Title	An Object-Oriented logic for software analysis and design
Author(s)	Yatake, Kenro; Aoki, Toshiaki; Katayama, Takuya
Citation	
Issue Date	2005-09-21
Type	Presentation
Text version	publisher
URL	http://hdl.handle.net/10119/8323
Rights	
Description	1st VERITE : JAIST/TRUST-AIST/CVS joint workshop on VERIfication TEchnologyでの発表資料, 開催：2005年9月21日～22日, 開催場所：金沢市文化ホール3F

An Object-Oriented logic for software analysis and design

Kenro Yatake, Toshiaki Aoki
Takuya Katayama

JAIST

2005.9.21

Background

- The Object-Oriented method has become the mainstream of software development.
- In the upstream phase of the development, analysis models are constructed with a language such as UML.
- To ensure that the system satisfies its requirements, formal verification method must be applied to the analysis models.

Verification target

- Invariant properties about object attributes. e.g.
  - AC: The thermometer value always less than 30.
  - Bank: Balance values never become negative.

Apply theorem proving

The HOL system

- The use of HOL theorem prover
  - Interactive prover of higher-order logic.
  - A lot of mathematical libraries.
  - No libraries which implement OO concepts.

Implement object theory in HOL

Existing object theories in higher-order logic

- Semantics of Java program verification
  - LOOP, Bali, Krakatoa, …
    - Types of object attributes are limited to primitive ones appearing in Java (integer, boolean, …).
  - Compared to program verification, analysis model verification requires high availability of types.
    - High abstract types (set, stack, tree …)
    - Domain-specific types (date, time, currency, …)

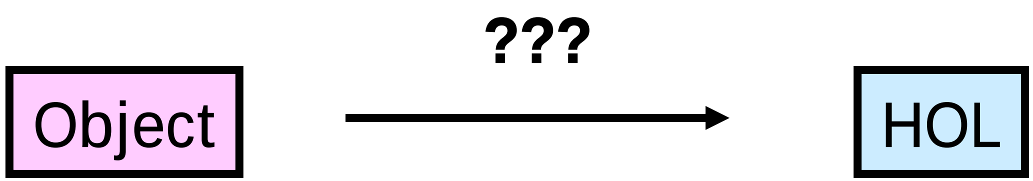
Objective

- Implement an object theory where object can have arbitrary types of attributes.

Verification with a wide variety of types becomes possible making use of plentiful mathematical libraries and powerful datatype definition facilities of HOL.

Embedding problem

- The type system of HOL is too simple to express object concepts as a general type.

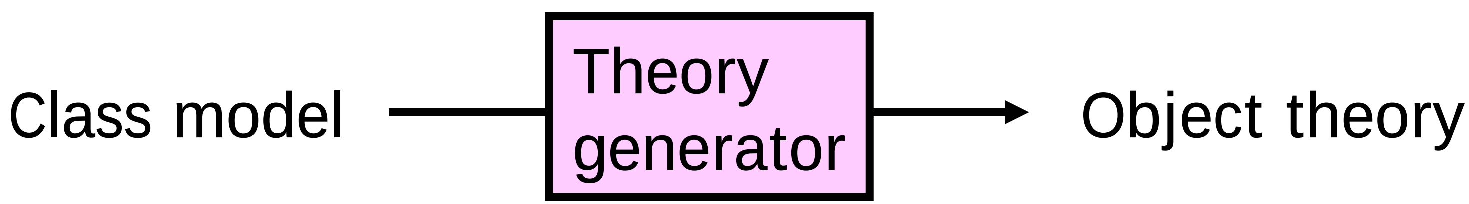


Unified type of arbitrary types
Subtyping
Referencing

Simply-typed lambda
calculus with a first-order
language of types

Approach

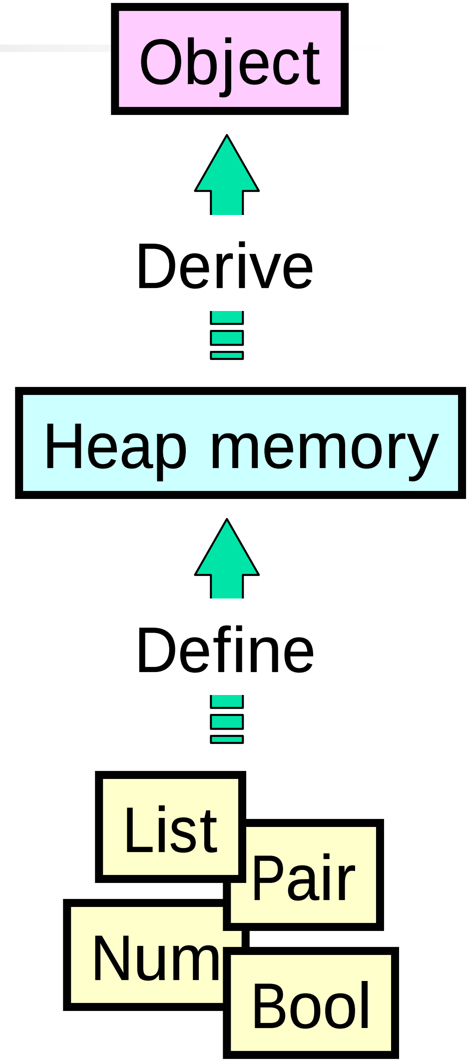
- **Application-specific theory**
  - Automatically construct object theories depending on the type information of individual applications.



