Software Design, Modelling and Analysis in UML

Lecture 21: Inheritance II

2013-02-05

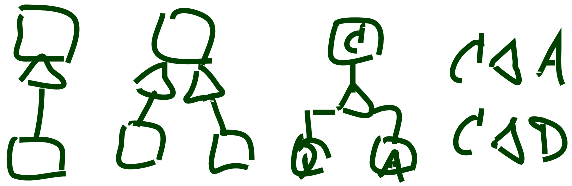
Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Contents & Goals

Last Lecture:

- State Machine semantics completed
- Inheritance in UML: concrete syntax



This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
    - What's the Liskov Substitution Principle?
    - What is late/early binding?
    - What is the subset, what the uplink semantics of inheritance?
    - What's the effect of inheritance on LSCs, State Machines, System States?
    - What's the idea of Meta-Modelling?
- **Content:**
    - Liskov Substitution Principle — desired semantics
    - Two approaches to obtain desired semantics

Inheritance: Syntax

Recall: Abstract Syntax

Recall: a signature (with signals) is a tuple $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$.

Now (finally): extend to

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$$

where F/mth are methods, analogously to attributes and

$$\lhd \ \subseteq (\mathscr{C} \times \mathscr{C}) \cup (\mathscr{E} \times \mathscr{E})$$

is a **generalisation** relation such that $C \lhd^+ C$ for **no** $C \in \mathscr{C}$ ("acyclic").

$C \lhd D$ reads as

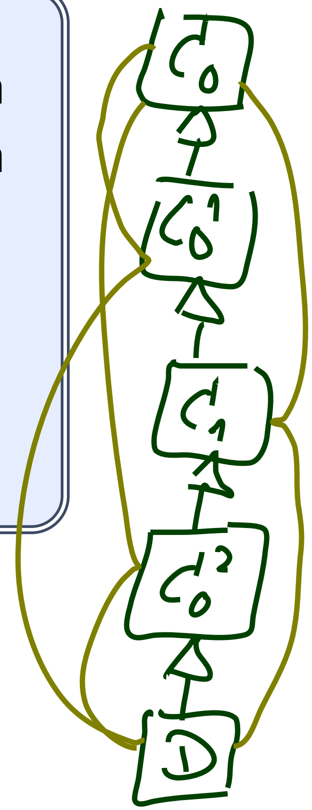
- C is a generalisation of D,
- D is a specialisation of C,
- D inherits from C,
- D is a sub-class of C,
- C is a super-class of D,
- ...

Recall: Reflexive, Transitive Closure of Generalisation

Definition. Given classes $C_0, C_1, D \in \mathscr{C}$, we say D inherits from C_0 **via** C_1 if and only if there are $C_0^1, \dots C_0^n, C_1^1, \dots C_1^m \in \mathscr{C}$ such that

$$C_0 \;\lhd C_0^1 \lhd \dots C_0^n \lhd\; C_1 \;\lhd C_1^1 \lhd \dots C_1^m \lhd\; D.$$

We use '$\preceq$' to denote the reflexive, transitive closure of '$\lhd$'.



In the following, we assume

- that all attribute (method) names are of the form

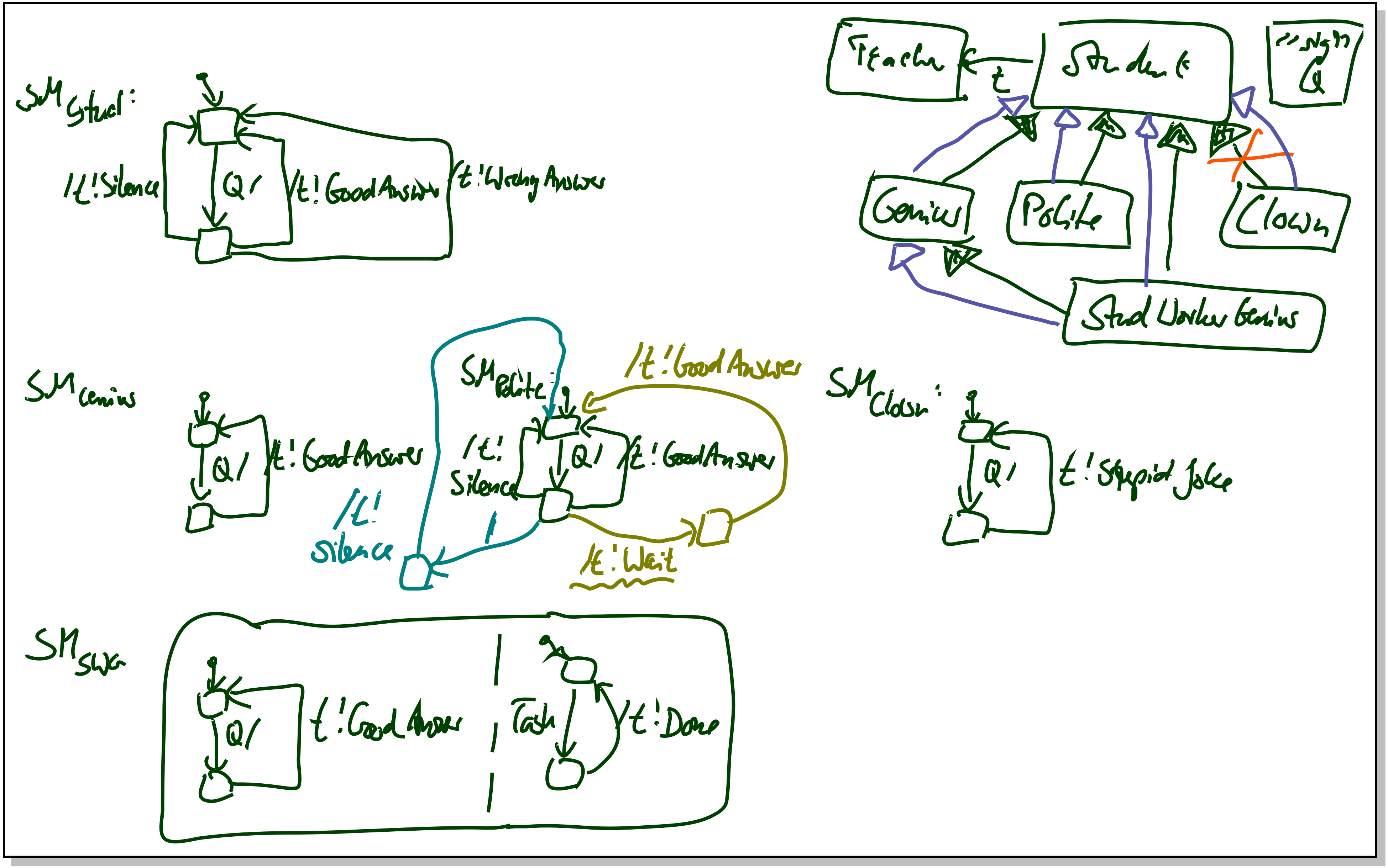
$$C::v, \quad C \in \mathscr{C} \cup \mathscr{E} \qquad (C::f, \quad C \in \mathscr{C}),$$

- that we have $C::v \in atr(C)$ resp. $C::f \in mth(C)$ **if and only if** v (f) appears in an attribute (method) compartment of C in a class diagram.

We still want to accept "context C inv : $v < 0$", which v is meant? Later!

Inheritance: Desired Semantics

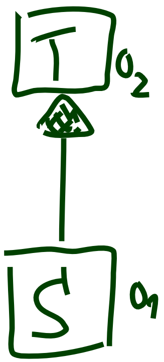
Recall


SM_Stud:
/t!Silence
Q/
/t!GoodAnswer
/t!WrongAnswer
Teacher
Student
t
Q
Genius
Polite
Clown
StudUnderGenius
SM_Genius
Q/
/t!GoodAnswer
/t!Silence
SM_Polite:
/t!Silence
Q/
/t!GoodAnswer
/t!Wait
SM_Clown:
Q/
/t!StupidJoke
Q/
t!GoodAnswer
Task
/t!Done

Desired Semantics of Specialisation: Subtyping

There is a classical description of what one **expects** from **sub-types**, which in the OO domain is closely related to inheritance:

The principle of type substitutability [Liskov, 1988, Liskov and Wing, 1994].
(**Liskov Substitution Principle** (LSP).)



