Computer Supported Modeling and Reasoning

David Basin, Achim D. Brucker, Jan-Georg Smaus, and Burkhart Wolff

April 2005

http://www.infsec.ethz.ch/education/permanent/csmr/

Higher-Order Logic Applications: HOL-OCL

Achim D. Brucker

Overview

- Motivation
- An Introduction to UML/OCL
- Formalizing Class Diagrams
- Excursus: Defining Semantics
- Embedding OCL into Isabelle/HOL
- Conclusion

Motivation

The Situation Today: A Software Engineering Problem

- Software systems are
  - getting more and more complex.
  - used in safety and security critical applications.
- We think:
  - Complex software systems require a precise specification of its architecture and components.
  - Semi-formal methods (like UML diagrams) are not strong enough.

Specification should be useful, i.e. not only documentation!

Why use Formal Methods in Software Development

There are many reasons for using formal methods:

- safety critical applications, e.g. flight or railway control.
- security critical applications, e.g. access control.
- financial reasons (e.g. warranty), e.g. embedded devices.
- legal reasons, e.g. certifications.

Many successful applications of formal methods proof their success!

Why Formal Methods are not widely accepted in software industry?

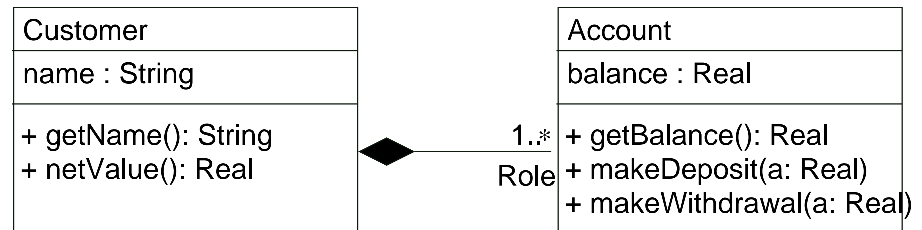
- Only a few formal methods address industrial needs:
  - support for object-oriented modeling and programming.
  - mainly automatic (?).
  - integration in standard CASE tools and processes.
- Formal methods people and industrial software developer are often speaking different languages.

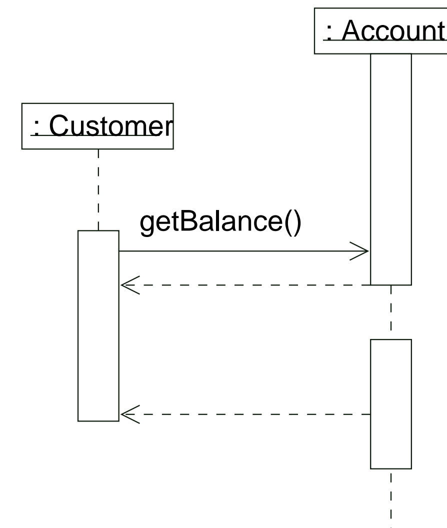
To tackle these challenges we provide a a formal foundation for (supporting object-orientation) for a industrial accepted specification languages (UML/OCL) [omg01, omg03].

An Introduction to UML/OCL

The Unified Modeling Language (UML)

- visual modeling language
- many diagram types, e.g.
  - class diagrams (static)
  - state charts (dynamic)
  - use cases
- object-oriented development
- industrial tool support
- OMG standard with semi-formal semantics





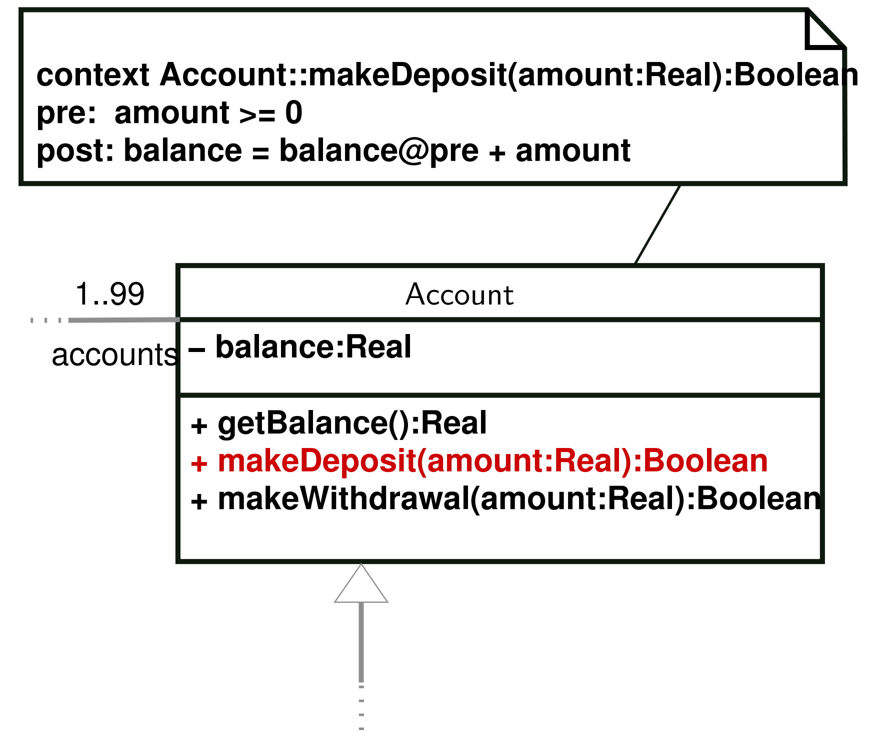
Are UML diagrams enough to specify OO systems formally?

- The short answer:
  - UML diagrams are not powerful enough for supporting formal reasoning over specifications.
- The long answer:

  We want to be able to
  - verify (proof) properties
  - refine specifications
- Thus we need:
  - a formal extension of UML.

The Object Constraint Language (OCL)

- based on first-order logic with equality and typed set theory
- designed for annotating UML diagrams
- in the context of class–diagrams:
  - preconditions
  - postconditions
  - invariants
- can be used for other diagrams too (not discussed here)



OCL — A Simple Examples

- "Uniqueness" constraint for the class Account:

```
context Account inv:
    Account.allInstances->forAll(a1,a2 | a1.id = a2.id
                                     implies a1 = a2)
```

- Properties of the class diagram can be described, e.g. multiplicities:

```
context Account inv: Account.owner->size = 1
```

- Meaning of the method makeDeposit():

```
context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
```

OCL keywords   Path expressions from UML model

Challenges of Formalizing UML/OCL

Only few formal methods are specialized for analyzing object oriented specifications.

- Problems and open questions:
  - object equality and aliasing
  - embedding of object structures into logics
  - referencing and dereferencing, including “null” references
  - dynamic binding
  - polymorphism
  - representing object-oriented concepts inside λ-calculi
  - providing a (suitable, shallow) representation in theorem provers
  - . . .

How to proceed

For Turning UML/OCL into a formal method we need

1. a formal foundation of UML class diagrams.
   - typed path expressions
   - inheritance
   - . . .
2. a formal semantics of OCL and proof support for OCL.
   - reasoning over UML path expressions
   - large libraries
   - . . .

