Implementing application-specific Object-Oriented theories in HOL

Kenro Yatake[1], Toshiaki Aoki[1,2], and Takuya Katayama[1]

[1] Japan Advanced Institute of Science and Technology,
1-1 Asahidai Nomi Ishikawa 923-1292, Japan
{k-yatake, toshiaki, katayama}@jaist.ac.jp
[2] PRESTO JST

Abstract. This paper presents a theory of Object-Oriented concepts embedded shallowly in HOL for the verification of OO analysis models. The theory is application-specific in the sense that it is automatically constructed depending on the type information of the application. This allows objects to have attributes of arbitrary types, making it possible to verify models using not only basic types but also highly abstracted types specific to the target domain. The theory is constructed by definitional extension based on the operational semantics of a heap memory model, which guarantees the soundness of the theory. This paper mainly focuses on the implementation details of the theory.

1 Introduction

The Object-Oriented developing method is becoming the mainstream of the software development. In the upstream phase of the development, analysis models are constructed with a language such as UML (Unified Modeling Language [1]). To ensure the correctness of the models, formal semantics must be given to them and verification method such as theorem proving must be applied.

A lot of OO semantics have been implemented in theorem provers of higher-order logic and most of them are for the verification of OO languages such as Java [5][6][7]. In these theories, available types are limited to the primitive ones such as integers and boolean sufficient for the program verification. But for the verification of analysis models, this type restriction is disadvantage as the models are constructed with highly abstracted types specific to the target domain, e.g. tree, stack, date, money. Therefore, an OO semantics which can accommodate various types are required. So, we defined a theory in the HOL system [2] as a semantics of OO concepts in which arbitrary concrete types can be incorporated in the types of object attributes. In general, an object is a data which holds multiple attributes of arbitrary types and even allows referencing and subtyping. This concept is too complex to implement as a general type in the simple first-order type system of HOL. To cope with this problem, we take the approach of automatically constructing the theory depending on the class model of the application which defines the type information of the system.

The theory is constructed by *definitional extension*. This is a standard method to construct sound theories in HOL, where new theories are derived from existing sound theories by only allowing introduction of definition and derivation by sound inference rules. Specifically, the theory is derived from the operational semantics of a heap memory model. If a class model is given in advance, objects and their referencing and subtyping are realized by a linked-tuple structure in the heap memory and the resulting theory becomes quite simple.

In this paper, we present the definition of the theory and its implementation details in HOL. As a verification example, we prove that a UML collaboration diagram satisfies an invariant written in OCL (Object Constraint Language [3]). In this paper, we call the theory ASOOT (for Application-Specific Object-Oriented Theory).

This paper is organized as follows. In section 2 and 3, we explain the definition of the class model and the definition of the theory corresponding to the class model. In section 4, we explain the implementation details. In section 5, we show the example verification. In section 6, we cite related works and section 7 is conclusion and future work.

2 Class models

The theory depends on the class model of each system which defines the static structure of the system like UML class diagrams. The class model is defined as a six tuple:

$$CM = (C, A, \mathcal{M}_{attr}, \mathcal{M}_{inher}, \mathcal{T}, \mathcal{V})$$

The sets C and A are class names and attribute names which appear in the system, respectively. The mapping $\mathcal{M}_{attr} : C \to Pow(A)$ relates a class to the attributes defined in the class. The mapping $\mathcal{M}_{inher} : C \to Pow(C)$ relates a class to its direct subclasses. We assume single inheritance. The mapping $\mathcal{T} : C \times A \to Type$ relates an attribute to its type. The set $Type$ is a set of arbitrary concrete types in HOL. We assume $C \subset Type$ and define the type of an object to be the name of the class it belongs to. The mapping $\mathcal{V} : C \times A \to Value$ relates an attribute to its default value. The set $Value$ is a set of values of all types in $Type$. By a symbol $\lhd$, we denote the super-sub relationship. The expression $c_1 \lhd c_2$ means c_2 is a direct subclass of c_1, which is equivalent to $c_2 \in \mathcal{M}_{inher}(c_1)$. In addition, $c_1 \lhd^+ c_2$ means c_2 is a descendant class of c_1 and $c_1 \lhd^* c_2$ means $c_1 = c_2$ or $c_1 \lhd^+ c_2$. By $attr(c)$, we denote the attributes and inherited attributes of the class c, i.e. $attr(c) = \{a | a \in \mathcal{M}_{attr}(d), d \lhd^* c\}$.