"If for each object o_1 of type S there is an object o_2 of type T such that
for all programs P defined in terms of T,
the behavior of P is unchanged when o_1 is substituted for o_2
then S is a **subtype** of T."

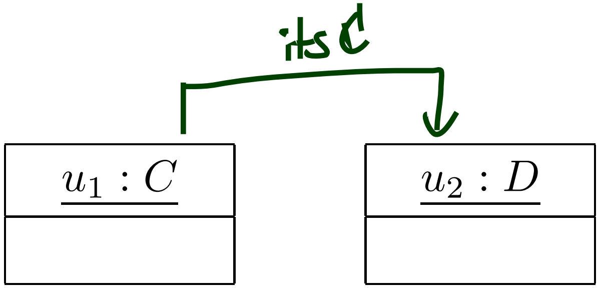
$\forall o_1 \in S\ \exists o_2 \in T \bullet_{\pi \in} [\![P]\!](o_1) \Rightarrow \underset{\pi[o_1/o_2]"}{[\![P]\!](o_1/o_2)}$

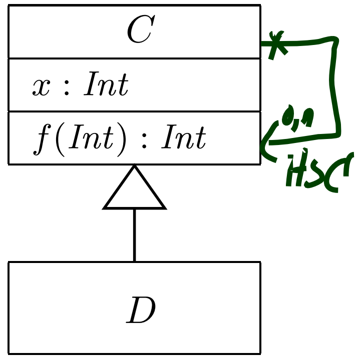
In other words: [Fischer and Wehrheim, 2000]

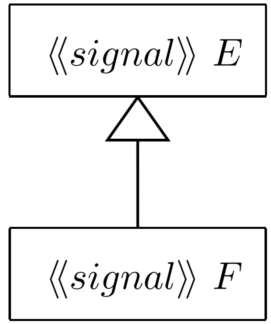
"An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected,
without a client being able to tell the difference."

So, what's "**usable**"? Who's a "**client**"? And what's a "**difference**"?

"...shall be usable..."?







$I[\![\varphi]\!](\sigma, \{self \mapsto u_1\})$

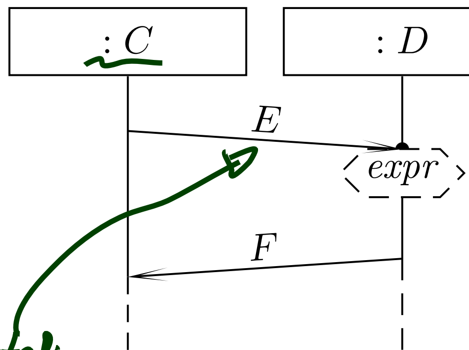
vs. $I[\![\varphi]\!](\sigma, \{self \mapsto u_2\})$

- **OCL**:
  - context C inv : $x > 0 =: \varphi$
- **Actions**:
  - $itsC.x = 0$
  - $itsC.f(0)$
  - $itsC\ !\ F$

  should apply to u_1 and u_2
- **Triggers**:
  - $E[\dots]/\dots$

wanted! bind u_1 as well as u_2 to this instance line

- **Sequence Diagrams**:

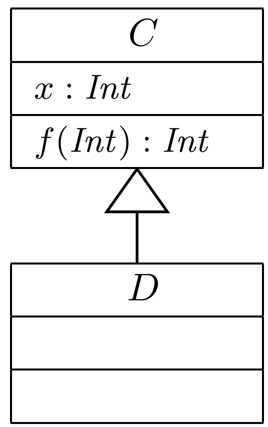


wanted: use F instances here

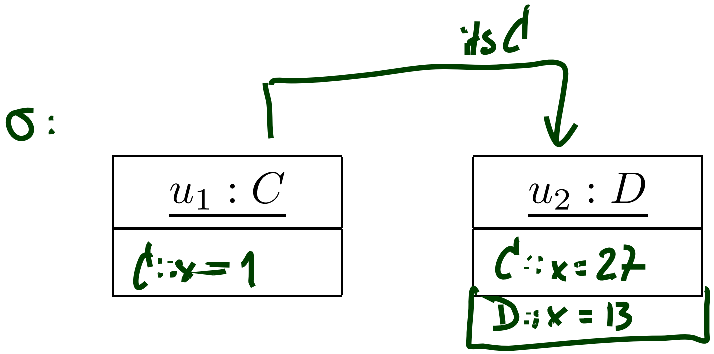
"...a client..."?

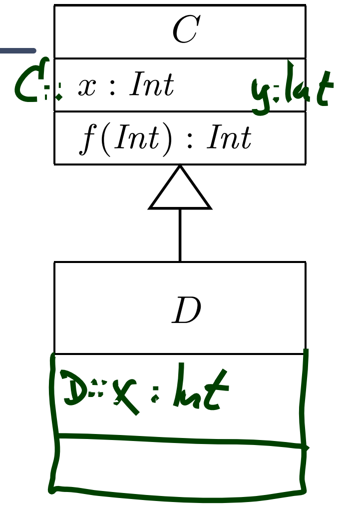
"An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected, without **a client** being able to tell the **difference**."

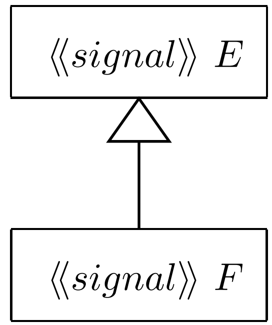
- **Narrow** interpretation: another object in the model.
- **Wider** interpretation: another modeler.



“...can’t tell difference...”?







- **OCL**:

  {self ↦ u_1} {self ↦ u_2}

  - $I[\![\text{context } C \text{ inv} : x > 0]\!](\sigma, \emptyset)$ vs. $I[\![\text{context } C \text{ inv} : x > 0]\!](\sigma, \emptyset)$

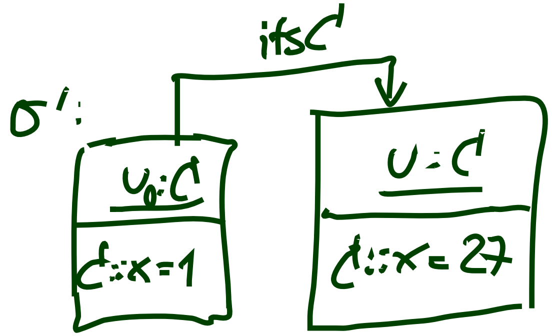
  : usage

LSP:

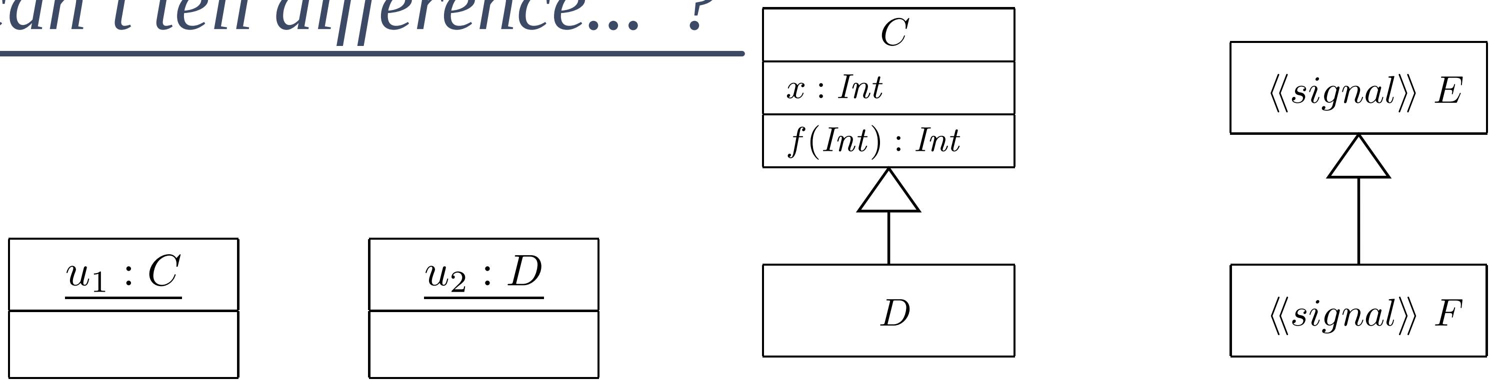
$I[\![\varphi]\!](\sigma, \{self \mapsto v_2\}) = true$

$\Rightarrow \exists \sigma', v \in \mathcal{D}(C) \bullet$

$I[\![\varphi]\!](\sigma', \{self \mapsto v\}) = true$



"...can't tell difference..."?



