Software Design, Modelling and Analysis in UML

Lecture 06: Type Systems and Visibility

2012-11-13

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).
- **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:**
  - Class diagram semantics.
  - Stereotypes – for documentation.
  - 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.

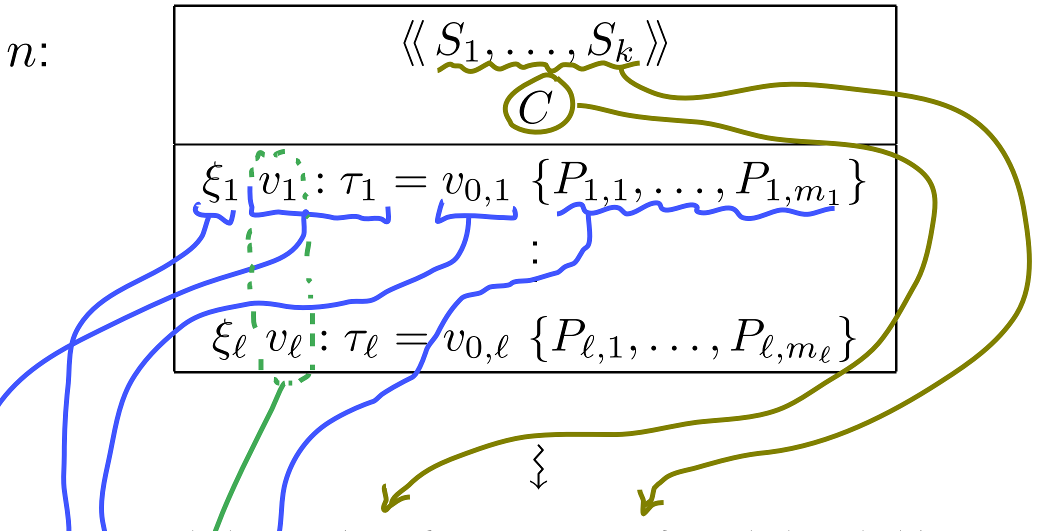
    We define $P_{\mathscr{C}}$ analogously to stereotypes.

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:



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

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

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

where

- "abstract" is determined by the font:

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

- "active" is determined by the frame:

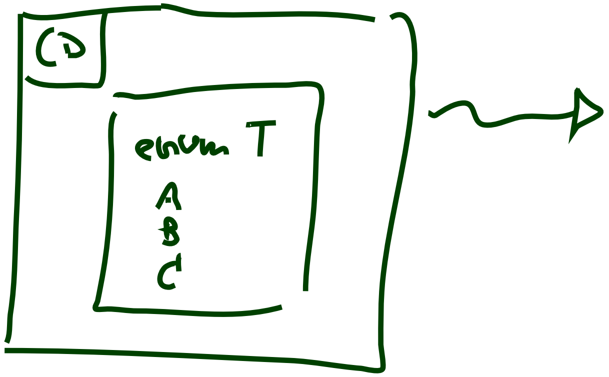
$$t(n) = \begin{cases} \textit{true} & \text{, if } n = \boxed{\mathsf{C}} \text{ or } n = \boxed{\boxed{\mathsf{C}}} \\ \textit{false} & \text{, otherwise} \end{cases}$$

Class Diagram Semantics

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$? **No.**

  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type τ, i.e. the set $\mathscr{D}(\tau)$, would be determined by the class diagram, and not free for choice.)



$\mathscr{S} = (\{T\}, \dots)$

$\mathscr{D}(T) = \{A, B, C\}$

Semantics

- The semantics of a set of **class diagrams** $\mathscr{CD}$ first of all is the induced (extended) **signature** $\mathscr{S}(\mathscr{CD})$.
- The **signature** gives rise to a set of **system states** given a **structure** $\mathscr{D}$.
- Do we need to redefine/extend $\mathscr{D}$? **No.**

  (Would be different if we considered the definition of enumeration types in class diagrams. Then the domain of an enumeration type τ, i.e. the set $\mathscr{D}(\tau)$, would be determined by the class diagram, and not free for choice.)
- What is the effect on $\Sigma_{\mathscr{S}}^{\mathscr{D}}$? **Little.**

  For now, we only **remove** abstract class instances, i.e.

  $$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_*)))$$

  is now **only** called **system state** if and only if, for all $\langle C, S_C, 1, t \rangle \in \mathscr{C}$,

  $$\mathrm{dom}(\sigma) \cap \mathscr{D}(C) = \emptyset.$$

  With $a = 0$ as default "abstractness", the earlier definitions apply directly. We'll revisit this when discussing inheritance.

What About The Rest?