⇨ In effect, arbitrary types can be incorporated into object attributes.

Soundness

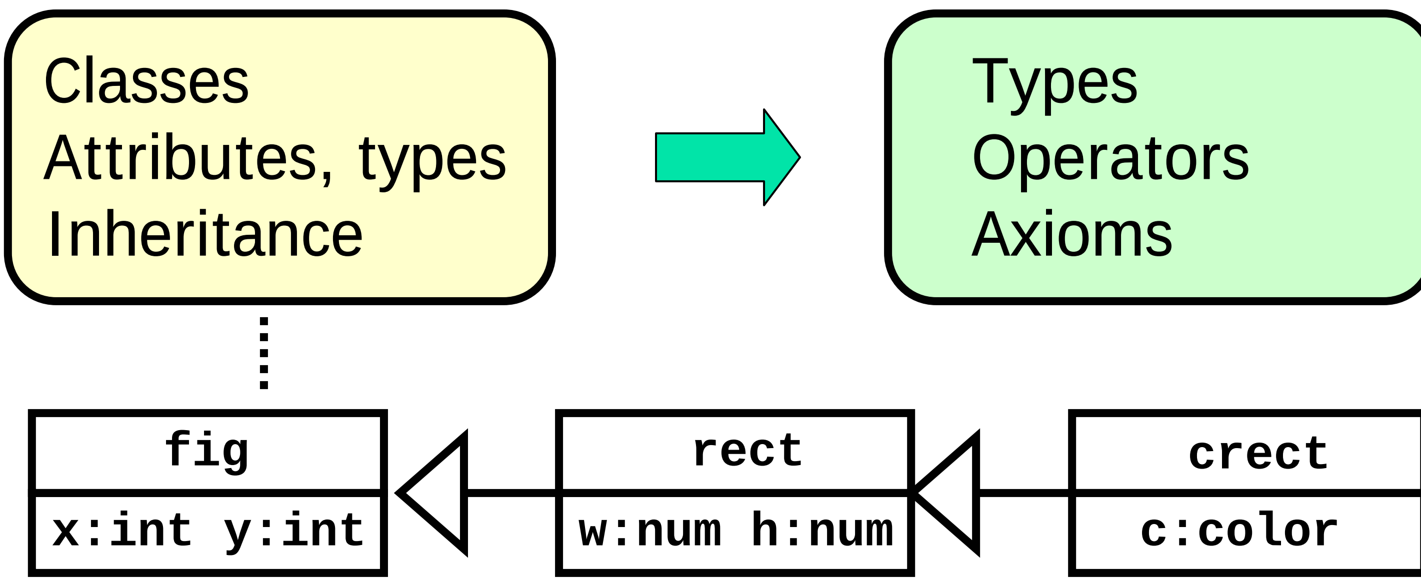
- Construct the theory by **definitional extension**
  - The standard technique to construct sound theories in HOL
  - Derive new theories from existing sound theories only by introduction of definitions and derivation by sound inference rules.



Overview of the theory

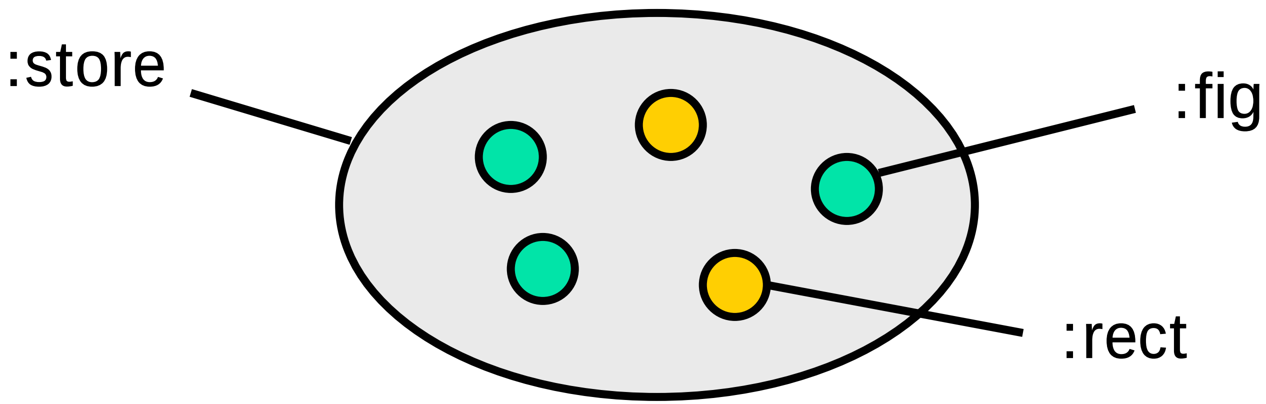
- The theory is defined by mapping the class model elements to theory elements