Formalizing Class Diagrams

The Challenges of UML

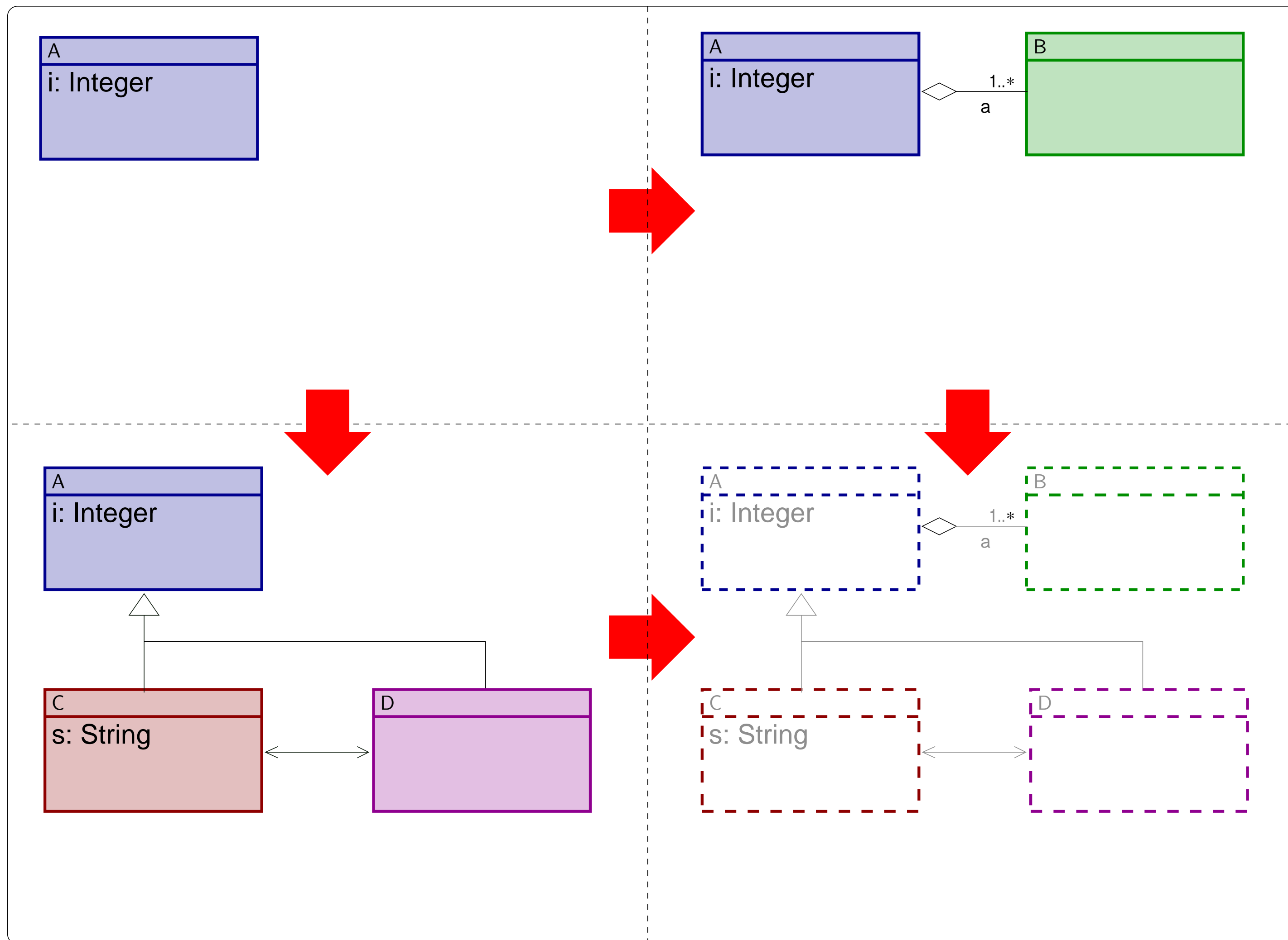
1. typed path expressions
2. typed object store
3. inheritance, thus extendibility in
   - data (by subtyping)
   - methods (late binding)
4. dynamic vs. static types
5. casting types

Typed Objects

Problem: • How to efficiently represent the types of objects?

What is the meaning of path expressions?

• How to represent the underlying state ?


A
i: Integer
A
i: Integer
B
1..*
a
A
i: Integer
C
s: String
D
A
i: Integer
B
1..*
a
C
s: String
D

Typed Objects

Problem: how to efficiently represent the types of objects

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)
3. its static type.
4. tests over the content of the extension field
   (constituting the dynamic type)

Typed Objects

Problem: how to efficiently represent the types of objects

Answer: Representing Class-Extensions by

1. a unique type key
2. a type extension
   (just the product of the type key and the attributes of this extension)
3. its static type.
4. tests over the content of the extension field
   (constituting the dynamic type)

5. projections and injections

5. projections and injections
6. develop mechanical support for injections, projections, and test theorems

5. projections and injections
6. develop mechanical support for injections, projections, and test theorems
7. generate UML path-syntax.

Typed Objects: Static Types

The static type of an object of a class is a product:

Typed Objects: Static Types

The static type of an object of a class is a product:

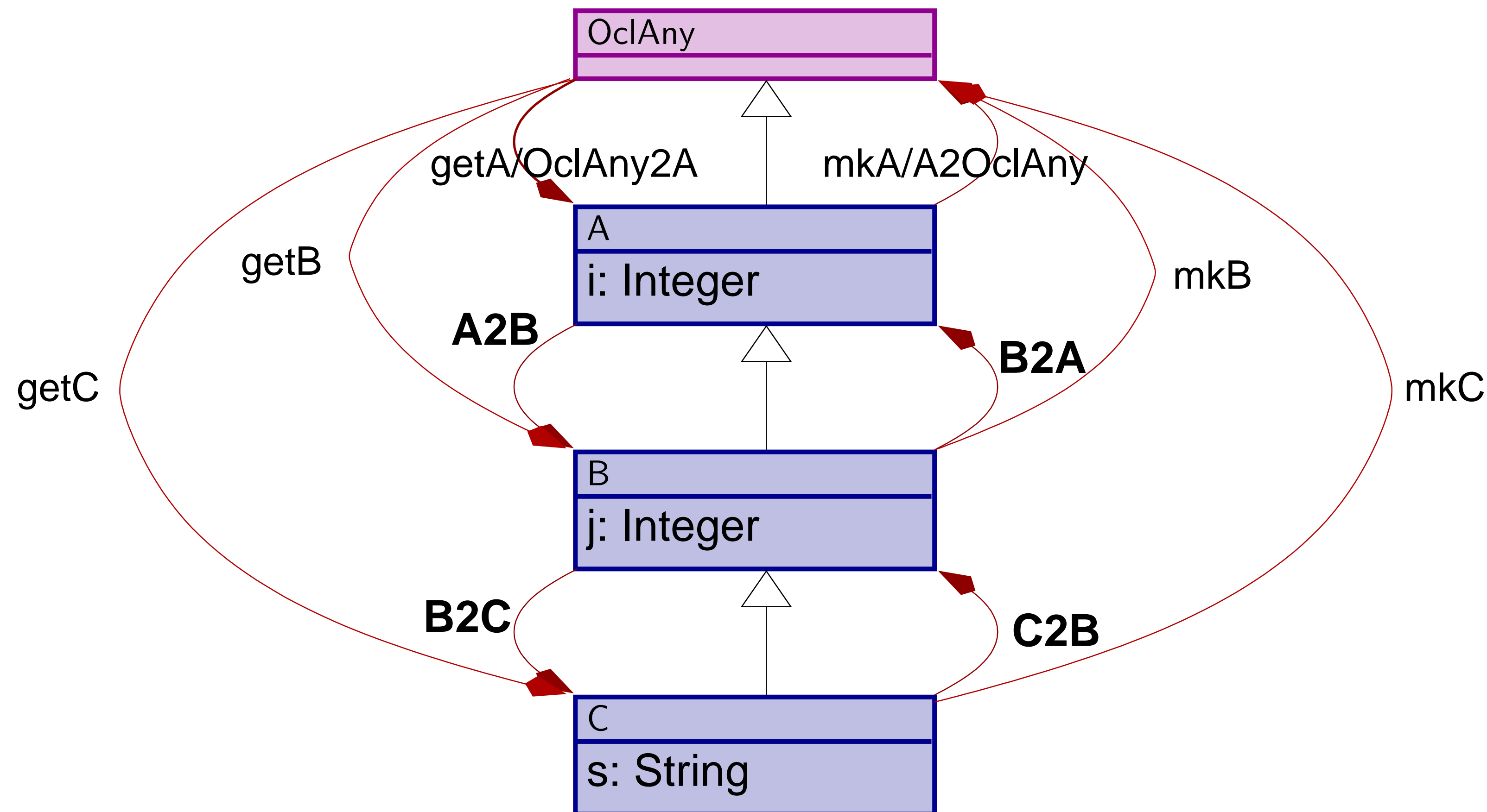
1. over all father class extensions (simplified)
   (Example A: $OclAny_key \times (A_key \times \alpha)$)

