- If $expr$ is an **integer constant**, then $expr$ is of type Int:

$$(INT) \qquad \frac{}{\vdash N : Int}, \qquad N \in \{0, 1, -1, \dots\}$$

- If $expr$ is the application of **operation** $\omega : \tau_1 \times \dots \times \tau_n \to \tau$ to expressions $expr_1, \dots, expr_n$ which are of type $\tau_1, \dots, \tau_n$, then $expr$ is of type τ:

$$(Fun_0) \qquad \frac{\vdash expr_1 : \tau_1 \quad \dots \quad \vdash expr_n : \tau_n}{\vdash \omega(expr_1, \dots, expr_n) : \tau}, \qquad \begin{array}{l} \omega : \tau_1 \times \dots \times \tau_n \to \tau, \\ n \geq 1,\ \omega \notin atr(\mathscr{C}) \end{array}$$

(Note: this rule also covers '$=_\tau$', 'isEmpty', and 'size'.)

Constants and Operations Example

$$(BOOL)\quad \frac{}{\vdash B : \mathit{Bool}}, \qquad B \in \{\mathit{true}, \mathit{false}\}$$

$$(INT)\quad \frac{}{\vdash N : \mathit{Int}}, \qquad N \in \{0, 1, -1, \dots\}$$

$$(Fun_0)\quad \frac{\vdash \mathit{expr}_1 : \tau_1 \quad \dots \quad \vdash \mathit{expr}_n : \tau_n}{\vdash \omega(\mathit{expr}_1, \dots, \mathit{expr}_n) : \tau}, \qquad \omega : \tau_1 \times \dots \times \tau_n \to \tau,\; n \geq 1,\; \omega \notin atr(\mathscr{C})$$

Example:

- not *true*

$$(Fun_0)\ \frac{(BOOL)\ \dfrac{}{\vdash true : Bool}}{\vdash not(true) : Bool}\ \ not : Bool \to Bool$$

- $\mathit{true} + 3$

got stuck – we cannot ① derive this from the rules

$$(Fun_0)\ \frac{\vdash true : Int \qquad \dfrac{}{\vdash 3 : Int}\,(INT)}{\vdash true + 3 : Int}\ \ + : Int \times Int \to Int$$

② $\Rightarrow$ 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 $T = T_B \cup T_{\mathscr{C}} \cup \{Set(\tau_0) \mid \tau_0 \in T_B \cup T_{\mathscr{C}}\}$.

Environment Introduction and Logical Variables

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

$$(EnvIntro) \quad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

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

$$(Fun_1) \quad \frac{A \vdash expr_1 : \tau_1 \quad \ldots \quad A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \ldots, expr_n) : \tau}, \quad \begin{array}{l} \omega : \tau_1 \times \cdots \times \tau_n \to \tau, \\ n \geq 1,\ \omega \notin atr(\mathscr{C}) \end{array}$$

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

$$(Var) \quad \frac{w : \tau \in A}{A \vdash w : \tau}$$

Type Environment Example

$$(EnvIntro) \qquad \frac{\vdash expr : \tau}{A \vdash expr : \tau}$$

$$(Fun_1) \qquad \frac{A \vdash expr_1 : \tau_1 \quad \dots \quad A \vdash expr_n : \tau_n}{A \vdash \omega(expr_1, \dots, expr_n) : \tau}, \qquad \begin{array}{l}\omega : \tau_1 \times \dots \times \tau_n \to \tau, \\ n \geq 1,\ \omega \notin atr(\mathscr{C})\end{array}$$

$$(Var) \qquad \frac{w : \tau \in A}{A \vdash w : \tau}$$

Example:

- $w + 3,\ A = w : Int$

$$\dfrac{\dfrac{w : Int \in A}{A \vdash w : Int}\,(Var) \qquad \dfrac{\dfrac{}{\vdash 3 : Int}\,(INT)}{A \vdash 3 : Int}\,(EnvIntro)}{\underbrace{w : Int}_{=:A} \vdash w+3 : Int}\,(Fun_1) \qquad + : Int \times Int \to Int$$