Class model HOL theory



Types

- Store
  - The environment of a system which holds a state of all alive objects.
- Object references
  - Object references are represented by types whose names are their class names.

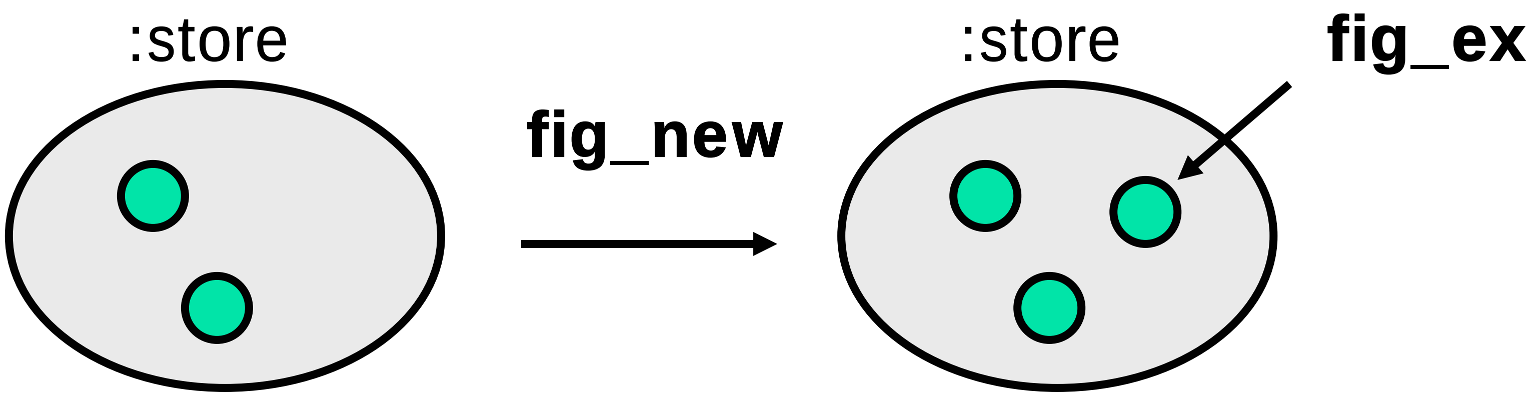


Operators & axioms

- 6 kinds of primitive operators to handle objects are defined on the store.
  - Object creation
  - Alive test
  - Attribute read
  - Attribute write
  - Casting
  - Instance-of
- 36 axioms are introduced on the operators.

Object creators & alive testers

- Object creators creates a new object in a store.
- Alive testers tests if an object exists in a store.