In the following, we visualize the class models like Fig.1. The class `fig` is a class of figures which has two attributes `x` and `y` as its coordinate position. The class `rect` is a class of rectangles which has two attributes `w` and `h` as its width and height. The class `crect` is a class of colored-rectangle which has an attribute `c` as its color. The type `color` is an enumeration type which has several colors as its elements.

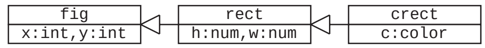


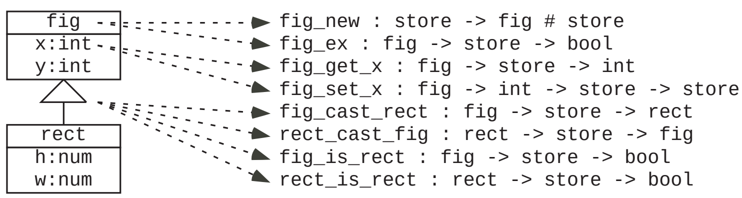
Fig. 1. A class model example

3 Definition of ASOOT

The theory is defined in HOL by mapping the class model elements to types and constants in the theory and introducing axioms on them. As the embedding policy, we chose a shallow embedding because our verification target is individual applications of the class model (the comparison of a shallow embedding and a deep embedding is found in [4]). We first explain the overview of the theory with the example, and then give the formal definition.

3.1 Overview

In order to implement object referencing, the concept *store* is introduced in the theory. The store is an environment which holds the attribute values of all alive objects in the system and defined as a type `store`. Objects are references to their data in the store and defined as types of their belonging class name. For example, the type of objects of the class `fig` is `fig`. Types of objects are "apparent" types and their type can be transformed to other types by casting.




```
[A1]|- !s. let (f,s1) = fig_new s in fig_ex f s1
[A2]|- !f v s. fig_ex f s ==> (fig_get_x f (fig_set_x f v s) = v)
[A3]|- !r s. rect_is_rect r s ==> fig_is_rect (rect_cast_fig r s) s
```

Fig. 2. The mapping from the class model to the theory

Several kinds of constants are introduced in the theory by mapping from the elements in the class model as shown in Fig.2. For example, corresponding to the class `fig`, two constants `fig_new` and `fig_ex` are introduced. The function `fig_new` creates a new `fig` instance in the store. It takes a store as an argument and returns a pair of a newly created object and the store after the creation. The predicate `fig_ex` tests if a `fig` object exists in the store. It takes a `fig` object and a store as arguments and return the result as a boolean value. The first axiom is a property about these operators saying "The newly created object is alive in the store after the creation."

Corresponding to the attribute `x` of the class `fig`, read and write operators `fig_get_x` and `fig_set_x` are introduced. The function `fig_get_x` takes a `fig` object and a store as arguments and returns the current value of the attribute `x`. The function `fig_set_x` takes a `fig` object, a new integer value and a store as arguments and returns the store after updating the attribute `x` to the new value. The second axiom says "If the `fig` object is alive in the store, the value of the attribute `x` of the object obtained just after updating it to `v` equals to `v`."