Typed Objects: Static Types

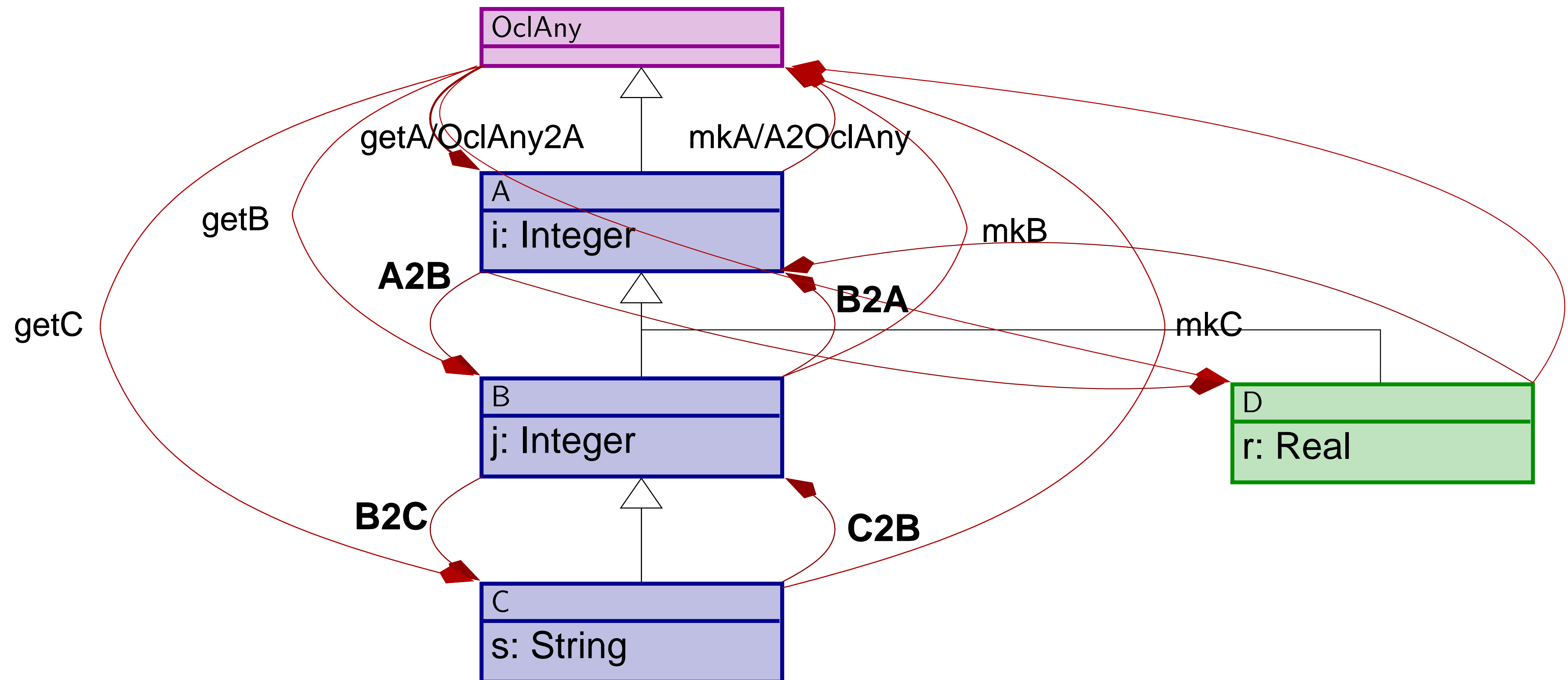
The static type of an object of a class is a product:

1. over all father class extensions (simplified)
   (Example A: $OclAny_key \times (A_key \times \alpha)$)
2. and an extension field α:
   (ranging over the sum of future extensions:
   $B_1 + \cdots + B_n + \beta$)

Typed Objects: Dynamic Types



Typed Objects: Dynamic Types



Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for $\circ$clAsType)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A (X), "castings" for ∘clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections (is_A(X), “castings” for $\circ$clAsType)
2. distinctness theorems (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems (all dynamic B of static A are dynamic A of static A . . .)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), “castings” for $\circ$ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems
   (all dynamic B of static A are dynamic A of static A . . .)
4. exhaustion theorems
   (A objects are void or have B extension or have D extension or other extension)

Automatically Derived Properties on Objects

1. dynamic tests, injections and projections
   (is_A(X), "castings" for ∘ clAsType)
2. distinctness theorems
   (dynamic B of static A is different from dynamic D of static A)
3. inclusion theorems
   (all dynamic B of static A are dynamic A of static A . . .)
4. exhaustion theorems
   (A objects are void or have B extension or have D extension or other extension)

5. invariance theorems

$(\ \mathrm{DEF}(\mathrm{A}::\mathrm{A}) \Longrightarrow \mathrm{INV_A})$

Open question: induction theorems.

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")
2. their is no need for "wellformedness"-predicates of objects

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: “open world”)
2. their is no need for “wellformedness”-predicates of objects
3. conversions between static types must be explicit
4. there is a uniform sum-type that comprises them all class extensions

Summary

In this representation,

1. for a static type, any dynamic extension will be typesafe accepted
   (by Isabelle/HOL; even if we do not know how many we have: "open world")
2. their is no need for "wellformedness"-predicates of objects
3. conversions between static types must be explicit
4. there is a uniform sum-type that comprises them all class extensions
5. proofs remain valid under extension