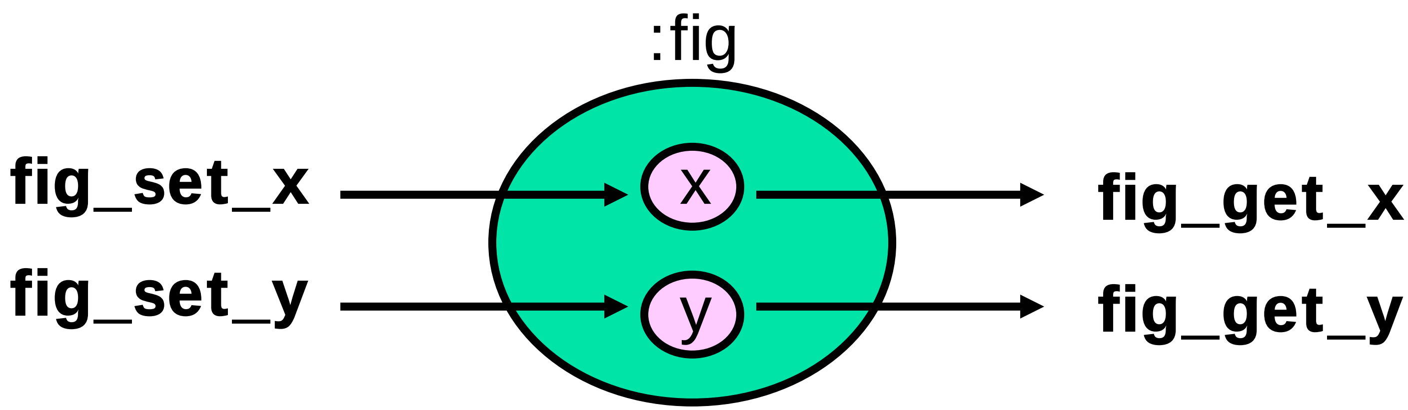


$$\boxed{\forall f\ s.\ \texttt{let}\ (f,s') = \texttt{fig\_new}\ s\ \texttt{in}\ \texttt{fig\_ex}\ f\ s'}$$

"The newly created object exists in the new store."

Attribute accessors

- Attribute accessors read and write the object attributes.

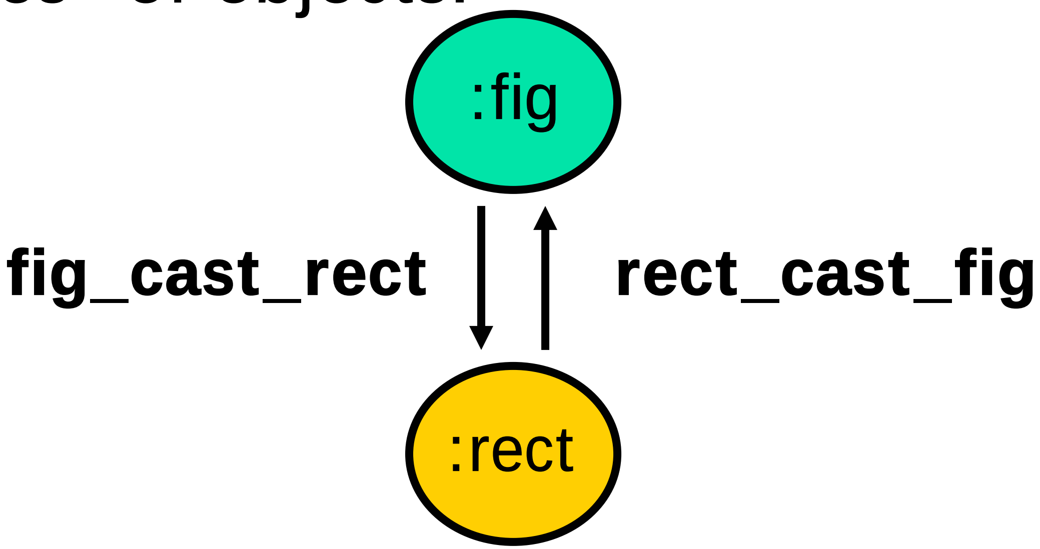



```
∀f x s. fig_ex f s ⇒
   (fig_get_x f (fig_set_x f A s) = A)
```

“The attribute x obtained just after updating it to A is equal to A.”

Casting functions

- Casting functions change the “apparent types” of objects.

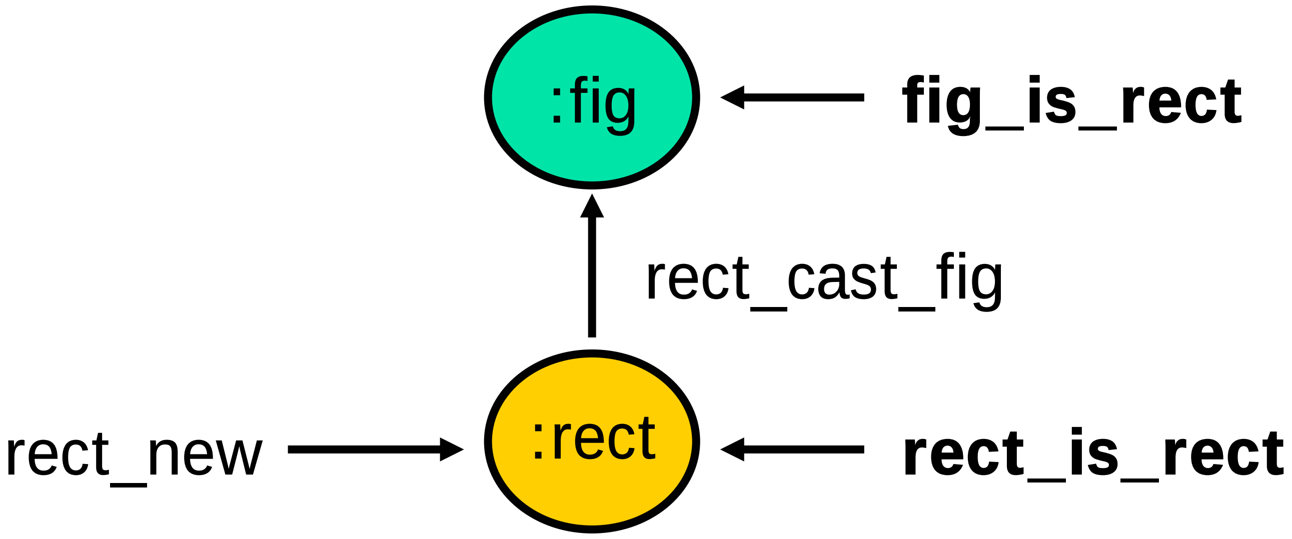



```
∀r s. rect_ex r s ⇒
   (fig_cast_rect (rect_cast_fig r s) s = r)
```