Corresponding to the inheritance relationship between the two classes `fig` and `rect`, type casting operators and instance-of operators are introduced. The function `fig_cast_rect` takes a `fig` object as an argument and casts it downward from `fig` to `rect`. The function `rect_cast_fig` takes a `rect` object and casts it upward from `rect` to `fig`. The predicate `fig_is_rect` tests if a `fig` object is an instance of the class `rect`. After an object is created, its apparent type can be changed by casting operators, but instance-of operators play a role of remembering the actual type of the object. For example, by applying `rect_cast_fig` to the `rect` instance which is created by `rect_new`, its apparent type is changed to `fig`, but as `fig_is_rect` holds for the `fig` object, it is identified as an instance of the class `rect`. The third axiom state this.

3.2 Types and constants

The store is represented by a type *store*. It has a constant *Emp* as its initial value which represents the empty store. Objects of the class c are represented as a type c. Each type c has a constant $Null^c : c$ which represents the null object.

Following operators are defined on the store:

$$\begin{aligned}
&Ex^c : c \to store \to bool \quad (c \in C)\\
&Get^c_a : c \to store \to \mathcal{T}(c,a) \quad (c \in C, a \in attr(c))\\
&Set^c_a : c \to \mathcal{T}(c,a) \to store \to store \quad (c \in C, a \in attr(c))\\
&Cast^c_d : c \to store \to d \quad (c, d \in C, c \lhd^+ d \text{ or } d \lhd^+ c)\\
&New^c : store \to c * store \quad (c \in C)\\
&Is^c_d : c \to store \to bool \quad (c, d \in C, c \lhd^* d)
\end{aligned}$$

The predicate Ex^c tests if the class c object is alive in the store. The function Get^c_a reads the attribute a of the class c object. If it is applied to an object not alive in the store, the constant $Unknown^c_a : \mathcal{T}(c,a)$ which represents the undefined value is returned. The function Set^c_a updates the attribute a of the class c object. The function $Cast^c_d$ transforms the object types from c to d. The function New^c creates a new instance of the class c in the store. The predicate Is^c_d tests if the class c object is an instance of the class d.

3.3 Axioms

Here, we introduce axioms for the operators defined above. There are 36 axioms altogether, but we show only main ones because of space limitations.

1. $\forall o\ s.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee ... \vee Is^c_{d_n}\ o\ s \quad (\{d_1, ..., d_n\} = \{d \mid c \lhd^* d\})$
   The c object o alive in the store is an instance of either the class c or one of the descendant-classes of c.
2. $\forall o\ s.\ Is^c_d\ o\ s \Rightarrow \neg(Is^c_e\ o\ s) \quad (d \neq e)$
   If the c object o is an instance of the class d, it is not an instance of the class e different from d, i.e. is-operators are exclusive.
3. $\forall o\ s.\ Is^d_e\ o\ s \Rightarrow Is^c_e\ (Cast^d_c\ o\ s)\ s \quad (c \lhd^+ d)$
   If the d object o is an instance of the class e, the object cast to the superclass c is also the instance of e, i.e. the actual type is invariable by casting.
4. $\forall o\ s.\ Is^c_c\ o\ (Snd\ (New^c\ s)) = (o = Fst\ (New^c\ s)) \vee Is^c_c\ o\ s$
   The c object o is an instance of the class c in the store after creating a new instance of the class c iff o is either the newly created object or the object which was already an instance of c before the creation.
5. $\forall o\ s.\ \neg(Ex^c\ (Fst\ (New^c\ s))\ s)$
   The newly created object does not exist in the previous store. This axiom implies that the new object is distinct from all previous objects.
6. $\forall o_1\ o_2\ s.\ Ex^d\ o_1\ s \wedge Ex^d\ o_2\ s \Rightarrow$
   $\neg(o_1 = o_2) \Rightarrow \neg(Cast^d_c\ o_1\ s = Cast^d_c\ o_2\ s) \quad (c \lhd^+ d)$
   If two c objects o_1 and o_2 are different objects, the two object obtained by casting to the superclass c are also different objects, i.e. cast-operators are injective.
7. $\forall o\ s.\ Is^c_e\ o\ s \Rightarrow (Cast^d_c\ (Cast^c_d\ o\ s)\ s = o) \quad (c \lhd^+ d, d \lhd^+ e)$
   If the c object o is an instance of the class e which is a descendant class of d, the object obtained by down-casting to d and then up-casting to c equals to o itself.
8. $\forall o\ s.\ Get^d_a\ o\ s = Get^c_a\ (Cast^d_c\ o\ s)\ s \quad (c \lhd^+ d \text{ and } a \in \mathcal{M}_{attr}(c))$
   When an attribute a is defined in the class c, getting a of the object o of the descendant-class d is the same as getting a by casting o to c.
9. $\forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get^c_a\ o\ (Set^c_a\ o\ x\ s) = x)$
   If the object o is alive in the store, the attribute a of o obtained just after updating it to x equals to x.
10. $\forall o_1\ o_2\ s.\ \neg(o_1 = o_2) \Rightarrow (Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s) = Get^c_a\ o_1\ s)$
    If the two objects o_1 and o_2 are different, getting the attribute a of o_1 is not affected by the updating of the same attribute of o_2.
11. $\forall o_1\ o_2\ s.\ Get^c_a\ o_1\ (Set^d_b\ o_2\ x\ s) = Get^c_a\ o_1\ s \quad ((c \not\lhd^* d \text{ and } d \not\lhd^* c) \text{ or } a \neq b)$
    If the two classes c and d are not in inheritance relationship or the attribute name a and b are different, getting the attribute is not affected by the updating.