$\underset{\omega}{+}(\underset{expr_1}{w},\underset{expr_2}{3})$ ← prefix normal form

⇒ $w+3$ 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 $Set(\tau_C)$,

$$(AllInst) \qquad \frac{}{\vdash \mathsf{allInstances}_C : Set(\tau_C)}$$

- If $expr$ is an **attribute access** of an attribute of type τ for an object of C as denoted by $expr_1$, then the premise is that $expr_1$ is of type τ_C:

$$(Attr_0) \qquad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}, \quad v : \tau \in atr(C),\ \tau \in \mathscr{T}$$

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

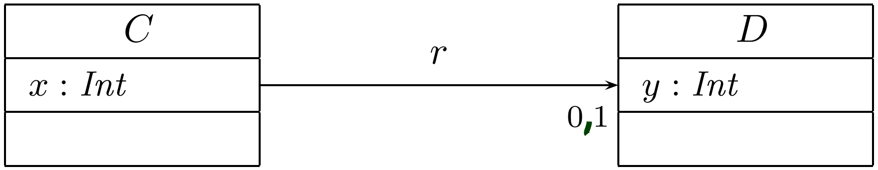
$$(Attr_0^{*}) \qquad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \quad r_2 : D_* \in atr(C)$$

Attributes in Type Environment Example

$$(Attr_0)\qquad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}, \qquad v : \tau \in atr(C),\ \tau \in \mathscr{T}$$

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

$$(Attr_0^{*})\qquad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \qquad r_2 : D_{*} \in atr(C)$$



derivable but not useful

$V = \{x : Int, r : D_{0,1}, y : Int\}$

$atr(C) = \{x, r\}$

$$\frac{self : \tau_C \vdash self : \tau_C \ (\tau_D)}{self : \tau_C \vdash self.y : Int}$$

↳ get stuck
↳ not well-typed

needed but not derivable

- $self : \tau_C \vdash self.y : Int$
- $self : \tau_C \vdash self.x : Int$ well-typed by $(Attr_0)$, (Var)
- $self : \tau_C \vdash self.r : \tau_D$ well-typed $(Attr_0^{0,1})$, (Var)
- $self : \tau_C \vdash self.r.x : Int$ not well-typed, get stuck after applying $(Attr_0^{0,1})$
- $self : \tau_C \vdash self.r.y : Int$ 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:

well-typedness of expr 2 depends on outer scope

… inner scope

$$(Iter)\quad \frac{A \vdash expr_1 : Set(\tau_1) \qquad A \vdash expr_2 : \tau_2 \qquad A' \vdash expr_3 : \tau_2}{A \vdash expr_1\texttt{->iterate}(w_1 : \tau_1\ ;\ w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}$$

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

override typing of w_1 and w_2 in A

("$w_1 : \tau_1$, $w_2 : \tau_2$ hide outer scope")

outer scope

allInst$_C$ → iterate(i … |

i.r → iterate(j … | …))

inner scope

Iterate Example

$$(AllInst)\quad \frac{}{\vdash \mathsf{allInstances}_C : Set(\tau_C)} \qquad (Attr)\quad \frac{A \vdash expr_1 : \tau_C}{A \vdash v(expr_1) : \tau}$$

$$(Iter)\quad \frac{A \vdash expr_1 : Set(\tau_1) \quad A' \vdash expr_2 : \tau_2 \quad A' \vdash expr_3 : \tau_2}{A \vdash expr_1\texttt{->}\mathsf{iterate}(w_1 : \tau_1 \,;\, w_2 : \tau_2 = expr_2 \mid expr_3) : \tau_2}$$

