Software Design, Modelling and Analysis in UML

Lecture 06: Type Systems and Visibility

2011-11-23

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 Lectures 07 and 08).
- **Insight**: **visibility** doesn't contribute to semantics in the sense that if $\mathscr{S}_1$ and $\mathscr{S}_2$ only differ in visibility of some attributes, then $\Sigma_{\mathscr{S}_1}^{\mathscr{D}} = \Sigma_{\mathscr{S}_2}^{\mathscr{D}}$ for each $\mathscr{D}$.
- **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	$: \tau$	. . . logical variable w
	$\mid$ true $\mid$ false	$: Bool$	. . . constants
	$\mid 0 \mid -1 \mid 1 \mid \ldots$	$: Int$	. . . constants
	$\mid expr_1 + expr_2$	$: Int \times Int \to Int$	. . . operation
	$\mid$ size$(expr_1)$	$: Set(\tau) \to Int$	

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

not w

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

size(w).

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

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

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:

$$\vdash expr : \tau$$
$$\vdash 1 + 2 : Int$$

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

$$A \vdash expr : \tau$$
$$self : \tau_C \vdash self.v : Int$$

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

$$A, D \vdash expr : \tau$$
$$self : \tau_C, C \vdash self . r . v : Int$$

Constants and Operations

name

premise

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

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

side - condition

conclusion

Constants and Operations

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

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

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

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

- 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:

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

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

Constants and Operations Example

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

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

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

$\omega : Bool \to Bool$, $expr_1$

Example:

- not (*true*)

$$\frac{\frac{}{\vdash true : Bool}\,(Bool)}{\vdash not(true) : Bool}\,(Fun_0)$$

thus ↳ not(true) is well-typed

- not("hello")

- *true* + 3

$+ : Int \times Int \to Int$

GOT STUCK, NO RULE TO SHOW $\vdash true : Int$

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

thus ↳ 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:

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

- 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

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

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

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

Example:

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

comes from

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

thus $w + 3$ well-typed in A

All Instances and Attributes in Type Environment

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

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

- 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:

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

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

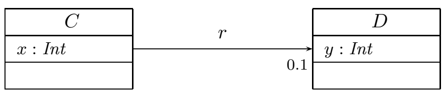
$$(Attr_0^{*}) \quad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \quad r_2 : D_* \in atr(C)$$

Attributes in Type Environment Example

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

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

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



- $self : \tau_C \vdash self.x$ ✓ : Int
- $self : \tau_C \vdash self.r.x$: Int ✗ syntax error, $x \notin atr(D)$
- $self : \tau_C \vdash self.r.y$: Int ✓
- $self : \tau_D \vdash self.x$: Int ✗ $x \notin atr(D)$

Iterate

if $expr_2$ may refer to w_2, w_1 from inner scope, then A' here

- 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:

$$(Iter)\quad \frac{A \vdash expr_1 : Set(\tau_1) \qquad \overset{(1)}{A'} \vdash expr_2 : \tau_2 \qquad A' \vdash expr_3 : \tau_2}{A \vdash \underbrace{expr_1}_{①}\text{->iterate}(w_1 : \tau_1\ ;\ w_2 : \tau_2 = \underbrace{expr_2}_{②} \mid \underbrace{expr_3}_{③}) : \tau_2}$$

iterator   result   init. value of result

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

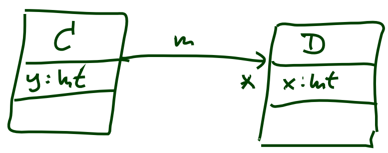
override typing of w_1, w_2 in A (w_1, w_2 hide outer scope)

$allInst_C \to iterate(i : \tau_C\ ,\ r : Bool = true \mid r \text{ and } i.x > 0)$

(equiv to $allInst_C \to$ ~~iterate~~ forAll $(i \mid i.x > 0)$)

- context C inv: self:

self.m -> iterate(self : D , r : Int = self.y | ~~expr~~ r + self.x)



Better evaluate $expr_2$ in the outer scope (A) instead of A' as $expr_2$ needs to be evaluated even with empty base set (as given by $expr_1$).

Iterate Example

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

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

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

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

$$\mathsf{allInstances}_C \text{ -> } \mathsf{iterate}(self : C; w : Bool = \mathit{true} \mid w \wedge self\,.\,x = 0)$$

$$\mathsf{allInstances}_C \text{ -> } \mathsf{forAll}(self : C \mid self\,.\,x = 0)$$

$$\mathsf{context}\ self : C\ \mathsf{inv} : self\,.\,x = 0$$

$$\mathsf{context}\ C\ \mathsf{inv} : x = 0$$

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...?