3.4 Modeling OO concepts in the theory

Basic OO concepts such as methods, inheritance, overriding and dynamic binding are expressible in the theory. We show a typical way to model these concepts using examples. In HOL, we denote the operators Ex^c, New^c, Get^c_a, Set^c_a, $Cast^c_d$ and Is^c_d as `c_ex`, `c_new`, `c_get_a`, `c_set_a`, `c_cast_d` and `c_is_d`, respectively.

Methods are defined using *Get*, *Set*, *New*, *Cast* and functions provided in HOL. Let us consider that the class `fig` has a method `move` which changes its position by `dx` and `dy`. This method is defined as follows.

```
fig_move : fig -> int -> int -> store -> store
fig_move f dx dy s =
   let (x,y) = (fig_get_x f s, fig_get_y f s) in
      fig_set_y f (y+dy) (fig_set_x f (x+dx) s)
```

Method inheritance is modeled by calling the superclass method from the subclass method, i.e. by casting the object to the superclass type and applying the superclass method. If the class `rect` inherits the method `move` of the superclass `fig`, this method is defined as follows.

```
rect_move : rect -> int -> int -> store -> store
rect_move r dx dy s = fig_move (rect_cast_fig r s) dx dy s
```

Method overriding is modeled in the same manner as method inheritance. If the class `crect` overrides the superclass method `move` to change the color to `red` after changing the position, this method is defined as follows.

```
crect_move : crect -> int -> int -> store -> store
crect_move c dx dy s =
   let s1 = rect_move (crect_cast_rect c s) dx dy s in
      crect_set_color c red s1
```

Dynamic binding is a mechanism to dynamically switch method bodies according to which class the applied object is instance of. This is modeled by defining a virtual method which selects the method body using Is. The virtual method `v_fig_move` corresponding to the method `fig_move` is defined as follows.

```
v_fig_move : fig -> int -> int -> store -> store
v_fig_move f dx dy s =
  if fig_is_fig f s then fig_move f dx dy s
  else if fig_is_rect f s then rect_move (fig_cast_rect f s) dx dy s
  else if fig_is_crect f s then crect_move (fig_cast_crect f s) dx dy s
  else s
```

4 Implementing ASOOT in HOL

We implemented a tool called ASOOT generator which inputs a class model and outputs the theory specific to the model. As mentioned in the introduction, the theory is constructed by definitional extension and thus sound. It implements the operational semantics of a heap memory using primitive theories such as natural numbers, lists and pairs and derives the theory from the operational semantics. We first explain the overview of the implementation using the example, and then, explain it formally.