- **Triggers**, **Actions**: if

$$(\sigma_0[u_1/u_2], \varepsilon_0) \xrightarrow[u_2]{(cons_0, Snd_0)} (\sigma_1[u_1/u_2], \varepsilon_1)$$

is possible, then

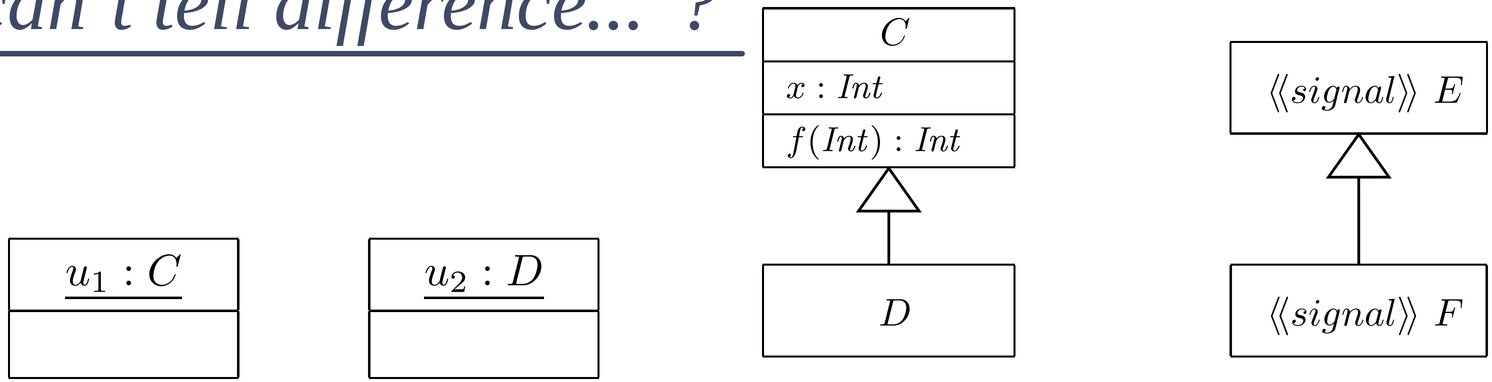
$$(\overbrace{\sigma_0, \varepsilon_0}^{[u_2/u_1]}) \xrightarrow[u_1]{(cons_0, Snd_0)} (\overbrace{\sigma_1, \varepsilon_1}^{[u_2/u_1]})$$

should be possible – sub-type does less on inputs of super-type.

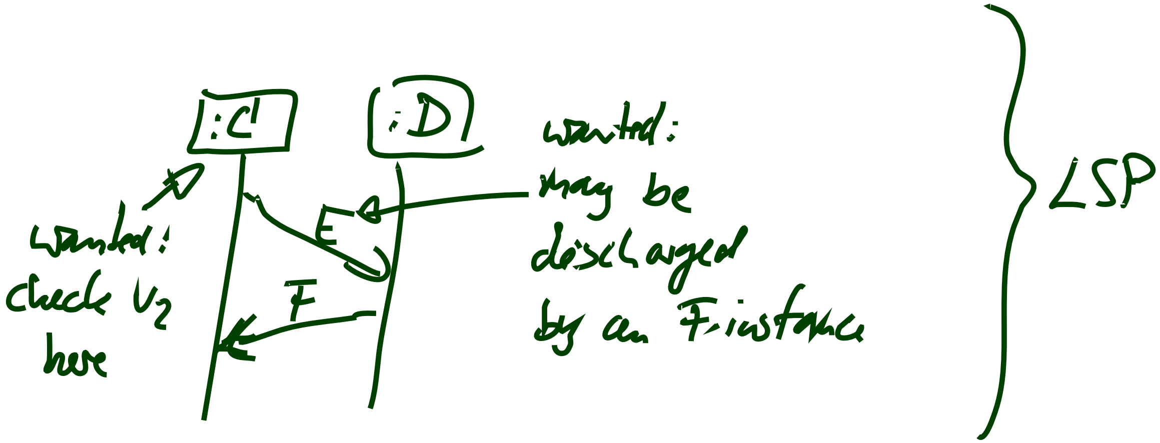
LSP

for some u_1 and
a proper definition of '$[\cdot/\cdot]$'

“...can’t tell difference...”?



- **Sequence Diagram**: $w[u_1/u_2] \in \mathcal{L}(\mathcal{B}_L)$ implies $w \in \mathcal{L}(\mathcal{B}_L)$.



Motivations for Generalisation

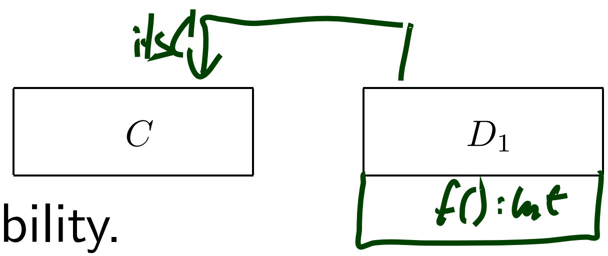
- **Re-use**,
- **Sharing**,
- **Avoiding Redundancy**,
- **Modularisation**,
- **Separation of Concerns**,
- **Abstraction**,
- **Extensibility**,
- ...

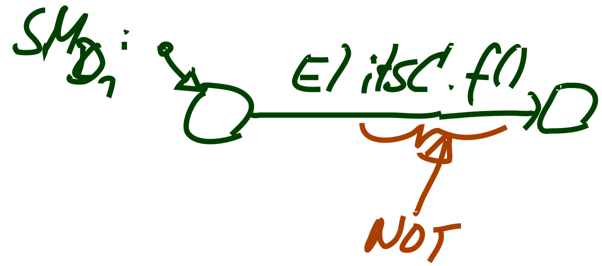
$\rightarrow$ See textbooks on object-oriented analysis, development, programming.

What Does [Fischer and Wehrheim, 2000] Mean for UML?

> "An instance of the **sub-type** shall be **usable** whenever an instance of the supertype was expected, without **a client** being able to tell the **difference**."

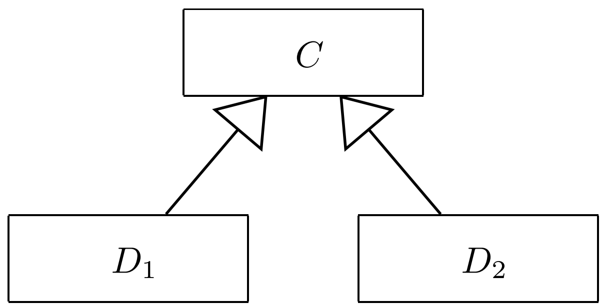
- Wanted: sub-typing for UML.
- With





we don't even have usability.