- **Classes**:
  - **Active**: not represented in σ.
    Later: relevant for behaviour, i.e., how system states evolve over time.
  - **Stereotypes**: in a minute.
- **Attributes**:
  - **Initial value**: not represented in σ.
    Later: provides an initial value as effect of "creation action".
  - **Visibility**: not represented in σ.
    Later: viewed as additional **typing information** for well-formedness of system transformers; and with inheritance.
  - **Properties**: such as `readOnly`, `ordered`, `composite`
    (**Deprecated** in the standard.)
    - `readOnly` — **later** treated similar to visibility.
    - `ordered` — too fine for our representation.
    - `composite` — cf. lecture on associations.

Stereotypes

Stereotypes as Labels or Tags

- So, a class is

  $$\langle C, S_C, a, t\rangle$$

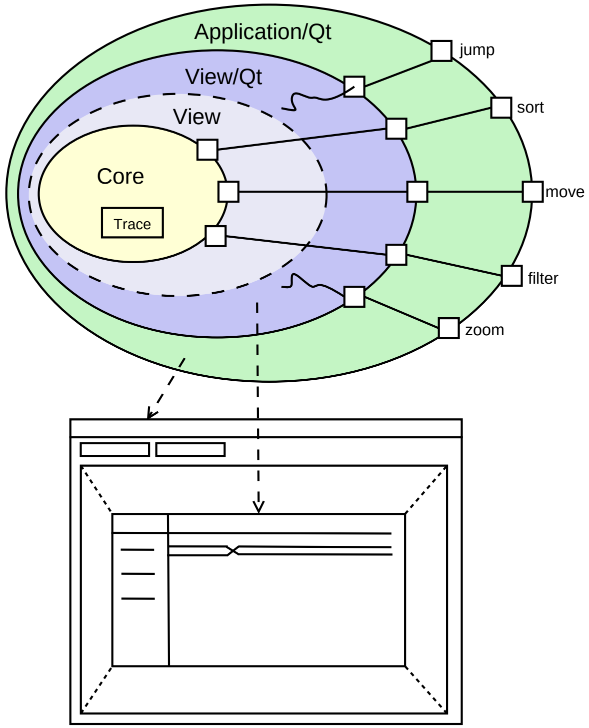
  with a the abstractness flag, t activeness flag, and S_C a set of **stereotypes**.

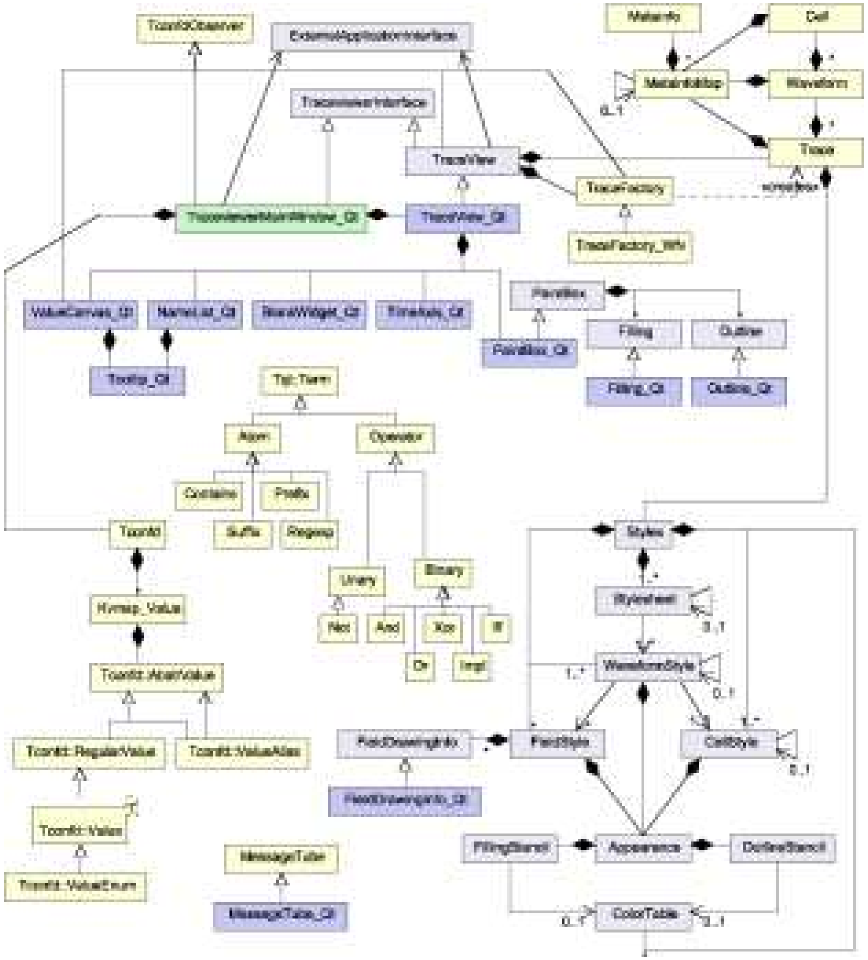
- What are Stereotypes?
  - **Not** represented in system states.
  - **Not** contributing to typing rules.
    (cf. type theory for UML **later**)

- [Oestereich, 2006]:
  View stereotypes as (additional) "**labelling**" ("tags") or as "**grouping**".