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

Example: $(\mathscr{S} = (\{Int\}, \{C\}, \{x : Int\}, \{C \mapsto \{x\}\}))$

allInstances$_C$ -> iterate($self : C; w : Bool = true \mid w \wedge self.x = 0$)

allInstances$_C$ -> forAll($self : C \mid self.x = 0$)

context $self : C$ inv : $self.x = 0$

$$\frac{self : \tau_C \in A'}{A' \vdash self : \tau_C} \qquad \frac{}{\vdash 0 : Int}$$

$$r : Bool \in A' \qquad \frac{A' \vdash self : \tau_C}{A' \vdash self(x) : Int} \qquad \frac{\vdash 0 : Int}{A' \vdash 0 : Int}$$

$$\frac{r : Bool \in A'}{A' \vdash r : Bool} \qquad \frac{A' \vdash self(x) : Int \quad A' \vdash 0 : Int}{A' \vdash =(self(x), 0) : Bool}$$

$$(AllInst)\ \frac{}{A \vdash allInstances_C : Set(\tau_C)} \qquad (Bool)\ \frac{}{true : Bool} \qquad (Env\ Intro)\ \frac{true : Bool}{A \vdash true : Bool} \qquad \frac{A' \vdash r : Bool \quad A' \vdash =(self(x), 0) : Bool}{\underbrace{r : Bool, self : \tau_C}_{=: A'} \vdash and(r, =(self(x), 0))}$$

$$\frac{A \vdash allInstances_C : Set(\tau_C) \quad A \vdash true : Bool \quad A' \vdash and(r, =(self(x), 0))}{A \vdash allInstances_C \rightarrow iterate(self : C; r : Bool = true \mid \underbrace{=(self(x), 0)}_{and(r,\ \ )})}$$

$A \vdash$ context C inv : $x = 0$

↳ 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...?

$$\mathscr{S} = (\{Int\}, \{C, D\}, \{x : Int, n : D_{0,1}\}, \{C \mapsto \{n\}, \{D \mapsto \{x\}\})$$

- Plain syntax error:

"inv" missing

context C : *false*

- Subtle syntax error (depends on signature)

not in $\mathscr{S}$

context C inv : $y = 0$

- Type error:

$:D_{0,1}$ $:Int$

context $self : C$ inv : $self\,.\,n = self\,.\,n\,.\,x$

Casting in the Type System

One Possible Extension: Implicit Casts

- We **may wish** to have

$$\vdash 1 \text{ and } \mathit{false} : \mathit{Bool} \qquad (*)$$

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

- Then just have a rule:

$$(\mathit{Cast}) \qquad \frac{A \vdash \mathit{expr} : \mathit{Int}}{A \vdash \mathit{expr} : \mathit{Bool}}$$

- 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 $(1$ and *false*$)$?

- First of all, we have (syntax)

$$expr_1 \text{ and } expr_2 : Bool \times Bool \to Bool$$

- Thus,

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

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

- By definition,

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

and **there we're stuck**.

Implicit Casts: Quickfix

- Explicitly define

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

  where

  - $b_1 := toBool(I[\![expr_1]\!](\sigma, \beta))$,
  - $b_2 := toBool(I[\![expr_2]\!](\sigma, \beta))$,

  and where

$$toBool : I(Int) \cup I(Bool) \to I(Bool)$$

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

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.

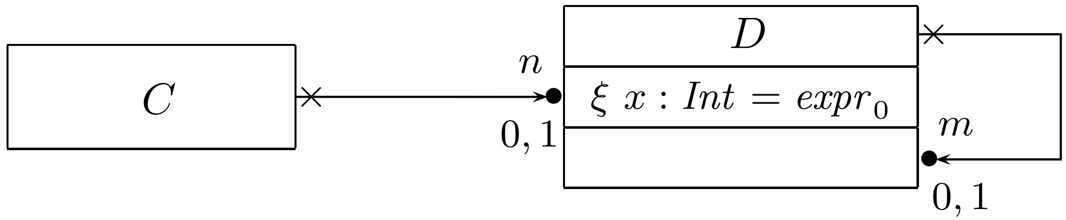
- **Note**: OCL has a far more elaborate type system which in particular addresses the relation between $Bool$ and Int (cf. [OMG, 2006]).