- It would be nice, if the well-formedness rules and semantics of



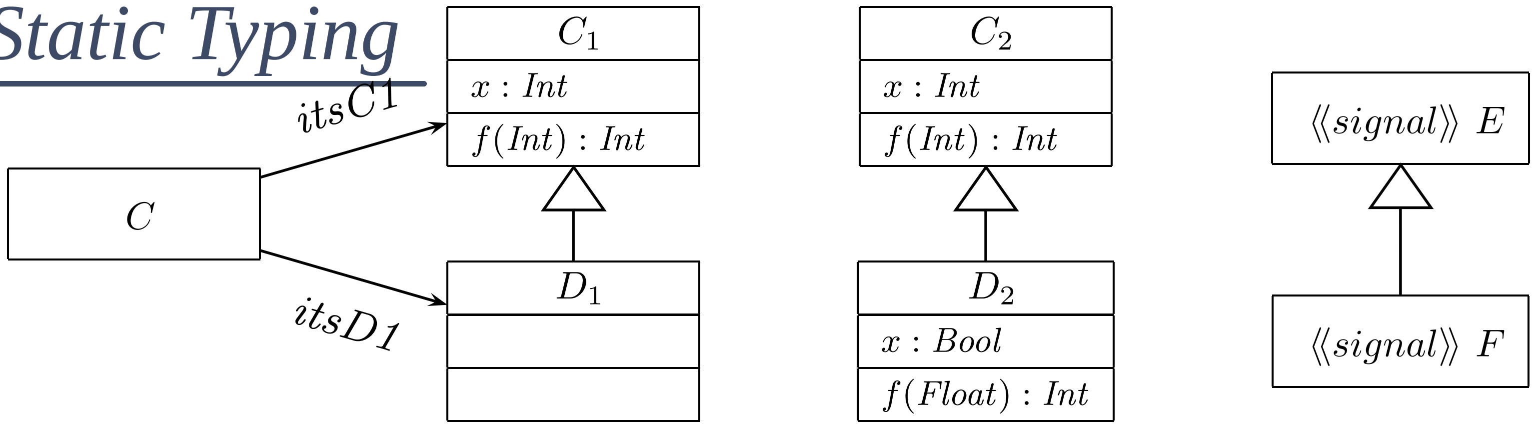
would ensure D_1 **is a sub-type** of C:

  - that D_1 objects can be used interchangeably by everyone who is using C's,
  - is not able to tell the difference (i.e. see unexpected behaviour).

“...shall be usable...” for UML

Easy: Static Typing

Given:



context ∘ inv: ⌴x > 0 :

Wanted:

- $x > 0$ also **well-typed** for D_1

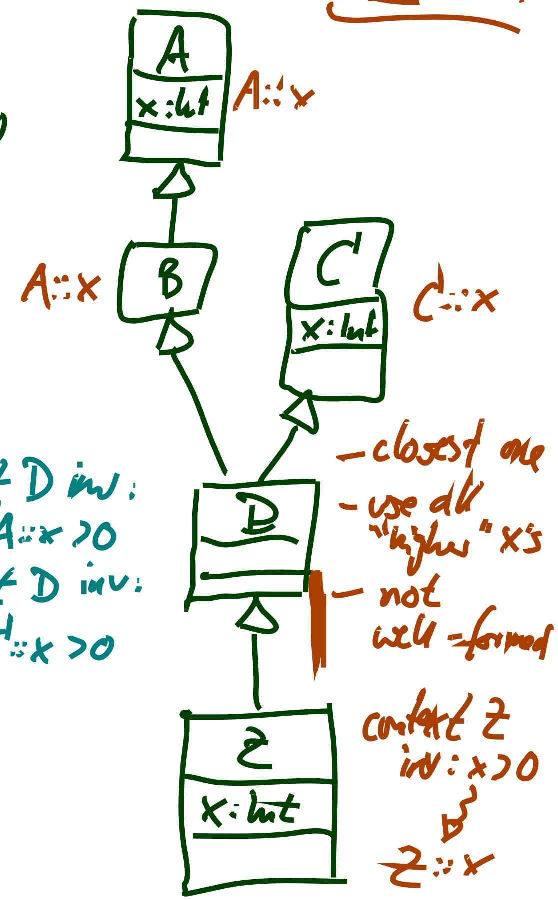
  e.g. context D_1 inv: $x > 0$ ⇝ context D_1 inv: C_1::$x > 0$
- assignment $itsC1 := itsD1$ being **well-typed**
- $itsC1.x = 0$, $itsC1.f(0)$, $itsC1 \; ! \; F$
  being well-typed (and doing the right thing).

OK:
context D inv: A::x > 0
context D inv: C::x > 0

Approach:

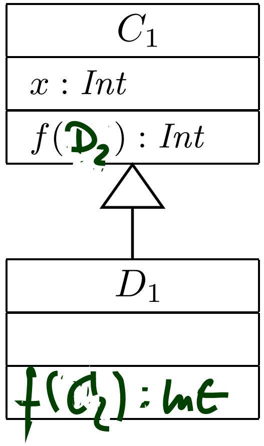
by extending type system

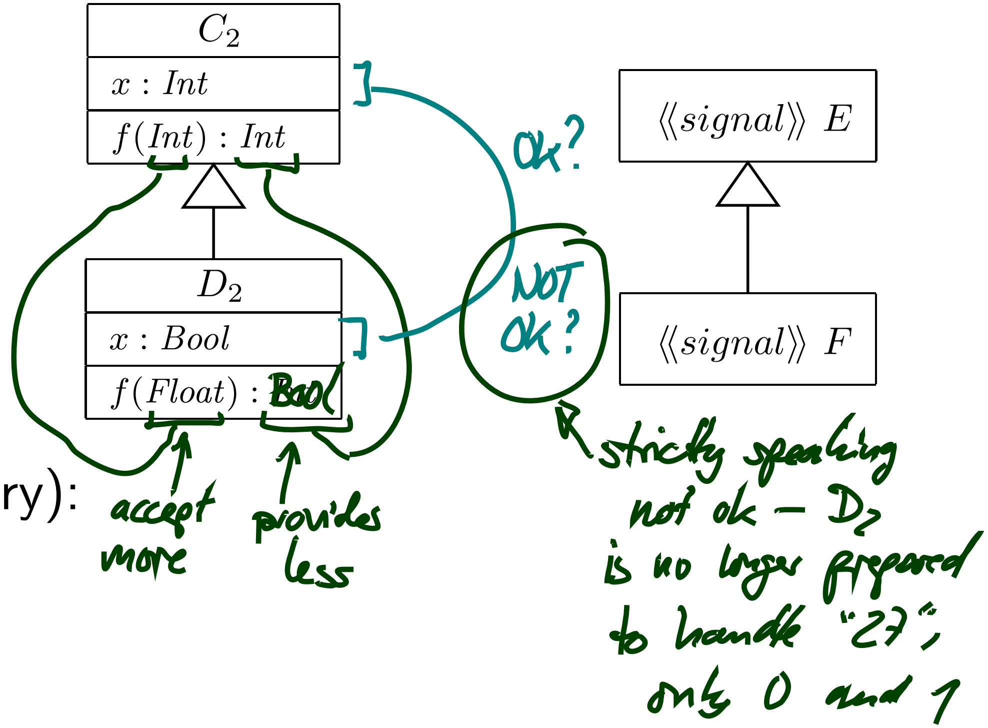
1. Simply define it as being well-typed,
2. adjust system state definition to do the right thing.



Static Typing Cont'd

Assuming $\mathcal{D}(Bool) \subseteq \mathcal{D}(Int) \subseteq \mathcal{D}(Float)$ plus corresponding type-system





Notions (from category theory):

- **invariance**,
- **covariance**,
- **contravariance**.

We could call, e.g. a method, **sub-type preserving**, if and only if it

- accepts **more general** types as input (**contravariant**),
- provides a **more specialised** type as output (**covariant**).

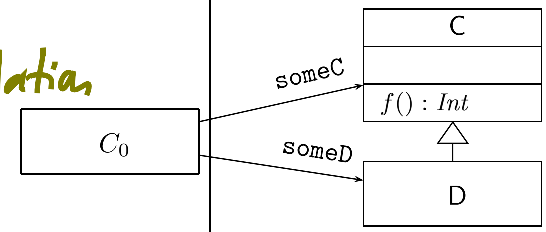
This is a notion used by many programming languages — and easily type-checked.