Excursus: Defining Semantics

Semantics

OCL Semantics

Semantics

OCL Semantics

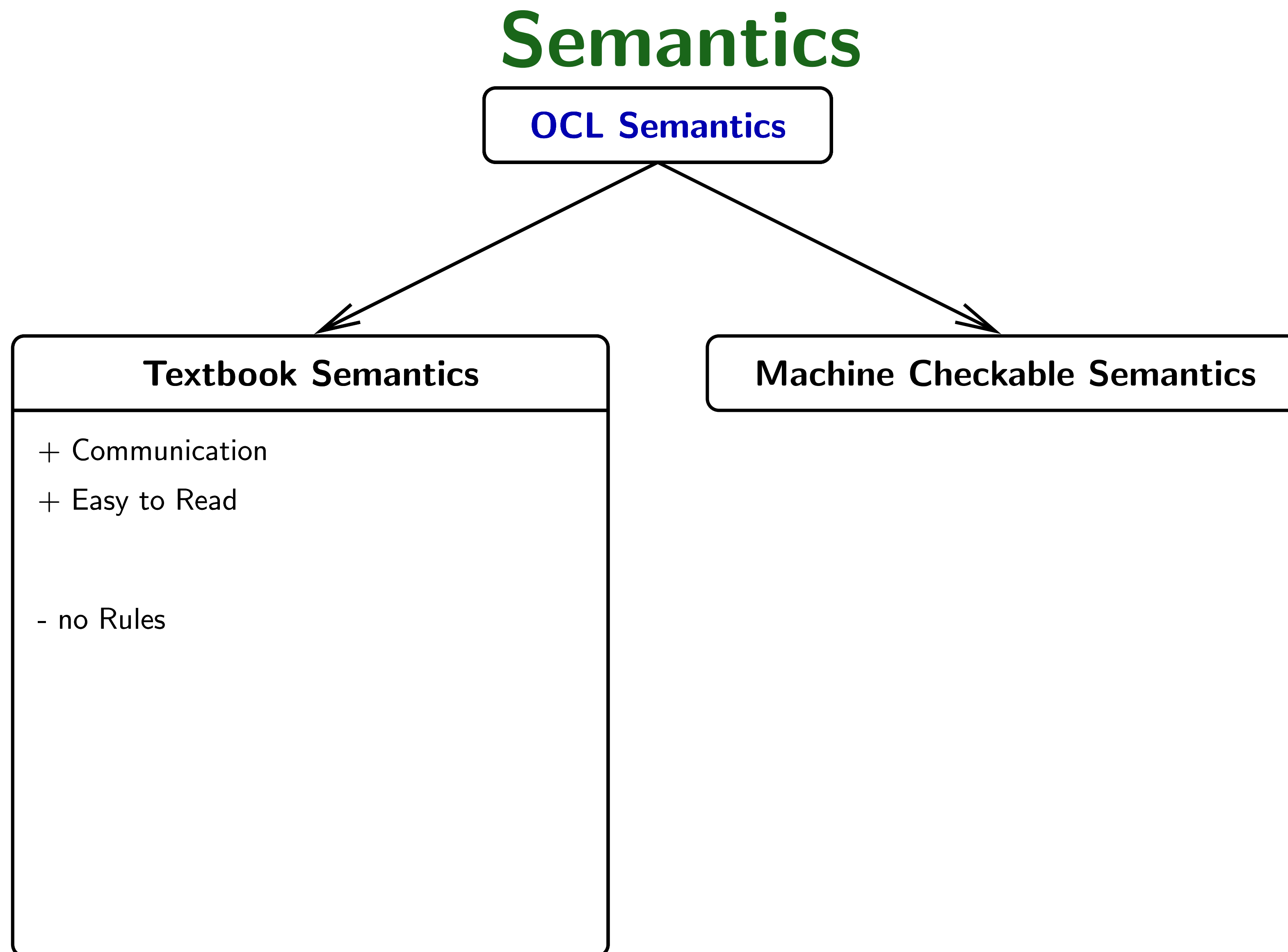
Textbook Semantics

+ Communication

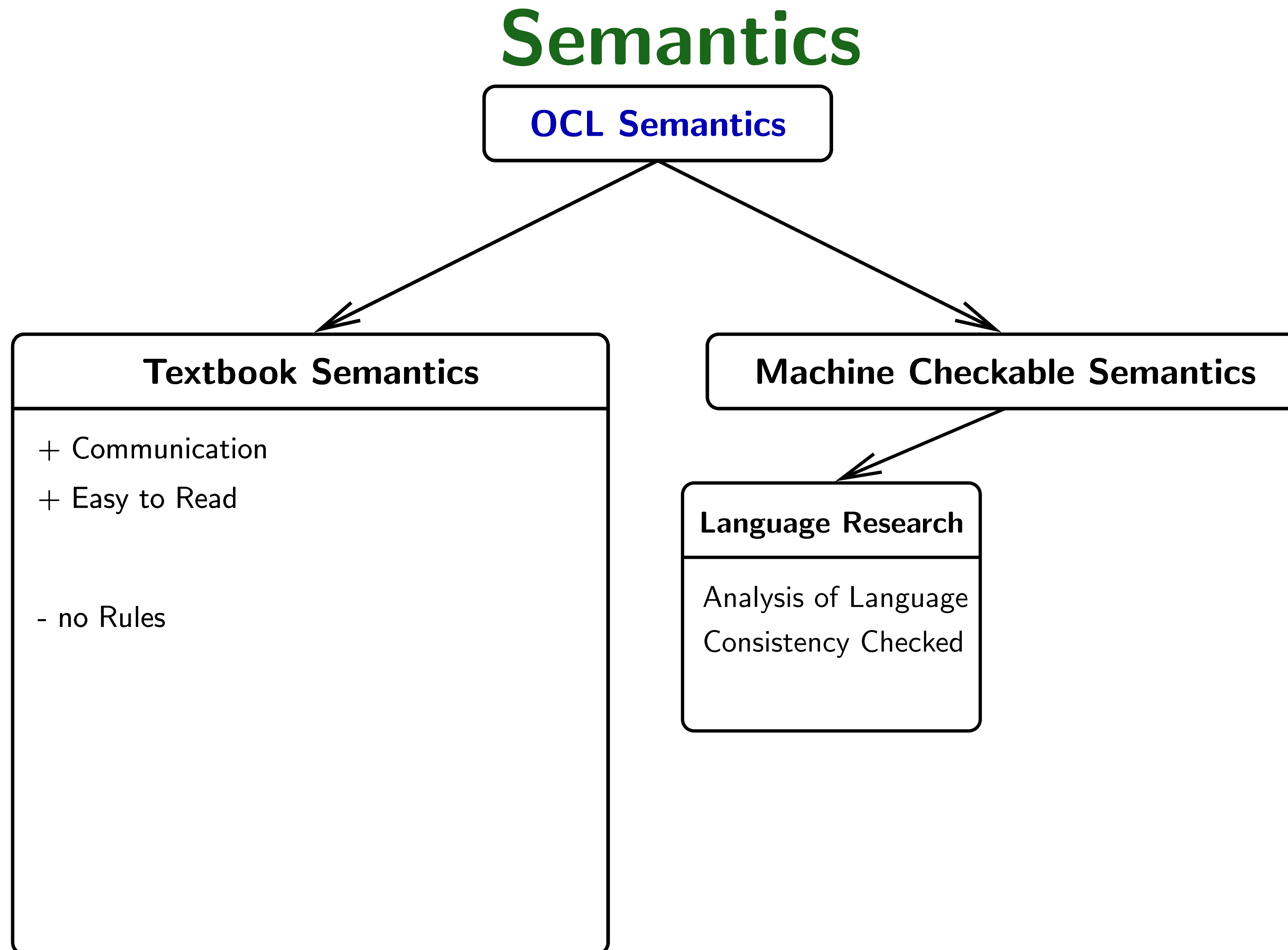
+ Easy to Read

- no Rules

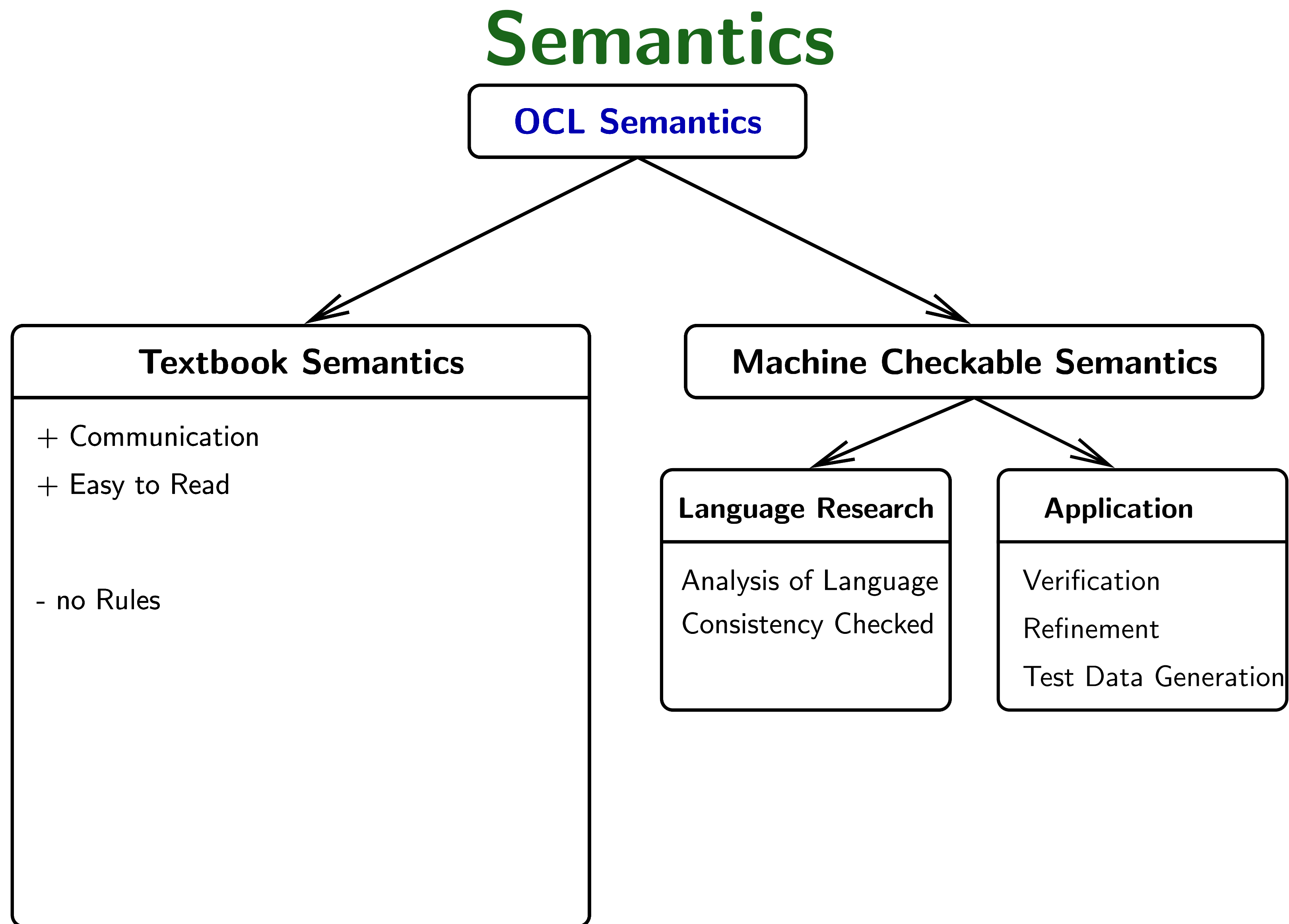
Semantics



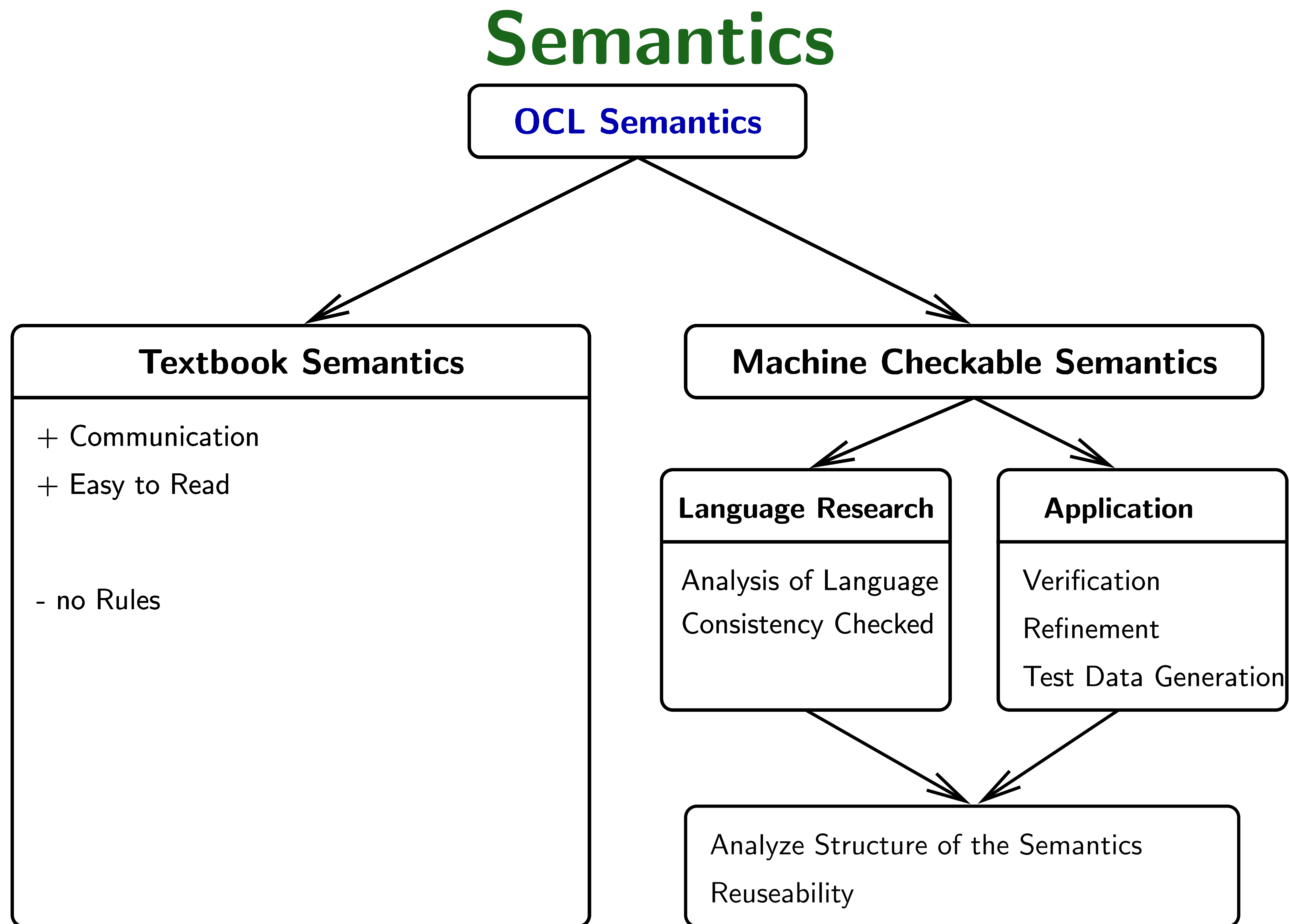
Semantics



Semantics



Semantics



Textbook Semantics: An Example

- The interpretation of the logical and is given by a truth-table:

a	b	a and b
$false$	$false$	$false$
$false$	$true$	$false$
$false$	$\bot_{\mathscr{L}}$	$false$

a	b	a and b
$true$	$false$	$false$
$true$	$true$	$true$
$true$	$\bot_{\mathscr{L}}$	$\bot_{\mathscr{L}}$

a	b	a and b
$\bot_{\mathscr{L}}$	$false$	$false$
$\bot_{\mathscr{L}}$	$true$	$\bot_{\mathscr{L}}$
$\bot_{\mathscr{L}}$	$\bot_{\mathscr{L}}$	$\bot_{\mathscr{L}}$

- The Interpretation of “X->union(Y)” for sets (“$X \cup Y$”):

$$I(\cup)(X,Y) \;\equiv\; \begin{cases} X \cup Y & \text{if } X \neq \bot_{\mathscr{L}} \text{ and } Y \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

This is a strict and lifted version of the union of “mathematical sets”.

Textbook Semantics

- "Paper-and-Pencil" work in mathematical notation.

+ Useful to communicate semantics.
+ Easy to read.

- No rules, no laws.
- Informal or meta-logic definitions
  (*"The Set is the mathematical set."*).
- It is easy to write inconsistent semantic definitions.

Machine-Checkable Semantics

Motivation: Honor the semantical structure of the language.

- A machine-checked semantics
  - conservative embeddings guarantee consistency of the semantics.
  - builds the basis for analyzing language features.
  - allows incremental changes of semantics.
- As basis of further tool support for
  - reasoning over specifications.
  - refinement of specifications.
  - automatic test data generation.

Shallow vs. Deep Embeddings

Representing the logical operations or and and via a

- shallow embedding:
  Direct definition of the semantics, e.g. each construct is represented by some function on a semantic domain.

$$x\,\mathsf{and}\,y \equiv \lambda e.\,.\,x\ e \wedge y\ e \qquad x\,or\,y \equiv \lambda e.\,.\,x\ e \vee y\ e$$

- deep embedding:
  The abstract syntax is presented as a datatype and a semantic function I from syntax to semantics.

$$expr = \text{var } var \mid expr \text{ and } expr \mid expr \text{ or } expr$$

and the explicit semantic function I:

$$\begin{aligned} I[\![\text{var } x]\!] &= \lambda e.\,.\,e(x) \\ I[\![x \text{and} y]\!] &= \lambda e.\,.\,I[\![x]\!]\ e \wedge I[\![y]\!]\ e \\ I[\![x or y]\!] &= \lambda e.\,.\,I[\![x]\!]\ e \vee I[\![y]\!]\ e \end{aligned}$$

Embedding OCL into Isabelle/HOL

The Challenges of OCL

1. OCL semantics evaluation-oriented (undefinedness $\perp$, strict evaluation)
2. large library in operational style
3. logics three-valued (Kleene-Logics) (i.e. $a \wedge true = a$ and $true \wedge a = a$ for $a \in \{true, false, \perp_{\mathscr{L}}\}$)
4. reasoning over UML path expressions

Defining the Library

- The OCL standard describes:

$$I(\cup)(X,Y) \equiv \begin{cases} X \cup Y & \text{if } X \neq \bot_{\mathscr{L}} \text{ and } Y \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- In Isabelle we define:

constdefs

```
 union :: "[('a,('b::bot) Set) VAL, ('a,'b Set) VAL]
                        ⇒ ('a,'b Set) VAL"
                                  ("_ ->union'( _ ')" [66,65]65)
  "union ≡ lift2 ( strictify (λX. strictify (λY. Abs_Set
          (⌊ Lifting.drop (Rep_Set X) ∪ Lifting.drop (Rep_Set Y)⌋)
     )))"
```

But is this faithful to the standard? (1/2)

Let's consider the not:

$$
\begin{aligned}
\text{Not X} &= \text{lift1 not' X} \\
&= \text{lift1 } \lfloor_\rfloor \circ \text{ strictify } (\neg \circ \lceil_\rceil) \text{ X} \\
&= \lambda\text{St. if DEF(X St)} \\
&\qquad\qquad \text{then } \lfloor \neg \lceil \text{X St} \rceil \rfloor \\
&\qquad\qquad \text{else } \bot
\end{aligned}
$$

