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 \( C_1 \) and \( C_2 \) only differ in visibility of some attributes, then \( C_1 \approx C_2 \) for each \( \theta \).
- 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.

Type Theory

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

- \( \text{expr} ::= \ldots \text{logical variable} \ldots \)
- \( \text{expr} ::= \text{true} \) \( \mid \text{false} \) \( \mid \text{0} \mid \text{−1} \mid \text{1} \mid \ldots \) \( \in \) \( \text{Int} \)
- \( \text{expr} ::= \ldots \text{constant} \ldots \)
- \( \text{expr} ::= \text{expr}_1 \text{+} \text{expr}_2 \) \( \in \text{Int} \times \text{Int} \rightarrow \text{Int} \) \( \in \) \( \text{operation} \)
- \( \text{expr} ::= \ldots \text{size}(\ldots) \) \( \in \text{Set}(\ldots) \rightarrow \text{Int} \)

Wanted: A procedure to tell well-typed, such as \((w : \text{Bool})\) not \(\in\) from not well-typed, such as \(\text{size}(w)\).

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

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

for some OCL type \( \tau \), i.e., \( \tau \in \text{T}_B \cup \text{T}_V \cup \{ \text{Set}(\tau_0) \mid \tau_0 \in \text{T}_B \cup \text{T}_V \}, C \in \mathbb{W} \).

A Type System for OCL

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

\[ (\text{"name"}) \quad (\text{"premise"}) \quad (\text{"side condition"}) \]

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

(i) Universal well-typedness:

\[ \Gamma \vdash \text{expr} : \tau \]

\( \Gamma \vdash \text{true} : \text{Bool} \) \( \vdash \text{false} : \text{Bool} \)

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

\[ \Delta \vdash \text{expr} : \tau \]

\( \Delta \vdash \text{self} : \tau \) \( \vdash \text{self}.v : \text{Int} \)

(iii) Well-typedness in type environment \( \Delta \) and context \( D \) (for visibility)

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

\( \Delta \vdash \text{self} : \text{C} \) \( \vdash \text{self}.v : \text{Int} \)
If type $\tau$ is a type of logical variables $\omega$ and $\phi$ then $\omega \vdash \tau$, $\phi \vdash \tau$, $\omega \vdash \phi$, and $\tau_1 \vdash \tau_2$, then

\[
A \vdash \tau \quad \text{and} \quad W \vdash \tau,
\]

where $A$, $\omega$, and $\phi$ are $\tau$-type variables, $\omega$ is a $\tau$-type variable, and $\phi$ and $\tau$ are $\tau$-type variables.

### Constants and Operations

\[\begin{align*}
\text{true} & : \text{BOOL} \\
\text{false} & : \text{BOOL} \\
\end{align*}\]

### Constants and Operations Example

\[\begin{align*}
A & \equiv \text{true} \\
\phi & \equiv \text{false} \\
\end{align*}\]
First Recapitulation

If an expression is a well-typed OCL expression, then the iterator variable must be of type consistent with the base set, and when it is used in an accessor expression, any accessor expression must be defined for well-typed expressions. For an object of type \( C \), let \( \tau \) denote the type of an attribute of type \( C \). If \( \tau \) is defined for well-typed expressions, then the iterator variable must be of type consistent with the base set. If an expression is a well-typed OCL expression, then the iterator variable must be of type consistent with the base set, and when it is used in an accessor expression, any accessor expression must be defined for well-typed expressions.
Casting in the Type System

One Possible Extension: Implicitcasts

- We may wish to have
  \[ 1 \text{ and false : Bool} \]

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

- With \((\text{Cast})\) and \((\text{Int})\), 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 \((1 \text{ and false})\)?

- First of all, we have (grammar)
  \[ \text{expr}_1 \text{ and } \text{expr}_2 : \text{Bool} \times \text{Bool} \rightarrow \text{Bool} \]

- Thus,
  \[ I(\text{and}) : I(\text{Bool}) \times I(\text{Bool}) \rightarrow I(\text{Bool}) \]
  where \(I(\text{Bool}) = \{\text{true, false}\} \cup \{\bot\}\).

- By definition,
  \[ I(1 \text{ and false})(a, b) = I(\text{and})(I(1)(a, b), I(\text{false})(a, b)) \]
  and there we're stuck.

- Explicitly define
  \[ I(\text{and})(\text{expr}_1, \text{expr}_2)(a, b) = \begin{cases} b_1 \land b_2 & \text{if } b_1 \neq \bot \text{ Bool} \neq b_2 \\ \bot \text{ Bool} & \text{otherwise} \end{cases} \]

where

- \( b_1 = \text{toBool}(I(\text{Int})(\text{expr}_1)(a, b)) \),
- \( b_2 = \text{toBool}(I(\text{Int})(\text{expr}_2)(a, b)) \),

and where

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

  \[ x \rightarrow \begin{cases} \text{true} & \text{if } x \in \{\text{true}\} \cup \{0, \bot\text{ Int}\} \\ \text{false} & \text{if } x \in \{\text{false}\} \cup \{\bot\text{ Int}\} \\ \bot \text{ Bool} & \text{otherwise} \end{cases} \]

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]).
\[ C \vdash \text{self} : \tau \]

Example 1, 0

\[ C \tau \text{Int} : v \]

\[ v \rightarrow \text{tosupportvisibility.} \]

I adjust the interpretation function \( \nu \) so that it need not depend on \( \nu \) later (when doing inheritance).

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 \( \text{vis} \) is private in \( D \))

\[ \text{context} C \vdash v \rightarrow \text{vis} > 0 ? \]
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 interface", "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).

Goals and operation bodies:

If it doubt, yes (= do take visibility into account).

Any so called "action language" typically takes visibility into account.

References