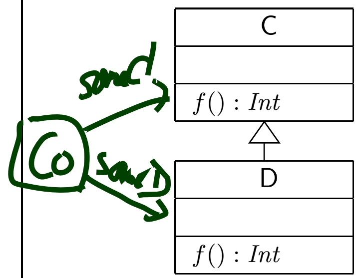
Excursus: Late Binding of Behavioural Features

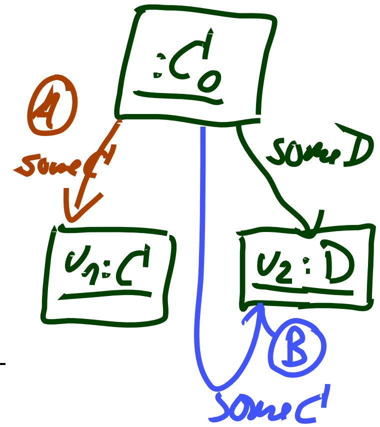
Late Binding

What transformer applies in what situation? (Early (compile time) binding.)

the type of the link determines which implementation is used (not caring for what an object really "is")







	f not overridden in D	f overridden in D	value of someC/someD
someC -> f()	C::f()	C::f()	A
someD -> f()	C::f()	D::f()	A
$\underbrace{\text{someC}}_{:\tau_C}$ -> f()	C::f()	$\underbrace{\text{C::f()}}_{\text{impl. of C}}$	B object actually is a "D"

What one could want is something different: (Late binding.)

type of object determines which implementation is used

someC -> f()	C::f()	C::f()	A
someD -> f()	C::f()	D::f()	A
$\underbrace{\text{someC}}_{:\tau_C}$ -> f()	C::f()	$\underbrace{\text{D::f()}}_{\text{impl. of D}}$	B obj. actually is a "D"

Late Binding in the Standard and Programming Lang.

- In **the standard**, Section 11.3.10, "CallOperationAction":

  "**Semantic Variation Points**
  The mechanism for determining the method to be invoked as a result of a call operation is unspecified." [OMG, 2007b, 247]

- In **C++**,
  - methods are by default "(early) compile time binding",
  - can be declared to be "late binding" by keyword "`virtual`",
  - the declaration applies to all inheriting classes.

- In **Java**,
  - methods are "late binding";
  - there are patterns to imitate the effect of "early binding"

Exercise: What could have driven the designers of C++ to take that approach?

Note: late binding typically applies only to **methods**, **not** to **attributes**.
(But: getter/setter methods have been invented recently.)

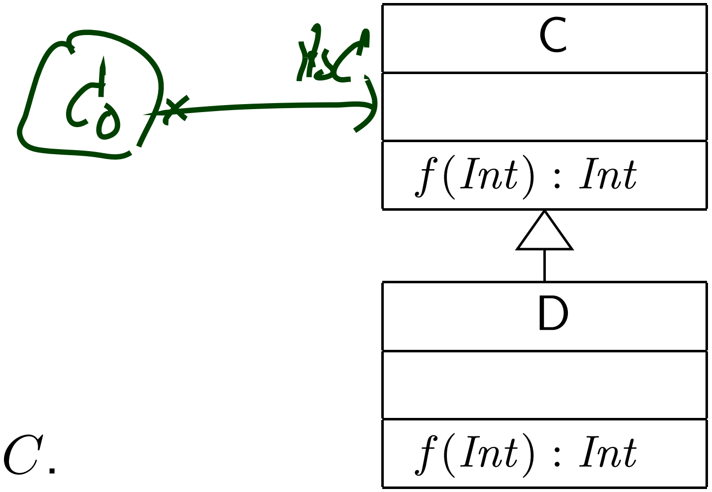
Back to the Main Track: “...tell the difference...” for UML

With Only Early Binding...

- ...we're **done** (if we realise it correctly in the framework).
- Then
  - if we're calling method f of an object u,
  - which is an instance of D with $C \preceq D$
  - via a C-link,
  - then we (by definition) only see and change the C-part.
  - We cannot tell whether u is a C or an D instance.

So we immediately also have behavioural/dynamic subtyping.

Difficult: Dynamic Subtyping



- C::f and D::f are **type compatible**, but D is **not necessarily** a **sub-type** of C.
- **Examples**: (C++)

```
int C::f(int) {
   return 0;
};
```

vs.

```
int D::f(int) {
   return 1;
};
```

```
int C::f(int) {
   return (rand() % 2);
};
```

vs.

```
int D::f(int x) {
   return (x % 2);
};
```

Sub-Typing Principles Cont'd

- In the standard, Section 7.3.36, "**Operation**":

  "**Semantic Variation Points**
  [...] When operations are redefined in a specialization, rules regarding **invariance**, **covariance**, or **contravariance** of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations." [OMG, 2007a, 106]

- So, better: call a method **sub-type preserving**, if and only if

  (i) accepts **more input values** (**contravariant**),

  (ii) on the "**old values**", has **fewer behaviour** (**covariant**).

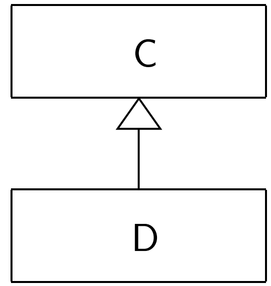
  Note: ~~This~~ (ii) is no longer a matter of simple type-checking!

- And not necessarily the end of the story:
  - One could, e.g. want to consider execution time.
  - Or, like [Fischer and Wehrheim, 2000], relax to "fewer observable behaviour", thus admitting the sub-type to do more work on inputs.

  Note: "testing" differences depends on the **granularity** of the semantics.

- **Related**: "has a weaker pre-condition," (**contravariant**),
  "has a stronger post-condition." (**covariant**).

Ensuring Sub-Typing for State Machines



- In the CASE tool we consider, multiple classes in an inheritance hierarchy can have state machines.

- But the state machine of a sub-class **cannot** be drawn from scratch.

- Instead, the state machine of a sub-class can only be obtained by applying actions from a **restricted** set to a copy of the original one.

  Roughly (cf. User Guide, p. 760, for details),

  - add things into (hierarchical) states,
  - add more states,
  - attach a transition to a different target (limited).

- They **ensure**, that the sub-class is a **behavioural sub-type** of the super class. (But method implementations can still destroy that property.)

- Technically, the idea is that (by late binding) only the state machine of the most specialised classes are running.

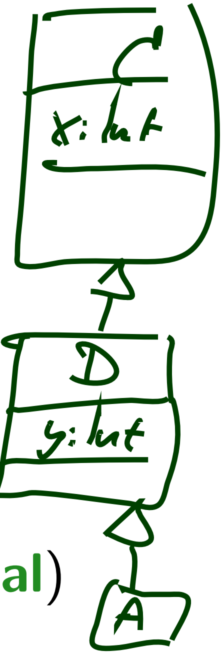
  By knowledge of the framework, the (code for) state machines of super-classes is still accessible — but using it is hardly a good idea...

Towards System States

Wanted: a formal representation of "if $C \preceq D$ then D '**is a**' C", that is,

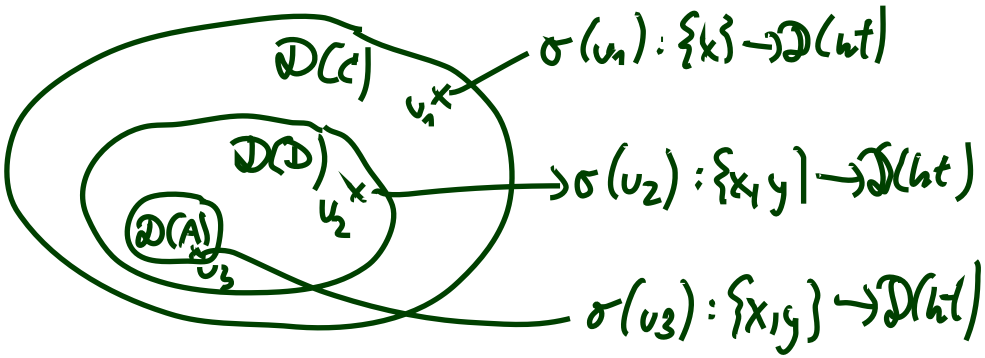
(i) D has the same attributes and behavioural features as C, and

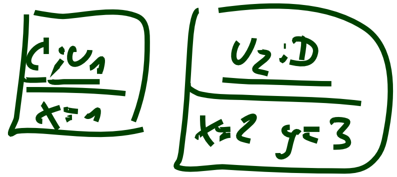
(ii) D objects (identities) can replace C objects.