“Upcasting and downcasting results in the original object.”

Instance-of predicates

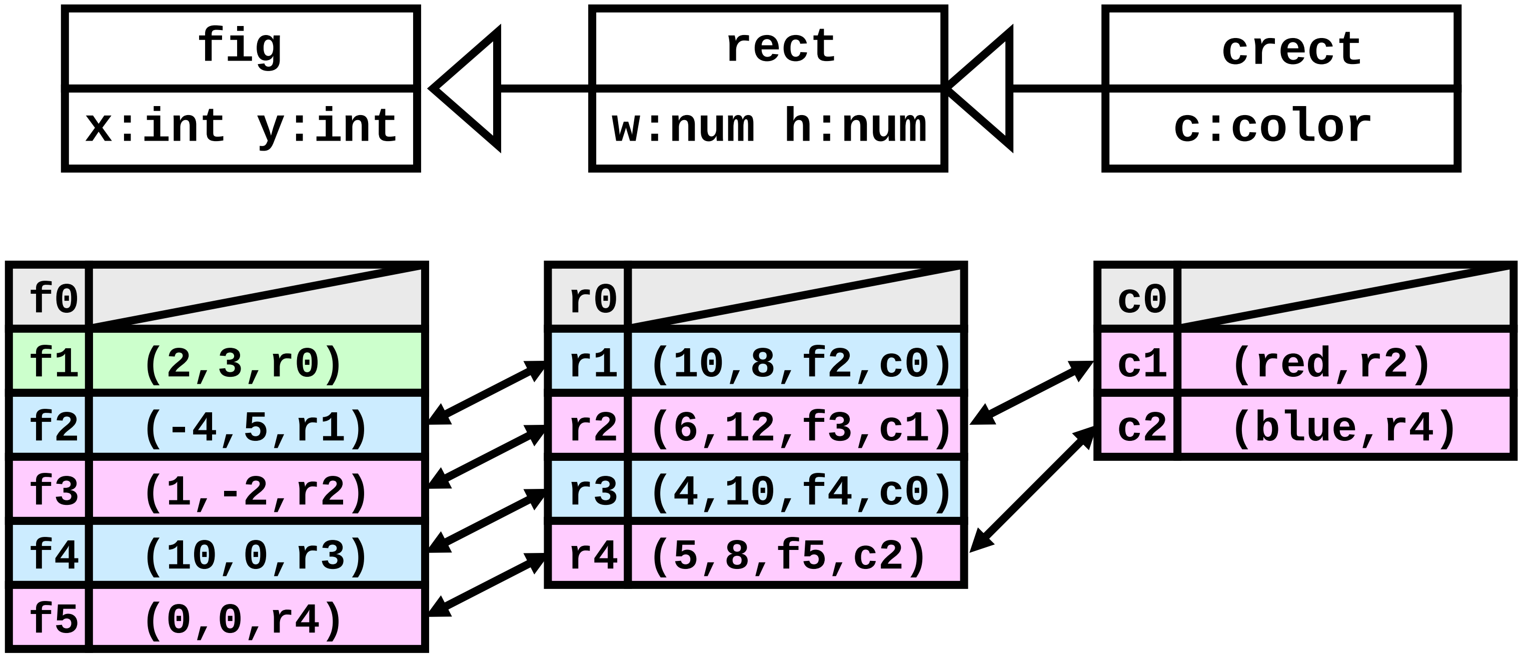
- Instance-of predicates remember the “actual types” of objects.




```
∀r s. rect_is_rect r s ⇒
   fig_is_rect (rect_cast_fig r s) s
```