But is this faithful to the standard? (1/2)

Let's consider the not:

$$
\begin{aligned}
\text{Not X} &= \text{lift1 not' X} \\
&= \text{lift1} \ \lfloor_\rfloor \circ \text{strictify} \ (\neg \circ \lceil_\rceil) \ \text{X} \\
&= \lambda \text{St. if DEF(X St)} \\
&\qquad\qquad \text{then } \lfloor \neg \lceil \text{X St} \rceil \rfloor \\
&\qquad\qquad \text{else } \bot
\end{aligned}
$$

By introducing the usual semantic function Sem (which happens to be the identity in a shallow embedding), we get:

$$
\begin{aligned}
\text{Sem}[\![\text{Not X}]\!] \ \text{St} = \ &\text{if DEF(Sem}[\![\text{X}]\!] \ \text{St)} \\
&\text{then } \lfloor \neg \lceil \text{Sem}[\![\text{X}]\!] \ \text{St} \rceil \rfloor \\
&\text{else } \bot
\end{aligned}
$$

But is this faithful to the standard? (2/2)

- Which looks already like the definition in the standard

$$I(\neg)(X) \equiv \begin{cases} \neg X & \text{if } X \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- Further, we prove (beeing conservative) all the usual laws (idempotency, associativity, . . .) for all defined operators.
- But this is a lot of work . . .

But is this faithful to the standard? (2/2)

- Which looks already like the definition in the standard

$$I(\neg)(X) \equiv \begin{cases} \neg X & \text{if } X \neq \bot_{\mathscr{L}} \\ \bot_{\mathscr{L}} & \text{otherwise} \end{cases}$$

- Further, we prove (beeing conservative) all the usual laws (idempotency, associativity, . . .) for all defined operators.
- But this is a lot of work . . . Let's try to automate [BW03] this

Library Construction

Problem: Large library in operational style

Library Construction

Problem: Large library in operational style

Answer: Combinator Style Semantics building a Theory Morphism

Library Construction

Problem: Large library in operational style

Answer: Combinator Style Semantics building a Theory

Morphism

Representing Phases of the Translation

by combinators on

1. types (type constructors)
2. terms (higher-order functions)

once and for all!

Library Construction

Problem: Large library in operational style

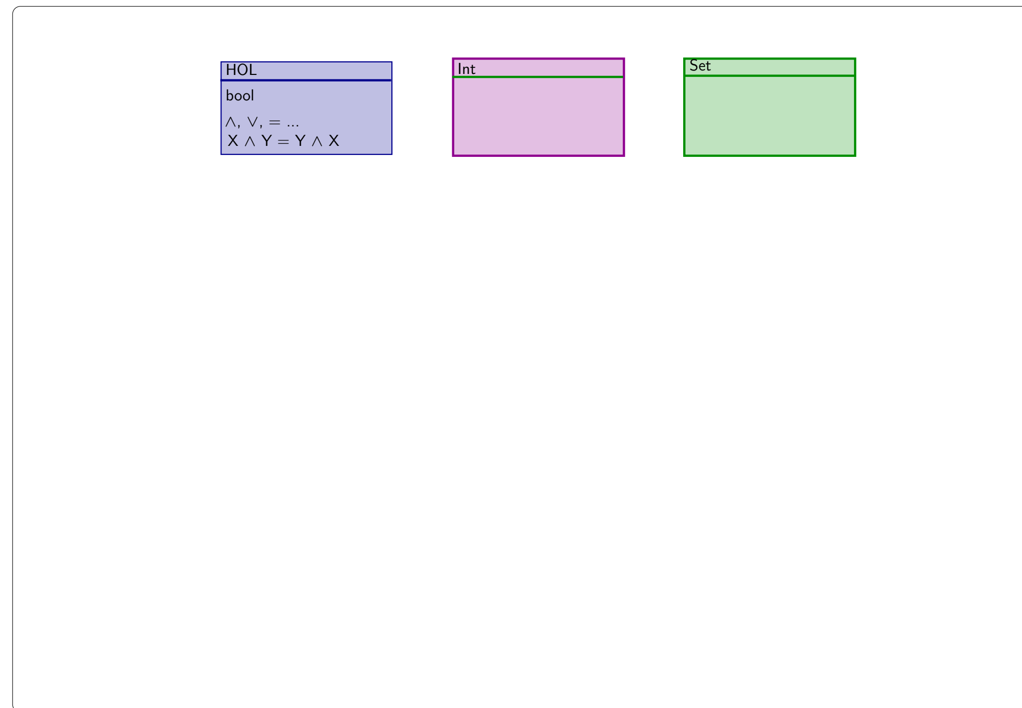
Answer: Combinator Style Semantics building a Theory
Morphism
Representing Phases of the Translation
by combinators on

1. types (type constructors)
2. terms (higher-order functions)

once and for all!
Develop mechanical support for "Lifting Theorems".

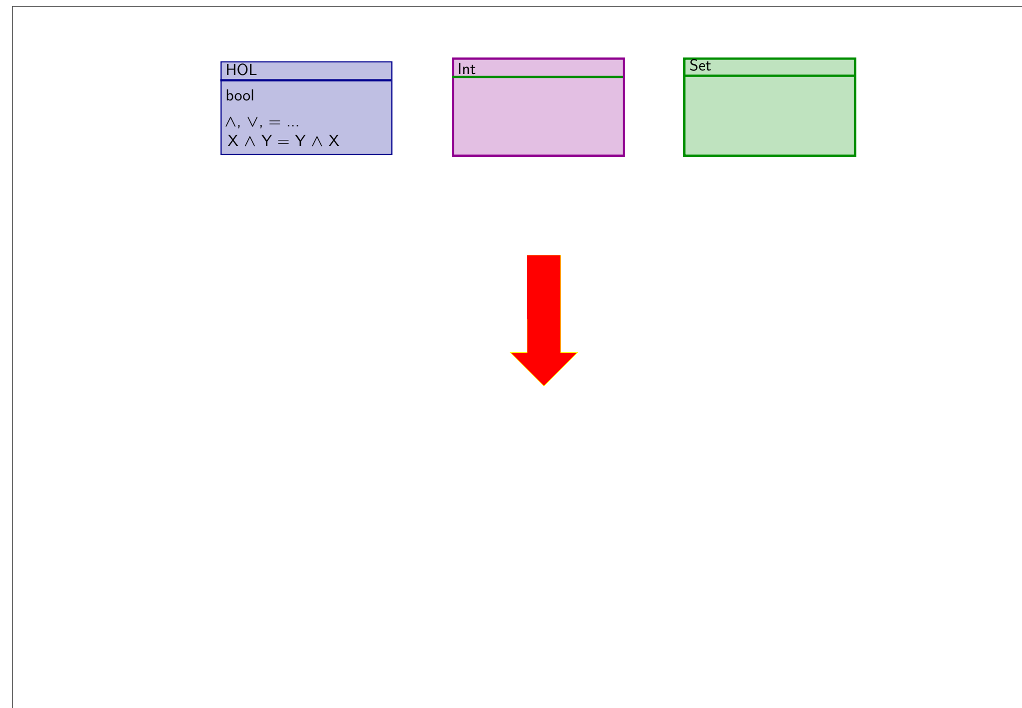
Observation:

90 percent of the embedding is a canonical theory morphism!