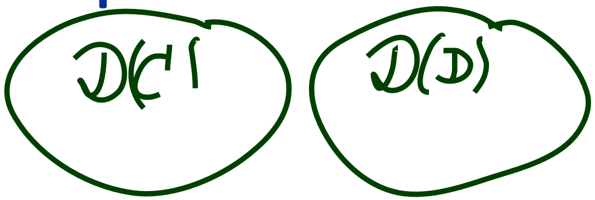
We'll discuss **two approaches** to semantics:

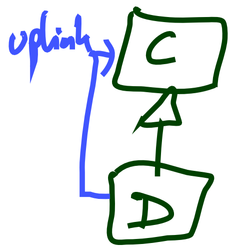
- **Domain-inclusion** Semantics (more **theoretical**)

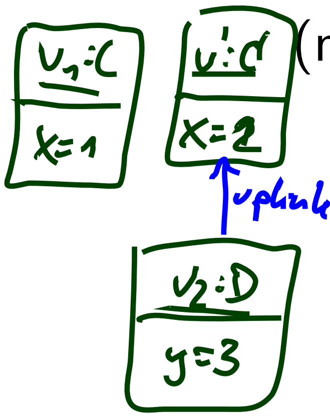




- **Uplink** Semantics (more **technical**)







someD.x $\xrightarrow{\text{rewrite}}$ someD.uplink.x

Domain Inclusion Semantics

Domain Inclusion Structure

Let $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}, F, mth, \lhd)$ be a signature.

Now a **structure** $\mathscr{D}$

- [**as before**] maps types, classes, associations to domains,
- [**for completeness**] methods to transformers,
- [**as before**] indentities of instances of classes not (transitively) related by generalisation are disjoint,

- [**changed**] the indentities of a super-class comprise all identities of sub-classes, i.e.

$$\forall\, C \in \mathscr{C} : \mathscr{D}(C) \supsetneq \bigcup_{C \lhd D} \mathscr{D}(D).$$

Note: the old setting coincides with the special case $\lhd\ = \emptyset$.

Domain Inclusion System States

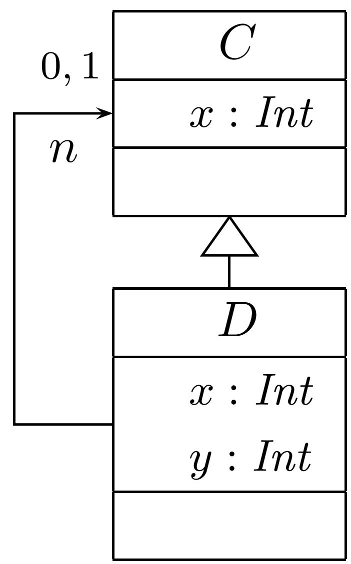
Now: a **system state** of $\mathscr{S}$ wrt. $\mathscr{D}$ is a **type-consistent** mapping

$$\sigma : \mathscr{D}(\mathscr{C}) \nrightarrow (V \nrightarrow (\mathscr{D}(\mathscr{T}) \cup \mathscr{D}(\mathscr{C}_{0,1}) \cup \mathscr{D}(\mathscr{C}_*)))$$

that is, for all $u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)$,

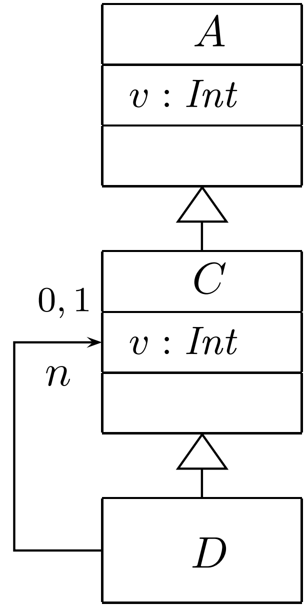
- [**as before**] $\sigma(u)(v) \in \mathscr{D}(\tau)$ if $v : \tau$, $\tau \in \mathscr{T}$ or $\tau \in \{C_*, C_{0,1}\}$.
- [**changed**] $\mathrm{dom}(\sigma(u)) = \bigcup_{C_0 \preceq C} atr(C_0)$,

Example:



Note: the old setting still coincides with the special case $\lhd\ = \emptyset$.

Preliminaries: Expression Normalisation

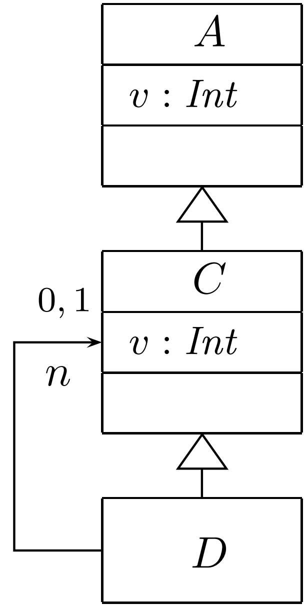


Recall:

- we want to allow, e.g., "context D inv : $v < 0$".
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, v shall denote the
"**most special more general**" $C::v$ according to $\lhd$.

Preliminaries: Expression Normalisation



Recall:

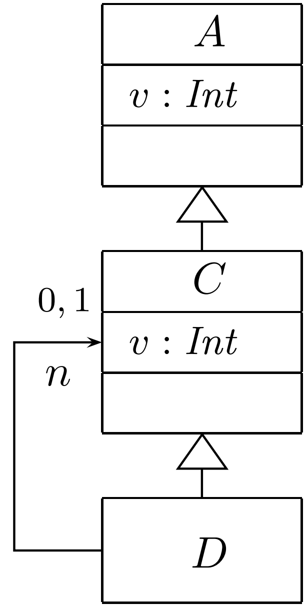
- we want to allow, e.g., "context D inv : $v < 0$".
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, v shall denote the "**most special more general**" $C::v$ according to $\lhd$.

To keep this out of typing rules, we assume that the following **normalisation** has been applied to all OCL expressions and all actions.

- Given expression v (or f) in **context** of class D, as determined by, e.g.
  - by the (type of the) navigation expression prefix, or
  - by the class, the state-machine where the action occcurs belongs to,
  - similar for method bodies,
- **normalise** v to (= replace by) $C::v$,
- where C is the **greatest** class wrt. "$\preceq$" such that
  - $C \preceq D$ and $C::v \in atr(C)$.

Preliminaries: Expression Normalisation



Recall:

- we want to allow, e.g., "context D inv : $v < 0$".
- we assume **fully qualified names**, e.g. $C::v$.

Intuitively, v shall denote the
"**most special more general**" $C::v$ according to $\lhd$.

To keep this out of typing rules, we assume that the following **normalisation** has been applied to all OCL expressions and all actions.

- Given expression v (or f) in **context** of class D, as determined by, e.g.
  - by the (type of the) navigation expression prefix, or
  - by the class, the state-machine where the action occcurs belongs to,
  - similar for method bodies,
- **normalise** v to (= replace by) $C::v$,
- where C is the **greatest** class wrt. "$\preceq$" such that
  - $C \preceq D$ and $C::v \in atr(C)$.

If no (unique) such class exists, the model is considered **not well-formed**; the expression is ambiguous. Then: explicitly provide the **qualified name**.

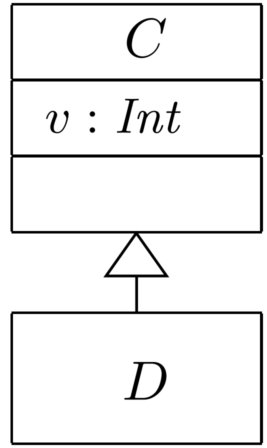
OCL Syntax and Typing

- Recall (part of the) OCL syntax and typing: $v, r \in V$; $C, D \in \mathscr{C}$

$$
\begin{aligned}
expr ::= \;& v(expr_1) && : \tau_C \to \tau(v), && \text{if } v : \tau \in \mathscr{T} \\
\mid\; & r(expr_1) && : \tau_C \to \tau_D, && \text{if } r : D_{0,1} \\
\mid\; & r(expr_1) && : \tau_C \to Set(\tau_D), && \text{if } r : D_*
\end{aligned}
$$

The definition of the semantics remains (textually) **the same**.