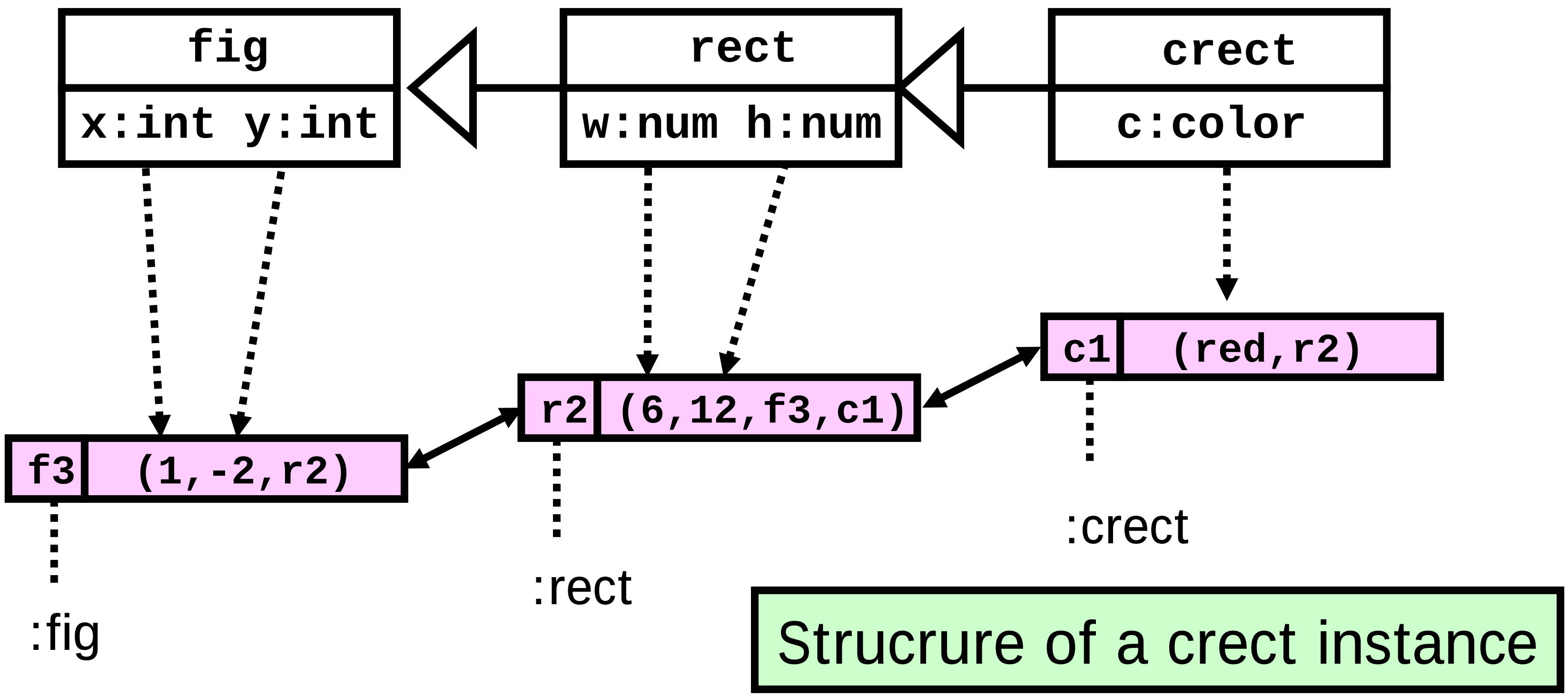
Heap memory model

- The theory is derived from the operational semantics of a heap memory model.



Object structure

- Subtyping mechanism is implemented by composing a single instance by multiple cells.