  Useful for documentation and MDA.
  - **Documentation**: e.g. layers of an architecture.
    Sometimes, packages (cf. the standard) are already sufficient and "right".
  - **Model Driven Architecture** (MDA): **later**.

Example: Stereotypes for Documentation





- Example: Timing Diagram Viewer [Schumann et al., 2008]
- Architecture of four layers:
    - core, data layer
    - abstract view layer
    - toolkit-specific view layer/widget
    - application using widget
- Stereotype “=” layer “=” colour

Stereotypes as Inheritance

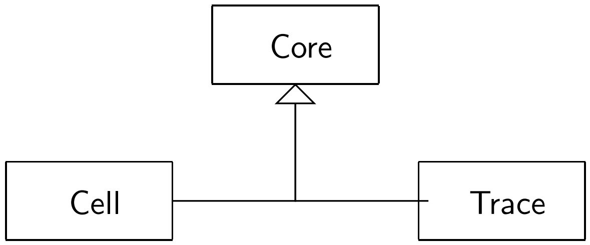
- Another view (due to whom?): distinguish
  - **Technical Inheritance**

    If the **target platform**, such as the programming language for the implementation of the blueprint, is object-oriented, assume a 1-on-1 relation between inheritance in the model and on the target platform.
  - **Conceptual Inheritance**

    Only meaningful with a **common idea** of what stereotypes stand for. For instance, one could label each class with the team that is responsible for realising it. Or with licensing information (e.g., LGPL and proprietary).
    Or one could have labels understood by code generators (cf. lecture on MDSE).

- **Confusing**:
  - Inheritance is often referred to as the "is a"-relation.
    Sharing a stereotype also expresses "being something".

  - We can always (ab-)use UML-inheritance for the conceptual case, e.g.



Excursus: Type Theory (cf. Thiemann, 2008)

Type Theory

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

$$\begin{array}{rlll} expr ::= & w & : \tau & \text{...logical variable } w \\ & \mid \mathsf{true} \mid \mathsf{false} & : Bool & \text{...constants} \\ & \mid 0 \mid -1 \mid 1 \mid \dots & : Int & \text{...constants} \\ & \mid expr_1 + expr_2 & : Int \times Int \to Int & \text{...operation} \\ & \mid \mathsf{size}(expr_1) & : Set(\tau) \to Int & \end{array}$$

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

$$\mathsf{not}\, w$$

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

$$
\begin{array}{lcl}
(BOOL) & \dfrac{}{\vdash B : Bool}, & B \in \{\mathit{true}, \mathit{false}\} \\[2ex]
(INT) & \dfrac{}{\vdash N : Int}, & N \in \{0, 1, -1, \dots\} \\[2ex]
(Fun_0) & \dfrac{\vdash expr_1 : \tau_1 \quad \dots \quad \vdash expr_n : \tau_n}{\vdash \omega(expr_1, \dots, expr_n) : \tau}, & \omega : \tau_1 \times \dots \times \tau_n \to \tau, \\
 & & n \geq 1,\ \omega \notin atr(\mathscr{C})
\end{array}
$$

Example:

- not *true*

$$\dfrac{\dfrac{}{\vdash true : Bool}}{\vdash not\ true : Bool}$$

- *true* + 3

get stuck — we cannot derive this from the rules

$$\dfrac{\vdash true : Int \qquad \dfrac{}{\vdash 3 : Int}}{\vdash true + 3 : Int}$$

↪ true + 3 is not well-typed

- isEmpty({1, 2}):

isEmpty({}(1, 2))

$$\dfrac{\dfrac{(Int)\ \dfrac{}{\vdash 1 : Int} \qquad \dfrac{}{\vdash 2 : Int}\ (Int)}{\vdash \{\}(1, 2) : Set(Int)}\ (Fun_0)}{\vdash isEmpty(\{\}(1, 2)) : Bool}\ (Fun_0)$$

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:

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

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

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

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

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

Type Environment Example

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

$$(\mathit{Fun}_1) \quad \frac{A \vdash \mathit{expr}_1 : \tau_1 \quad \dots \quad A \vdash \mathit{expr}_n : \tau_n}{A \vdash \omega(\mathit{expr}_1, \dots, \mathit{expr}_n) : \tau}, \qquad \begin{array}{l} \omega : \tau_1 \times \dots \times \tau_n \to \tau, \\ n \geq 1,\ \omega \notin \mathit{atr}(\mathscr{C}) \end{array}$$

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

Example:

- $w + 3,\ A = w : \mathit{Int}$

$$\dfrac{(Var)\,\dfrac{w:Int \in \overbrace{w:Int}^{A}}{w:Int \vdash w:Int} \qquad \dfrac{\dfrac{}{\vdash 3:Int}\,(IntT)}{w:Int \vdash 3:Int}\,(EnvIntro)}{\underbrace{w:Int}_{A} \vdash w+3 : Int}\,(Fun_1)$$

⤷ w+3 well-typed in type environment 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:

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

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

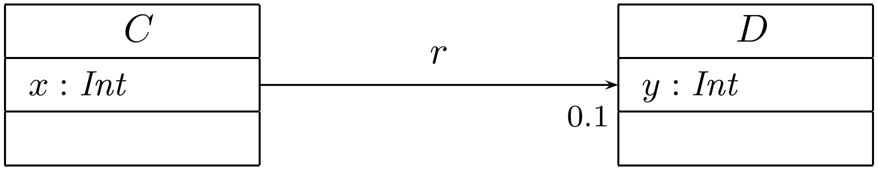
$$(Attr_0^{*}) \qquad \frac{A \vdash expr_1 : \tau_C}{A \vdash r_2(expr_1) : Set(\tau_D)}, \qquad r_2 : D_{*} \in atr(C)$$

Attributes in Type Environment Example

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

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

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



- $self : \tau_C \vdash self.y : Int$
- $self : \tau_C \vdash self.x : Int$ well-typed by $(Attr_0)$, (Var)
- $self : \tau_C \vdash self.r : \tau_D$ well-typed by $(Attr_1)$, (Var)
- $self : \tau_C \vdash self.r.x : Int$ not well-typed, $x \notin Atr(D)$
- $self : \tau_C \vdash self.r.y : Int$ well-typed by

derivable, get stuck because $y \notin atr(C)$

not derivable from type environment

$: \tau_C$

$$\frac{self : \tau_C \vdash self : \tau_D}{self : \tau_C \vdash y(self) : Int}\ (Attr_0)$$

↳ not well typed

$$\dfrac{\dfrac{\dfrac{self : \tau_D \in A}{A \vdash self : \tau_D}\ (Var)}{A \vdash r(self) : \tau_D}\ (Attr_0^{0,1})}{A \vdash y(r(self)) : Int}\ (Attr_0)$$

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:

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

may use

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

"iterator"

"result"

override typing of w_1, w_2 in A

(w_1, w_2 hide outer scope)

Iterate Example

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

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

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

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

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

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

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

$$(Var)\ \dfrac{self : \tau_C \in A'}{A' \vdash self : \tau_C}$$

$$(Attr_0)\ \dfrac{A' \vdash self : \tau_C}{A' \vdash x(self) : Int} \qquad \dfrac{}{\vdash 0 : Int}\ (Int) \qquad \dfrac{\vdash 0 : Int}{A' \vdash 0 : Int}\ ([illegible])$$

$$(Var)\ \dfrac{res : Bool \in A'}{A' \vdash res : Bool} \qquad \dfrac{A' \vdash x(self) : Int \quad A' \vdash 0 : Int}{A' \vdash {=}(x(self), 0)}\ (Fun_1)$$

$$A' := \underbrace{res : Bool,\ self : \tau_C}\ \vdash and(res, =(x(self), 0))\quad (Fun_1)$$

$$\dfrac{}{\vdash allInstances_C : Set(\tau_C)}\ (AllInst) \qquad \dfrac{}{\vdash true : Bool}\ (Bool)$$

$$\dfrac{\vdash allInstances_C : Set(\tau_C) \quad \vdash true : Bool \quad res : Bool,\ self : \tau_C \vdash and(res, =(x(self), 0))}{\vdash allInstances_C \rightarrow iterate(self : \tau_C ; res : Bool = true \mid and(res, =(x(self), 0))) : Bool}\ (Iter)$$

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

⇝ 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:

$$\text{context } C \overset{\text{inv}}{\downarrow} : \textit{false}$$

- Type error:

$$\text{context } C \text{ inv} : \overset{\text{not attribute of } C}{\overset{\downarrow}{y}} = 0$$

- Type error:

$$\text{context } \mathit{self} : C \text{ inv} : \overbrace{\mathit{self}\ .\ n}^{: \tau_D} \overset{\text{↯}}{=} \overbrace{\mathit{self}\ .\ n\ .\ x}^{: Int}$$

References

References

[Oestereich, 2006] Oestereich, B. (2006). *Analyse und Design mit UML 2.1, 8. Auflage*. Oldenbourg, 8. edition.

[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.

[Schumann et al., 2008] Schumann, M., Steinke, J., Deck, A., and Westphal, B. (2008). Traceviewer technical documentation, version 1.0. Technical report, Carl von Ossietzky Universität Oldenburg und OFFIS.