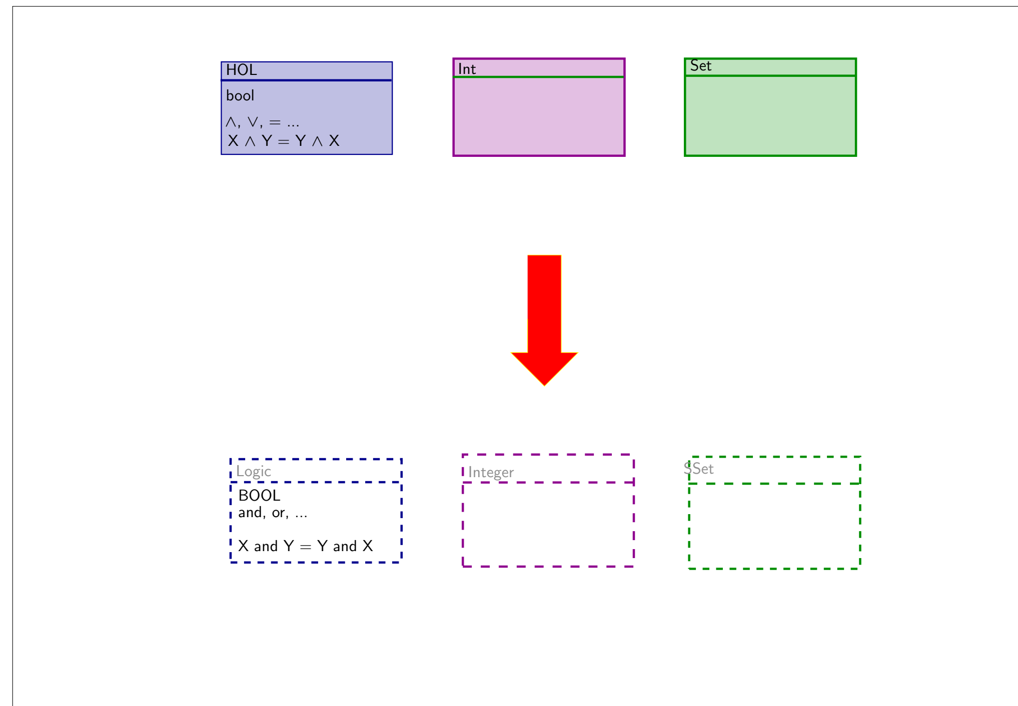
Observation:

90 percent of the embedding is a canonical theory morphism!



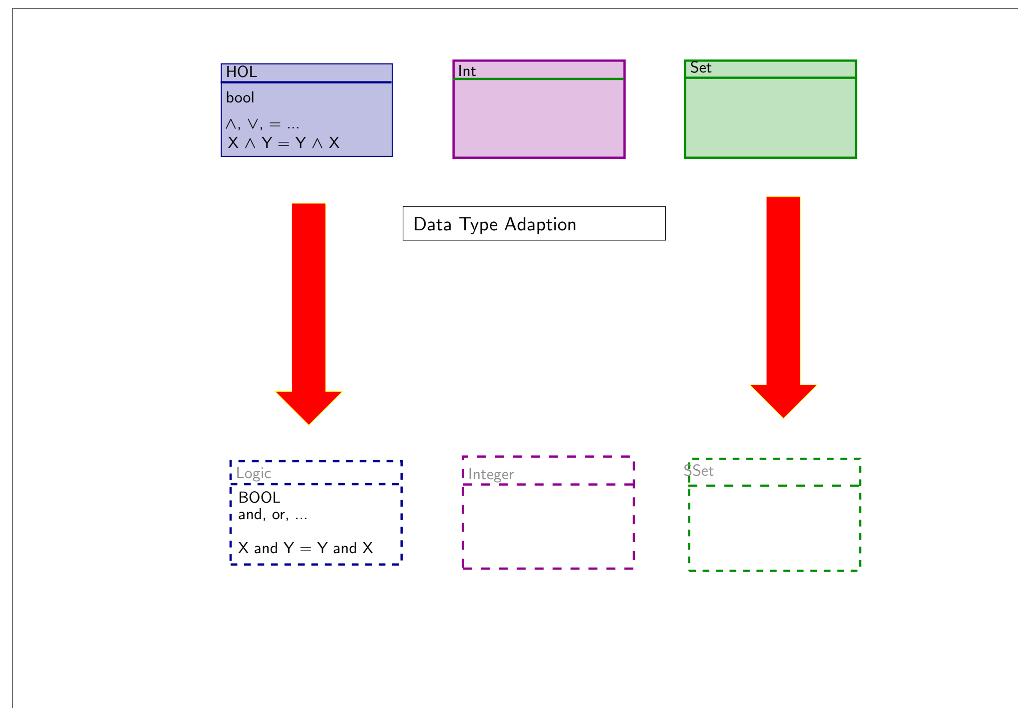
Observation:

90 percent of the embedding is a canonical theory morphism!



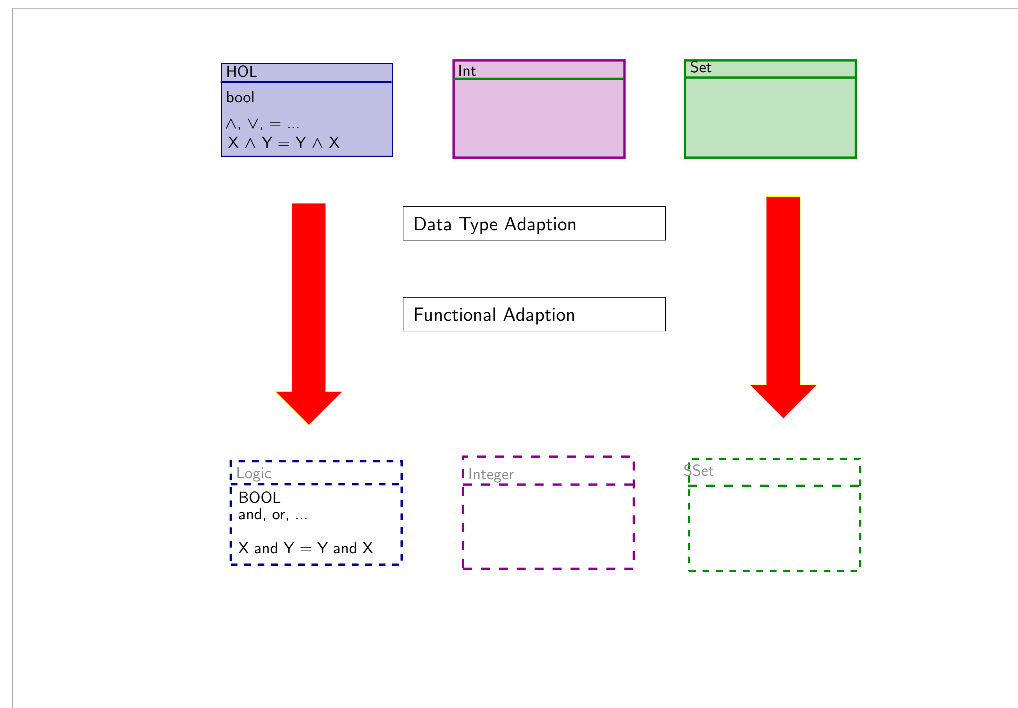
Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



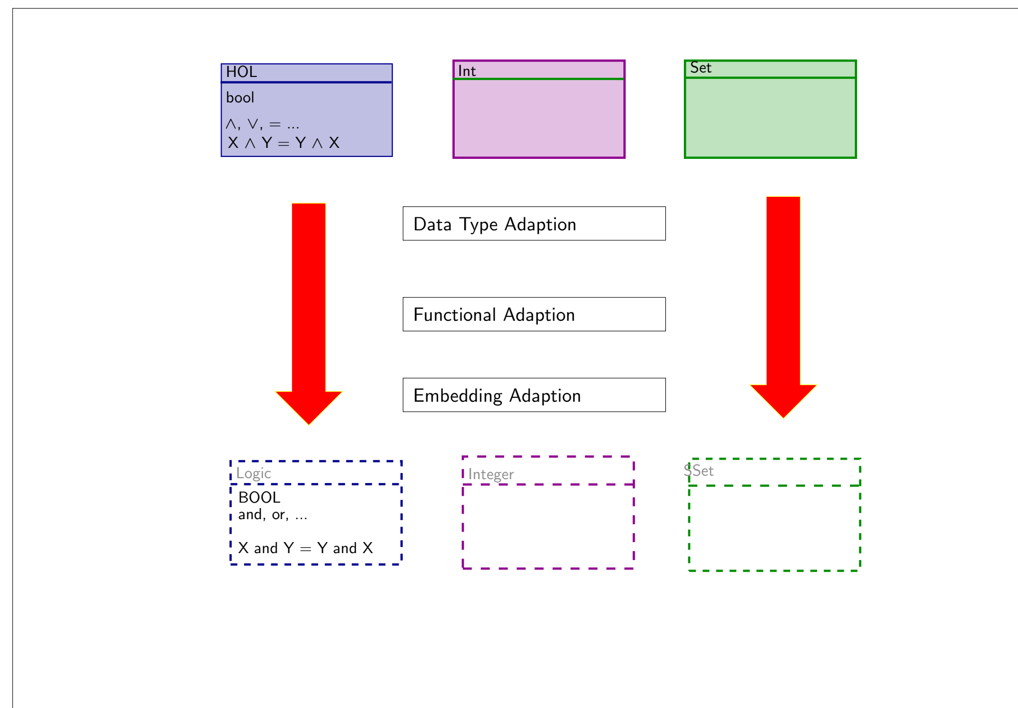
Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



Overall Idea:

Organize the morphism into phases represented by "semantic combinators"!