Visibility in the Type System

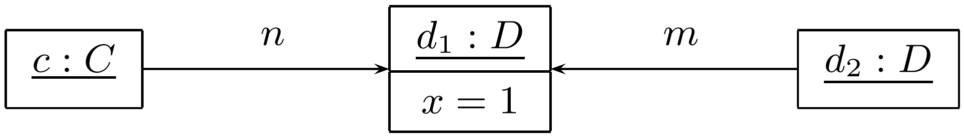
Visibility — The Intuition

$$\mathscr{S} = (\{Int\}, \{C, D\}, \{n : D_{0,1},\\ m : D_{0,1}, \langle x : Int, \xi, expr_0, \emptyset\rangle\},\\ \{C \mapsto \{n\}, D \mapsto \{x, m\}\}$$

Let's study an **Example**:



and



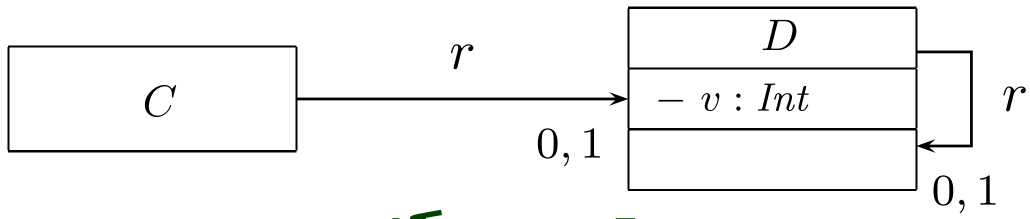
Assume $w_1 : \tau_C$ and $w_2 : \tau_D$ are logical variables. **Which** of the following **syntactically correct** (?) OCL expressions **shall** we consider to be **well-typed**?

ξ of x:	public	private	protected	package
$w_1 . n . x = 0$	✔	✔ –	later	not
	✘	✘ ~~IIII~~ II	privateness is by class,	
	?	? rest	not by object	
$w_2 . m . x = 0$	✔	✔ ~~IIII~~	later	not
$x(m(w_2)) = 0$	✘	✘ III		
	?	? rest		

Context

$\mathscr{S} = (\{Int\}, \{C, D\}, \{r : D_{0,1}, v : Int\}, \{C \mapsto \{r\}, D \mapsto \{r, v\}\})$

- **Example**: A problem?



$$self : \tau_D \vdash \overset{:\tau_D}{self} \; . \overset{:\tau_D}{r} \; . \, v > 0 \qquad \checkmark$$

$$self : \tau_C \nvdash \underset{:\tau_C}{self} \; . \, r \; . \, v > 0 \qquad \times$$

- 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** $B \in \mathscr{C}$:

$$A, B \vdash expr : \tau$$

- In particular: prepare to treat "protected" later (when doing inheritance).

Attribute Access in Context

- If $expr$ is of type τ in a type environment, then it is in **any context**:

$$(Context\cancel{Intro}) \qquad \frac{\overbrace{A}^{B,} \vdash expr : \tau}{A, \cancel{B} \vdash expr : \tau}$$

Drop

- **Accessing attribute** v of a C-object via logical variable w is well-typed if
  - ~~v is public, or~~ w is of type τ_B

$$(Attr_1) \qquad \frac{A \vdash w : \tau_B}{A, B \vdash v(w) : \tau}, \qquad \langle v : \tau, \xi, expr_0, P_{\mathscr{C}} \rangle \in atr(B)$$