More Interesting: Well-Typed-ness



- We want

$$\text{context } D \text{ inv} : v < 0$$

  to be well-typed.

  Currently it isn't because

$$v(expr_1) : \tau_C \to \tau(v)$$

  but $A \vdash self : \tau_D$.

  (Because τ_D and τ_C are still **different types**, although $\mathrm{dom}(\tau_D) \subset \mathrm{dom}(\tau_C)$.)

- So, add a (first) new typing rule

$$\frac{A \vdash expr : \tau_D}{A \vdash expr : \tau_C}, \text{ if } C \preceq D. \qquad \text{(Inh)}$$

  Which is correct in the sense that, if '$expr$' is of type τ_D, then we can use it everywhere, where a τ_C is allowed.

  The system state is prepared for that.

Well-Typed-ness with Visibility Cont'd

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = + \qquad \text{(Pub)}$$

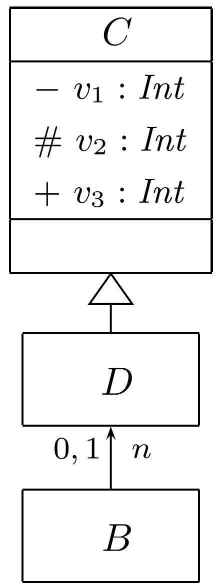
$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = \#, \; C \preceq D \qquad \text{(Prot)}$$

$$\frac{A, D \vdash expr : \tau_C}{A, D \vdash C{::}v(expr) : \tau}, \quad \xi = -, \; C = D \qquad \text{(Priv)}$$

$\langle C{::}v : \tau, \xi, v_0, P \rangle \in atr(C)$.

Example:

context/ inv	$(n.)v_1 < 0$	$(n.)v_2 < 0$	$(n.)v_3 < 0$
C			
D			
B			



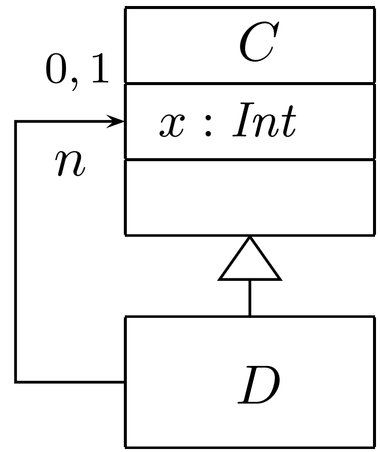
Satisfying OCL Constraints (Domain Inclusion)

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say $\mathcal{M} \models \mathit{expr}$ for $\underbrace{\text{context } C \text{ inv : } \mathit{expr}_0}_{=\mathit{expr}} \in \mathit{Inv}(\mathcal{M})$ iff

$$\forall\, \pi = (\sigma_i, \varepsilon_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!] \quad \forall\, i \in \mathbb{N} \quad \forall\, u \in \mathrm{dom}(\sigma_i) \cap \mathscr{D}(C) :$$
$$I[\![\mathit{expr}_0]\!](\sigma_i, \{\mathit{self} \mapsto u\}) = 1.$$

- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\mathit{Inv}(\mathcal{M})$.

- **Example**:



Transformers (Domain Inclusion)

- Transformers also remain **the same**, e.g. [VL 12, p. 18]

$$update(expr_1, v, expr_2) : (\sigma, \varepsilon) \mapsto (\sigma', \varepsilon)$$

with

$$\sigma' = \sigma[u \mapsto \sigma(u)[v \mapsto I[\![expr_2]\!](\sigma)]]$$

where $u = I[\![expr_1]\!](\sigma)$.

Semantics of Method Calls

- **Non late-binding**: clear, by normalisation.
- **Late-binding**:
  Construct a **method call** transformer, which is applied to all method calls.

Inheritance and State Machines: Triggers

- **Wanted**: triggers shall also be sensitive for inherited events,
  sub-class shall execute super-class' state-machine (unless overridden).

$$(\sigma, \varepsilon) \xrightarrow[u]{(cons, Snd)} (\sigma', \varepsilon') \text{ if}$$

- $\exists\, u \in \mathrm{dom}(\sigma) \cap \mathscr{D}(C)\ \exists\, u_E \in \mathscr{D}(\mathscr{E}) : u_E \in ready(\varepsilon, u)$
- u is stable and in state machine state s, i.e. $\sigma(u)(stable) = 1$ and $\sigma(u)(st) = s$,
- a transition is enabled, i.e.

$$\exists\,(s, F, expr, act, s') \in \rightarrow (\mathcal{SM}_C) : F = E \wedge I[\![expr]\!](\tilde{\sigma}) = 1$$

  where $\tilde{\sigma} = \sigma[u.params_E \mapsto u_e]$.

and

- (σ', ε') results from applying t_{act} to (σ, ε) and removing u_E from the ether, i.e.

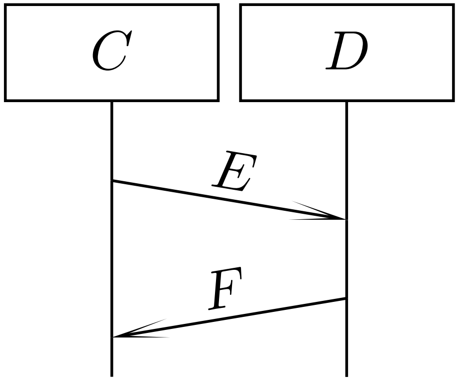
$$(\sigma'', \varepsilon') = t_{act}(\tilde{\sigma}, \varepsilon \ominus u_E),$$

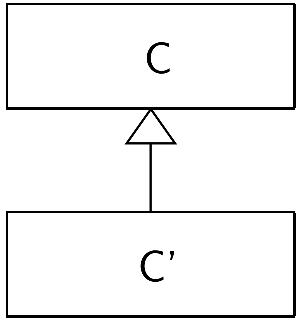
$$\sigma' = (\sigma''[u.st \mapsto s', u.stable \mapsto b, u.params_E \mapsto \emptyset])|_{\mathscr{D}(\mathscr{C}) \setminus \{u_E\}}$$

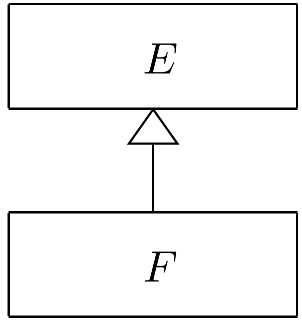
  where b **depends**:
  - If u becomes stable in s', then $b = 1$. It **does** become stable if and only if there is no transition **without trigger** enabled for u in (σ', ε').
  - Otherwise $b = 0$.
- Consumption of u_E and the side effects of the action are observed, i.e.

$$cons = \{(u, (E, \sigma(u_E)))\}, Snd = Obs_{t_{act}}(\tilde{\sigma}, \varepsilon \ominus u_E).$$

Domain Inclusion and Interactions







- Similar to satisfaction of OCL expressions above:
  - An instance line stands for all instances of C (exact or inheriting).
  - Satisfaction of event observation has to take inheritance into account, too, so we have to **fix**, e.g.

    $$\sigma, cons, Snd \models_\beta E^!_{x,y}$$

    if and only if

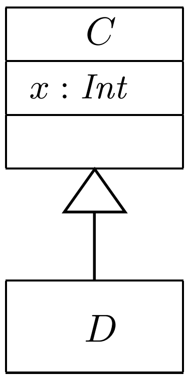
    $$\beta(x) \text{ sends an } F\text{-event to } \beta y \text{ where } E \preceq F.$$

  - **Note**: C-instance line also binds to C'-objects.