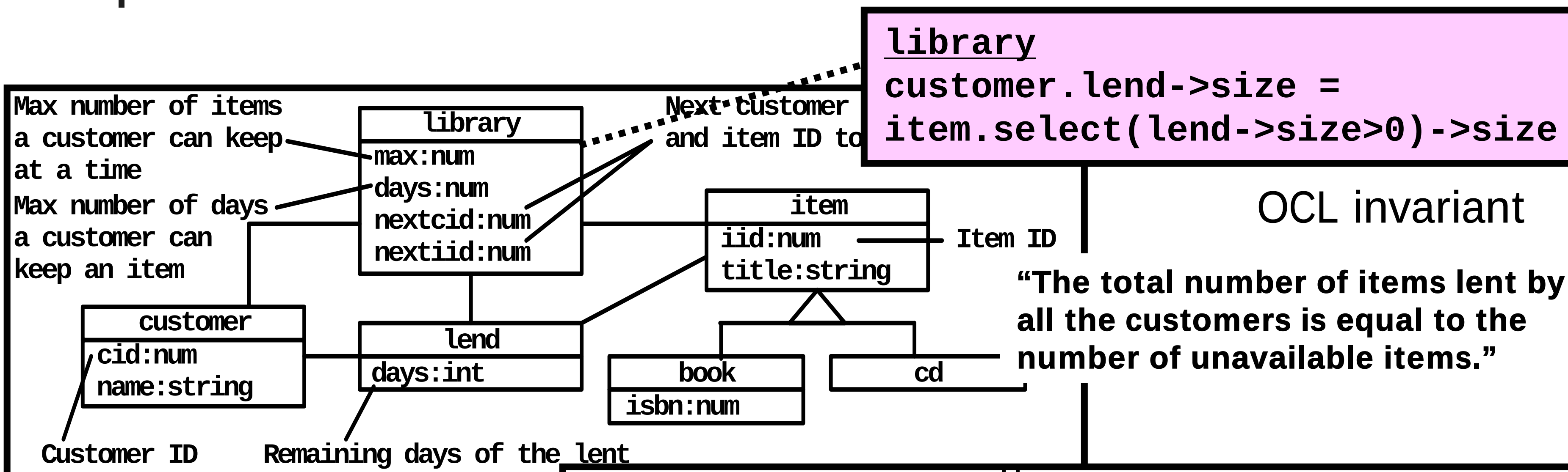


Prototype execution

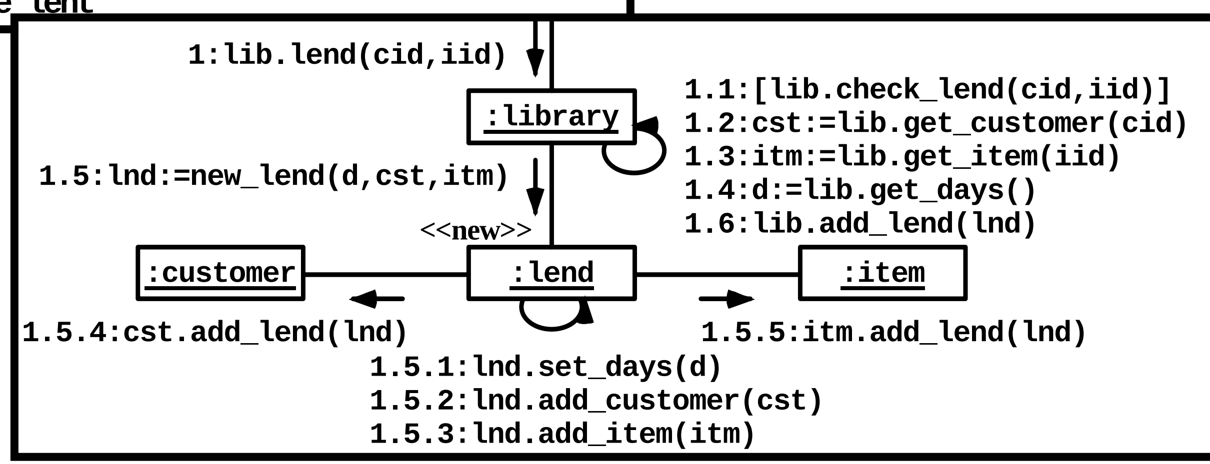
- The theory is executable in moscow-ML
  - Define the operational semantics of the heap memory model as actual operations in ML.

```
- val (f,s1) = fig_new store_emp;
> val f = <fig> : fig
> val s1 = <store> : store
- val s2 = fig_set_x f 10 s1;
> val s2 = <store> : store
- val x = fig_get_x f s2;
> val x = 10 : int
```

Example:
a simple library system



Class diagram



Collaboration diagram of lending service

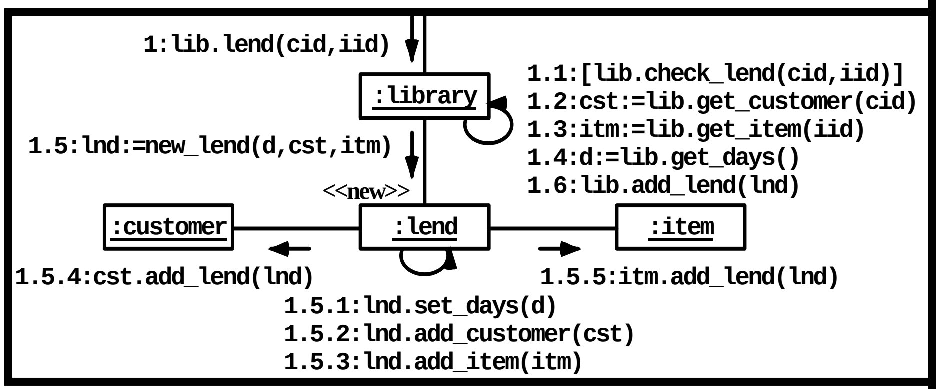
Invariant verification

- Verify that the collaboration maintains the invariant.