- **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:

$$(Attr_2) \qquad \frac{A, B \vdash expr_1 : \tau_C}{A, B \vdash v(expr_1) : \tau}, \qquad \langle v : \tau, \xi, expr_0, P_{\mathscr{C}} \rangle \in atr(C), \; \xi = +, \text{ or } C = B$$

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

Context in Operator Application

- Operator Application:

$$(Fun_2) \quad \frac{A, B \vdash expr_1 : \tau_1 \;\; \dots \;\; A, B \vdash expr_n : \tau_n}{A, B \vdash \omega(expr_1, \dots, expr_n) : \tau}, \quad \begin{array}{l}\omega : \tau_1 \times \dots \times \tau_n \to \tau,\\ n \geq 1,\ \omega \notin atr(\mathscr{C})\end{array}$$

- Iterate:

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

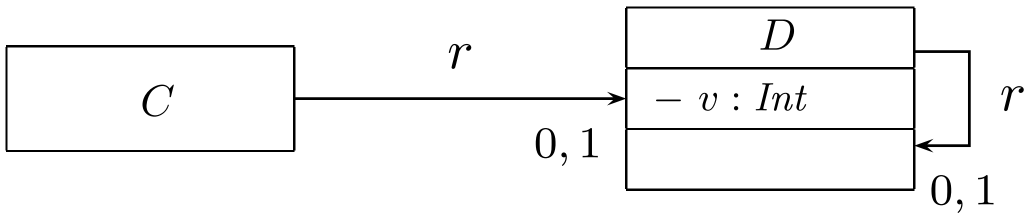
where $A' = A \oplus (w_1 : \tau_1) \oplus (w_2 : \tau_2)$.

– 07 – 2013-11-18 – Svisityp –

Attribute Access in Context Example

$(\mathit{Context}\cancel{\mathit{Intro}})$ Drop $\qquad \dfrac{A\,{}^{,B} \vdash \mathit{expr} : \tau}{A, \cancel{B} \vdash \mathit{expr} : \tau}$

$(\mathit{Attr}_1) \qquad \dfrac{A, B \vdash \mathit{expr}_1 : \tau_C}{A, B \vdash v(\mathit{expr}_1) : \tau}, \qquad \langle v : \tau, \xi, \mathit{expr}_0, P_{\mathscr{C}} \rangle \in \mathit{atr}(C),$
$\xi = +$, or $\xi = -$ and $C = B$



Example:

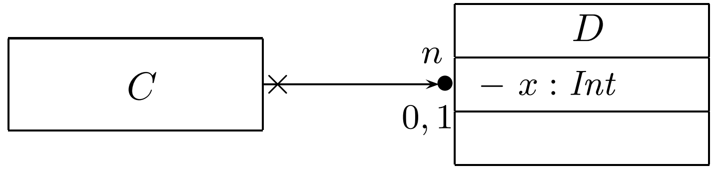
$$\mathit{self} : \tau_C \qquad \vdash \mathit{self} \,.\, r \,.\, v > 0$$

The Semantics of Visibility

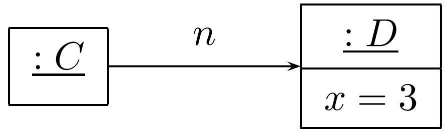
- **Observation**:
  - Whether an expression **does** or **does not** respect visibility is a matter of well-typedness **only**.
  - We only evaluate ($=$ apply I to) **well-typed** expressions.

$\rightarrow$ 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)

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

- **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

$$\begin{array}{c}\text{Class Diagrams } \mathscr{C}\mathscr{D} \\ \Big\downarrow \textbf{ induces} \\ \text{extended (!) signature } \mathscr{S}(\mathscr{C}\mathscr{D}) \\ \Big\downarrow \textbf{ gives rise to} \\ \text{Basic Type System}\end{array}$$

- 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

[OMG, 2006] OMG (2006). Object Constraint Language, version 2.0. Technical Report formal/06-05-01.

[OMG, 2007a] OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

[OMG, 2007b] OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.