Uplink Semantics

Uplink Semantics

- **Idea**:
  - Continue with the existing definition of **structure**, i.e. disjoint domains for identities.
  - Have an **implicit association** from the child to each parent part (similar to the implicit attribute for stability).



  - Apply (a different) pre-processing to make appropriate use of that association, e.g. rewrite (C++)

$$\mathtt{x} = 0;$$

in D to

$$\mathtt{uplink}_C\ \mathtt{->}\ \mathtt{x} = 0;$$

Pre-Processing for the Uplink Semantics

- For each pair $C \lhd D$, extend D by a (fresh) association

$$uplink_C : C \text{ with } \mu = [1,1],\ \xi = +$$

  (**Exercise**: public necessary?)

- Given expression v (or f) in the **context** of class D,
  - let C be the **smallest** class wrt. "$\preceq$" such that
    - $C \preceq D$, and
    - $C{::}v \in atr(D)$
  - then there exists (by definition) $C \lhd C_1 \lhd \ldots \lhd C_n \lhd D$,
  - **normalise** v to ($=$ replace by)

$$uplink_{C_n} \texttt{->} \cdots \texttt{->}\, uplink_{C_1}.C{::}v$$

- Again: if no (unique) smallest class exists,
  the model is considered **not well-formed**; the expression is ambiguous.

Uplink Structure, System State, Typing

- Definition of structure remains **unchanged**.
- Definition of system state remains **unchanged**.
- Typing and transformers remain **unchanged** — the preprocessing has put everything in shape.

Satisfying OCL Constraints (Uplink)

- Let $\mathcal{M} = (\mathscr{CD}, \mathscr{OD}, \mathscr{SM}, \mathscr{I})$ be a UML model, and $\mathscr{D}$ a structure.
- We (**continue to**) say
$$\mathcal{M} \models expr$$
for
$$\underbrace{\text{context } C \text{ inv} : expr_0}_{=expr} \in \mathit{Inv}(\mathcal{M})$$
if and only if
$$\begin{aligned} &\forall\, \pi = (\sigma_i)_{i \in \mathbb{N}} \in [\![\mathcal{M}]\!] \\ &\quad \forall\, i \in \mathbb{N} \\ &\qquad \forall\, u \in \operatorname{dom}(\sigma_i) \cap \mathscr{D}(C) : \\ &\qquad\quad I[\![expr_0]\!](\sigma_i, \{self \mapsto u\}) = 1. \end{aligned}$$
- $\mathcal{M}$ is (still) consistent if and only if it satisfies all constraints in $\mathit{Inv}(\mathcal{M})$.

Transformers (Uplink)

- What **has to change** is the **create** transformer:

$$create(C, expr, v)$$

- Assume, C's inheritance relations are as follows.

$$C_{1,1} \lhd \ldots \lhd C_{1,n_1} \lhd C,$$
$$\ldots$$
$$C_{m,1} \lhd \ldots \lhd C_{m,n_m} \lhd C.$$

- Then, we have to
  - create one fresh object for each part, e.g.

$$u_{1,1}, \ldots, u_{1,n_1}, \ldots, u_{m,1}, \ldots, u_{m,n_m},$$

  - set up the uplinks recursively, e.g.

$$\sigma(u_{1,2})(uplink_{C_{1,1}}) = u_{1,1}.$$

  - And, if we had constructors, be careful with their order.

Late Binding (Uplink)

- Employ something similar to the "`mostspec`" trick (in a minute!). But the result is typically far from concise.

  (Related to OCL's `isKindOf()` function, and RTTI in C++.)

Domain Inclusion vs. Uplink Semantics

Cast-Transformers

  - `C c;`
  - `D d;`
- **Identity upcast** (C++):
  - `C* cp = &d;` // *assign address of 'd' to pointer 'cp'*
- **Identity downcast** (C++):
  - `D* dp = (D*)cp;` // *assign address of 'd' to pointer 'dp'*
- **Value upcast** (C++):
  - `*c = *d;` // *copy attribute values of 'd' into 'c', or,*
    // *more precise, the values of the C-part of 'd'*

Casts in Domain Inclusion and Uplink Semantics

	Domain Inclusion	**Uplink**
`C* cp = &d;`	**easy**: immediately compatible (in underlying system state) because `&d` yields an identity from $\mathscr{D}(D) \subset \mathscr{D}(C)$.	**easy**: By pre-processing, `C* cp =` $\texttt{d.uplink}_C$`;`
`D* dp = (D*)cp;`	**easy**: the value of cp is in $\mathscr{D}(D) \cap \mathscr{D}(C)$ because the pointed-to object is a D. Otherwise, error condition.	**difficult**: we need the identity of the D whose C-slice is denoted by cp. (See next slide.)
`c = d;`	**bit difficult**: set (for all $C \preceq D$) $(C)(\,\cdot\,,\,\cdot\,) : \tau_D \times \Sigma \to \Sigma\|_{atr(C)}$ $(u, \sigma) \mapsto \sigma(u)\|_{atr(C)}$ Note: $\sigma' = \sigma[u_C \mapsto \sigma(u_D)]$ is not type-compatible!	**easy**: By pre-processing, `c = *(`$\texttt{d.uplink}_C$`);`

Identity Downcast with Uplink Semantics

- **Recall** (C++): `D d;` `C* cp = &d;` `D* dp = (D*)cp;`
- **Problem**: we need the identity of the D whose C-slice is denoted by cp.
- **One technical solution**:
  - Give up disjointness of domains for **one additional type** comprising all identities, i.e. have

$$\mathtt{all} \in \mathscr{T}, \qquad \mathscr{D}(\mathtt{all}) = \bigcup_{C \in \mathscr{C}} \mathscr{D}(C)$$

  - In each $\preceq$-**minimal class** have associations "`mostspec`" pointing to **most specialised** slices, plus information of which type that slice is.
  - Then **downcast** means, depending on the `mostspec` type (only finitely many possibilities), **going down and then up** as necessary, e.g.

$$
\begin{array}{l}
\mathtt{switch(mostspec\_type)\{} \\
\quad \mathtt{case}\ C : \\
\qquad \mathtt{dp = cp \text{->} mostspec \text{->} uplink}_{D_n}\ \mathtt{\text{->}} \dots \mathtt{\text{->} uplink}_{D_1}\ \mathtt{\text{->} uplink}_{D}; \\
\dots \\
\}
\end{array}
$$

Domain Inclusion vs. Uplink Semantics: Differences

- **Note**: The uplink semantics views inheritance as an abbreviation:
  - We only need to touch transformers (create) — and if we had constructors, we didn't even needed that (we could encode the recursive construction of the upper slices by a transformation of the existing constructors.)
- **So**:
  - Inheritance **doesn't add** expressive power.
  - And it also **doesn't improve** conciseness **soo dramatically**.

As long as we're "**early binding**", that is...

Domain Inclusion vs. Uplink Semantics: Motives

- **Exercise**:

  What's the point of

  - having the **tedious** adjustments of the **theory**
    if it can be approached **technically**?
  - having the **tedious** technical **pre-processing**
    if it can be approached **cleanly** in the **theory**?

References

References

[Buschermöhle and Oelerink, 2008] Buschermöhle, R. and Oelerink, J. (2008). Rich meta object facility. In Proc. 1st IEEE Int'l workshop UML and Formal Methods.

[Fischer and Wehrheim, 2000] Fischer, C. and Wehrheim, H. (2000). Behavioural subtyping relations for object-oriented formalisms. In Rus, T., editor, AMAST, number 1816 in Lecture Notes in Computer Science. Springer-Verlag.

[Liskov, 1988] Liskov, B. (1988). Data abstraction and hierarchy. SIGPLAN Not., 23(5):17–34.

[Liskov and Wing, 1994] Liskov, B. H. and Wing, J. M. (1994). A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS), 16(6):1811–1841.

[OMG, 2003] OMG (2003). Uml 2.0 proposal of the 2U group, version 0.2, `http://www.2uworks.org/uml2submission`.

[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.

[Stahl and Völter, 2005] Stahl, T. and Völter, M. (2005). Modellgetriebene Softwareentwicklung. dpunkt.verlag, Heidelberg.