4.1 Overview

The store is represented as a heap memory to store object attributes. Fig.3 shows a snapshot of the heap memory for the example model. The heap memory consists of three sub-heaps which are introduced corresponding to the three classes `fig`, `rect` and `crect`. Each sub-heap is represented by a list and the whole heap is represented by a tuple of them.

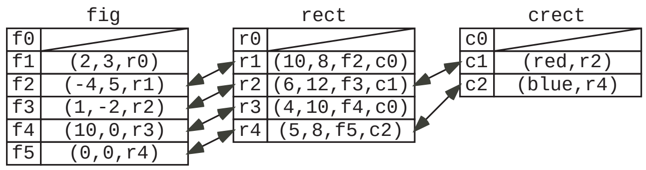


Fig. 3. Representation of the store

Object references are represented by indices of the memory. In the case of the `fig` memory, the reference `f1`, `f2`,... of type `fig` is represented by a natural number 1,2, ... The reference `f0` is used as a null reference `fig_null`. Object instances are represented by a tuple or several tuples in the sub-heaps. For example, the tuple in `f_1` represents a `fig` instance whose attribute are `x=2` and `y=3`. Two tuples in `f2` and `r1` together represent a `rect` instance whose attribute are `x=-4`, `y=5`, `w=10`, and `h=8`. Three tuples in `f3`, `r2`, and `c1` together represent a `crect` instance whose attribute are `x=1`, `y=-2`, `w=6`, `h=12`, and `c=red`. Multiple tuples which compose an instance are linked to each other by storing object references. The two tuples in `f2` and `r1` which compose a `rect` instance link to each other by storing the references `r1` and `f2`, respectively. If there are no tuples for a tuple to link, the null references are stored. As the tuple in `f1` does not link to any `rect` tuples, it stores the null reference `r0`. Object subtyping is modeled by this linked-tuple structure. For example, three references `f3`, `r2` and `c1` all point at the same `crect` instance. This means `crect` instance can have three apparent types `fig`, `rect`, and `crect`.

Now, we explain how the operators on the store are implemented in the heap memory. The *New* operator `rect_new` is implemented as a function to add new tuples in the sub-heaps for `fig` and `rect` and connects them to each other. The *Ex* operator `fig_ex` is implemented as a predicate to test if the `fig` reference is not null and not out of bounds of the sub-heap for `fig` . The *Cast* operator `fig_cast_rect` is implemented as a function to read the `rect` reference stored in the tuple pointed by the `fig` reference. The *Get* and *Set* operator `fig_get_x` and `fig_set_x` are implemented as functions to read and update the first element in the tuple pointed by the `fig` reference. The *Is* operator `fig_is_rect` is implemented as a predicate to test if the tuple pointed by the `fig` reference is linked with a tuple in the sub-heap for `rect`.

4.2 Representation of the store: a heap memory model

A sub-heap is defined independent of the class model and is represented generally as $'a\ list$. Addresses of data is represented by list indices, or natural numbers 0, 1, 2... The initial value of a sub-heap is defined as $[null]$ which is a list with a dummy constant $null : 'a$ in the address 0. Four operators add, $valid$, $read$ and $write$ are defined on the sub-heap as follows.