Advantages of the Combinator Representation

- generic theorems for strictness and definedness lead to tactics that construct specialized theorems for each operation in the theory morphism, e.g.: Example

  $\mathrm{DEF}(\text{strictify}\ \ f\ \ X) \Longrightarrow \mathrm{DEF}\ f \wedge \mathrm{DEF}\ X$

- generic "lifting theorems" for commutativity . . . allowing to "lift" these properties automatically . . .
  Example

  $$(\bigwedge x\ y.\ f\ x\ y = f\ y\ x) \Longrightarrow$$
  $$(\ \text{strictify}\ \ (\lambda x.\ \ \text{strictify}\ (f\ x)))\ X\ Y$$
  $$=(\ \text{strictify}\ \ (\lambda x.\ \ \text{strictify}\ (f\ x))\ Y\ X)$$

Do we honour the standard?

- The “high-level meaning” of the operators is given as pre-/post-condition pairs, e.g.:

```
context: Collection.count(object:T):integer
post: result = self→iterate(elem;
                                    acc: integer = 0
                                    | if elem = object then acc
```

- These can be proven in HOL-OCL:

```
lemma "((self::('a, 'b::bot Set)VAL)->count((obj::('a, 'b)VAL))) =
                      (if  self->includes(obj)
                       then 1 else 0 endif)"
```

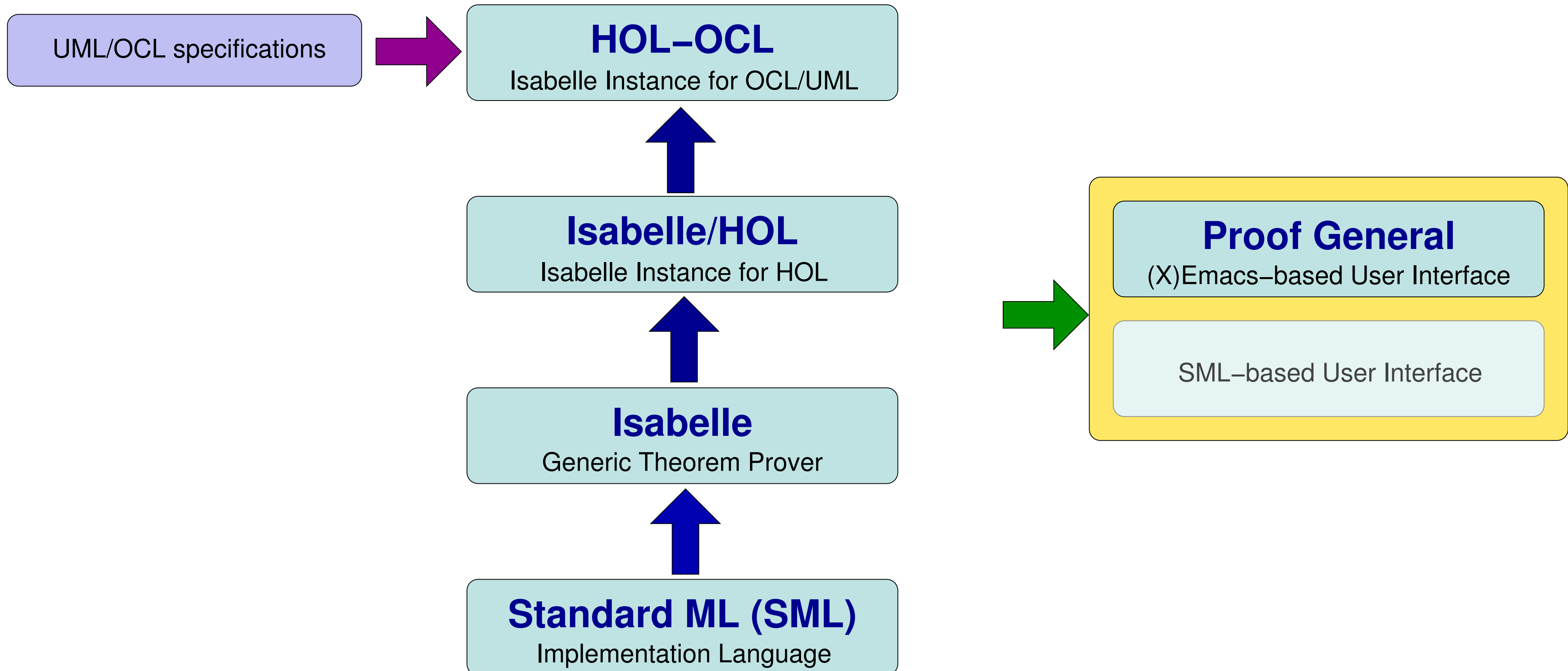
```
lemma "((self::('a, 'b::bot Set)VAL)->count((obj::('a, 'b)VAL))) =
                        (if self->includes(obj)
                         then 1 else 0 endif)"
apply (rule ext)
apply (simp add: count_def includes_def ocl_if_def weak_eq_def
                 OCL_Integer.Zero_ocl_int_def
                 OCL_Integer.One_ocl_int_def)
apply (simp only: lift0_def lift1_def lift2_def lift3_def
                  strictify_def o_def not_def oclIsDefined_def T
                 RUE_def)
apply (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def
               split add: split_if up.split)
done
```

Summing up: The OCL-Level

- We have seen:
  - Generic Theorems for handling undefindness, lifting, etc.
  - Generic Support for Calculi
  - Automated “Lifting” of theorems from the HOL level to the OCL level
- Overall, HOL-OCL provides:
  - a consistend OCL semancics build as an conservative (shallow) embedding into Isabelle/HOL
  - proof support for OCL (several calculi)

Conclusion

The overall picture



Conclusion

- formal semantics of UML/OCL 2.0

Conclusion

- formal semantics of UML/OCL 2.0
- layered, “combinator-style” representation

Conclusion

- formal semantics of UML/OCL 2.0
- layered, “combinator-style” representation
- derived library

Conclusion

- formal semantics of UML/OCL 2.0
- layered, “combinator-style” representation
- derived library
- derived 3-valued logical calculi
  inlcuding machine-support

Conclusion

- formal semantics of UML/OCL 2.0
- layered, “combinator-style” representation
- derived library
- derived 3-valued logical calculi
  inlcuding machine-support
- extensible, typed object-state
  enabling subtyping on dynamic types

HOL-OCL: A Shallow Embedding of OCL into HOL

The HOL-OCL system [BW02a, BW02b]:

- is build on top of Isabelle/HOL.
- is a shallow embedding of OCL into HOL.
- provides a consistent (machine checked) OCL semantics.
- allows the examination of OCL features.
- builds the basis for OCL tool development.
- follows OCL 2.0
- over 2000 theorems (language properties) proven.

The Technical Design of HOL-OCL

- Reuseability:
  - Reuse old proofs for class diagrams constructed via inheritance introduction of new classes.
  - Extendible semantics approach.
- Representing semantics structurally:
  - Organize semantic definitions by certain combinators capturing the semantical essence (e.g. lifting and strictness).
  - Automatically construct theorems out of uniform definitions.

References

[BW02a] Achim D. Brucker and Burkhart Wolff. HOL-OCL: Experiences, consequences and design choices. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002: Model Engineering, Concepts and Tools*, number 2460, pages 196–211. Springer-Verlag, Dresden, 2002.

[BW02b] Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In César Muñoz, Sophiène Tahar, and Víctor Carreño, editors, *Theorem Proving in Higher Order Logics*, number 2410, pages 99–114. Springer-Verlag, Hampton, VA, USA, 2002.

[BW03] Achim D. Brucker and Burkhart Wolff. Using theory morphisms for implementing formal methods tools. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proof and Programs*, number 2646, pages 59–77. Springer-Verlag, Nijmegen, 2003.

[omg01] OMG Unified Modeling Language Specification, September 2001.

[omg03] Uml 2.0 ocl specification, October 2003.