Collaboration	⇨	**F : store -> store**
Invariant	⇨	**P : store -> bool**

$$\boxed{\vdash \forall s.\ P(s) \Rightarrow P(F(s))}$$

Collaboration definition in HOL



Collaboration diagram of the lending service

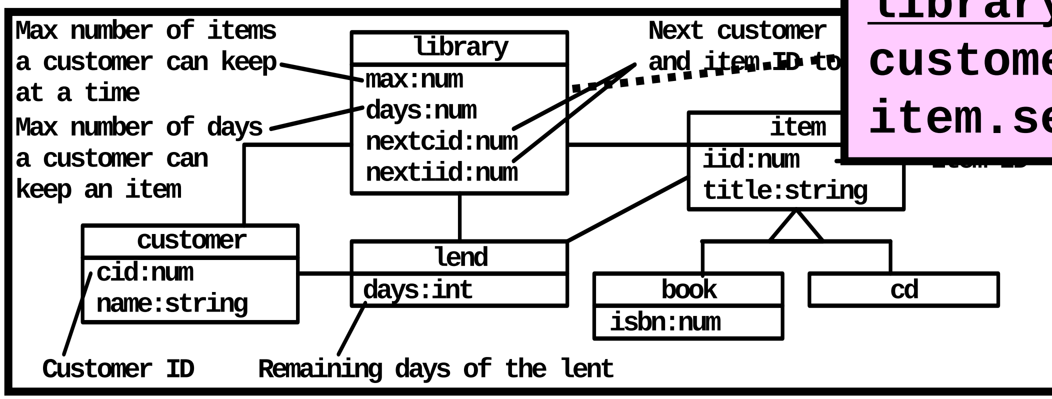
HOL representation of the collaboration

```
library_lend lib cid iid s =
  if library_check_lend lib cid iid s then
    let cst = library_get_custmer lib cid s in
    let itm = library_get_item lib iid s in
    let d = library_get_days lib s in
    let (lnd,s1) = new_lend cst itm d s in
    let s2 = library_add_lend lib lnd s1 in
      (“ok”, s2)
  else (“fail”, s)

new_lend cst itm d s =
  let (lnd,s1) = lend_new s in
  let s2 = lend_set_days lnd d s1 in
  let s3 = lend_add_customer lnd cst s2 in
  let s4 = lend_add_item lnd itm s3 in
  let s5 = customer_add_lend cst lnd s4 in
  let s6 = item_add_lend lnd s5 in
    (lnd, s6)

Library_get_customer lib cid s =
  let l = library_get_customerlist lib s in
    FST (FILTER
      (¥x. customer_get_cid x s = cid) l)
customer_add_lend cst lnd s =
  let l = customer_get_lendlist cst s in
    customer_set_lendlist (lnd::l) s
```

Invariant definition in HOL




```
library
customer.lend->size =
item.select(lend->size>0)->size
```

OCL invariant

```
Inv lib s = library_ex lib s ==>
  (library_get_customer_lendsum lib s = library_get_item_lendsum lib s)

library_get_customer_lendsum lib s =
  let l = library_get_customer_list lib s in
    SUM (MAP (¥x. customer_get_lendnum x s) l)

library_get_item_lendsum lib s =
  let l = library_get_itemlist lib s in
    LENGTH (FILTER (¥x. ~(item_is_available x s)) l)

customer_get_lendnum cst s = LENGTH (customer_get_lendlist cst s)
item_is_avalable itm s = (LENGTH (item_get_lendlist itm s) = 0)
```

HOL representation of the invariant

Proof

- The collaboration of the lending service maintains the invariant.

```
|- ∀lib cid iid s.
   Inv lib s ⇒
   Inv lib (library_lend lib cid iid s)
```

Related work

- Object embedding using extensible records
  - W. Narashewski et al. 1998
  - No referencing concept.
- Axiomatic semantics of UML models
  - T. Aoki et al. 2001
  - Directly introducing axioms in HOL.
- UML/OCL verification in B
  - R. Marcano et al. 2002
  - Methods are defined only by attaching pre- and post-conditions.

Conclusion

- Defined an object theory for analysis model verification in HOL
  - Application-specific
  - Sound
  - Executable
- Verification of a library system
  - Verify a collaboration maintains an invariant.

Future work

- Make the proof efficient
  - Define a collaboration as a sequencial OO program and implement verification condition generator.
- COE-related work (security)
  - Verify security requirements of a system using the object theory.
    - FW: bad packets never goes into the internal network.