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

  - Plain syntax error:

$$\mathsf{context}\ C : \mathit{false}$$

  - Subtle syntax error:

$$\mathsf{context}\ C\ \mathsf{inv} : y = 0$$

  - Type error:

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

Casting in the Type System

One Possible Extension: Implicit Casts

- We **may wish** to have

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

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:

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

- 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

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

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.

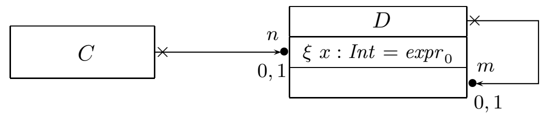
- **Note**: OCL has a far more elaborate type system which in particular addresses the relation between $Bool$ and Int (cf. [OMG, 2006]).

Visibility in the Type System

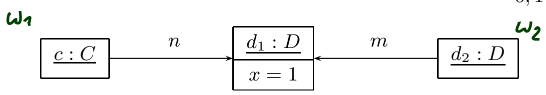
Visibility — The Intuition

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

Let's study an **Example**:



and



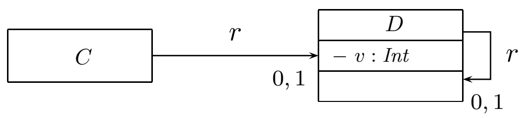
Assume $w_1 : \tau_C$ and $w_2 : \tau_D$ are logical variables. **Which** of the following **syntactically correct** (?) OCL expressions **shall** we consider to be **well-typed**?

ξ of x:	public	private	protected	package
$w_1 . n . x = 0$	✔ |	✔	later	not
	✘	✘ |		
	?	?		
$w_2 . m . x = 0$	✔ |	✔ |||	later	not
	✘	✘ |		
	?	?		

privateness is by class, not by object

Context

- **Example**: A problem?



$$self : \tau_D \vdash \overset{D}{self . r} . v > 0$$

$$self : \tau_C \nvdash \underset{C}{self . r} . v > 0$$

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

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

- In a sense, already preparing to treat "protected" later (when doing inheritance).

Attribute Access in Context

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

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

- **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:

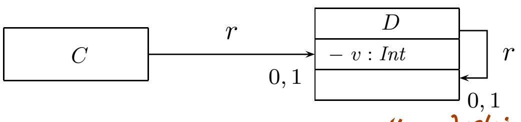
$$(Attr_1) \quad \frac{A, D \vdash expr_1 : \tau_C}{A, D \vdash v(expr_1) : \tau}, \quad \begin{array}{l}\langle v : \tau, \xi, expr_0, P_{\mathscr{C}} \rangle \in atr(C), \\ \xi = +, \text{ or } \xi = - \text{ and } C = D\end{array}$$

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

Attribute Access in Context Example

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

$$(Attr_1) \quad \frac{A, D \vdash expr_1 : \tau_C}{A, D \vdash v(expr_1) : \tau}, \quad \begin{array}{l}\langle v : \tau, \xi, expr_0, P_{\mathscr{C}} \rangle \in atr(C), \\ \xi = +, \text{ or } \xi = - \text{ and } C = D\end{array}$$



Example:

actually stuck: rule's side-condition $C = D$ is not met

$$\frac{\dfrac{\dfrac{\vdots}{A, C \vdash r(self) : D}}{A, C \vdash v(r(self)) : Int} \qquad \dfrac{\dfrac{\dfrac{}{\vdash 0 : Int}}{A \vdash 0 : Int}}{A, C \vdash 0 : Int}}{\underbrace{self : \tau_C}_{A}, C \vdash \underbrace{self \,.\, r \,.\, v}_{} > 0}$$

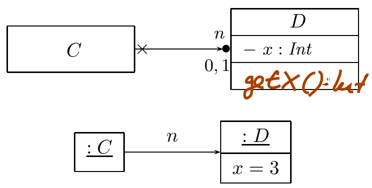
thus not well typed

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?

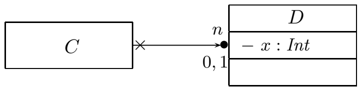


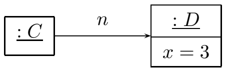
- Visibility is a property of attributes — is it useful to consider it in OCL?
- In other words: given the picture above, **is it useful** to state the following invariant (even though x is private in D)

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

getX()

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

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

yesterday

today

- 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

[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.