$$
\begin{aligned}
& add\ x\ l = (Length\ l, Append\ l\ [x]) \\
& valid\ n\ l = (0 < n) \wedge (n < length\ l) \\
& read\ n\ l = if\ valid\ n\ l\ then\ read_1\ n\ l\ else\ unknown \\
& \quad (read_1\ 0\ l = Hd\ l) \wedge (read_1\ (Suc\ n)\ l = read_1\ n\ (Tl\ l)) \\
& write\ n\ x\ l = if\ valid\ n\ l\ then\ write_1\ n\ x\ l\ else\ l \\
& \quad (write_1\ 0\ x\ l = x :: (Tl\ l)) \wedge (write_1\ (Suc\ n)\ x\ l = (Hd\ l) :: (write_1\ n\ x\ (Tl\ l)))
\end{aligned}
$$

The function add adds the new data x at the tail of the list and returns the new address and the list after the operation. The predicate $valid$ tests if a data is stored in the address n. The address is valid if it is in the range greater than 0 and less than the current list length. The function $read$ reads the data in the address n. If the address is not valid, the constant $unknown$ which represents undefined data is returned. The function $write$ updates the data in the address n by the data x. If the address is not valid, the list is left unchanged.

Sub-heaps are introduced corresponding to each class and each of them stores different types of tuples depending on the class. The type of tuples stored in the sub-heap for the class c is defined as:

$$tuple_c \equiv \mathcal{T}(c, a_1) * ... * \mathcal{T}(c, a_n) * d * e_1 * ... * e_m \quad (a_i \in \mathcal{M}_{attr}(c), d \lhd c, c \lhd e_j)$$

The first n elements are the attributes defined in c. The next element is a reference of a superclass object. The last m elements are references of subclass objects. The type of the sub-heap storing these tuples is defined as $heap_c \equiv tuple_c\ list$.

The type of object references of the class c is obtained by defining bijections between the type c and natural numbers as follows.

$$HOL\_datatype\ c = AbsObj_c\ of\ num, \quad RepObj_c\ (AbsObj_c\ n) \equiv n$$

The function $AbsObj_c$ maps a natural number to a c object reference. The function $RepObj_c$ maps a c object reference to a natural number. The null object is represented by 0, i.e. $Null^c \equiv AbsObj_c\ 0$.

The whole heap memory is obtained by gathering sub-heaps into a tuple. The type of the heap memory is defined as:

$$Heap \equiv heap_{c_1} * \ ... \ * heap_{c_n} \quad (c_i \in C)$$

The four operators on the sub-heap add, $valid$, $read$ and $write$ are extended to operate on the whole heap as follows.

$$
\begin{aligned}
& Add^c : tuple_c \to Heap \to c * Heap, \qquad Valid^c : c \to Heap \to bool \\
& Read^c_u : c \to Heap \to T, \qquad Write^c_u : c \to T \to Heap \to Heap
\end{aligned}
$$

The function Add_c adds a new tuple in the sub-heap of the class c. The predicate $Valid^c$ tests if the c object is valid in the sub-heap of the class c. The function $Read^c_u$ reads the element u in the tuple referenced by the c object. The element u is either one of a_i for attributes, d for the superclass object, or e_j for the subclass object. In the case $u = a_i$, $T = \mathcal{T}(c, a)$ and for other case, $T = u$. The function $Write^c_u$ writes at the same location in the heap as $Read^c_u$ reads. These operators are easily defined using pair functions Fst and Snd and the bijections $AbsObj_c$ and $RepObj_c$.

4.3 Representation of ASOOT constants

We define constants $EmpRep$, $ExRep^c$, $CastRep^c_d$, $GetRep^c_a$, $SetRep^c_a$, $NewRep^c$ and $IsRep^c_d$ using the operators defined on the heap memory. They are the heap representations of the ASOOT constants Emp, Ex^c, $Cast^c_d$, Get^c_a, Set^c_a, New^c and Is^c_d, respectively.

The constant $EmpRep$ is defined as:

$$EmpRep \equiv ([null : tuple_{c_1}], ..., [null : tuple_{c_n}]) \quad (c_i \in C)$$

The empty store is represented by a tuple of the initial values of the sub-heaps.

The predicate $ExRep^c$ is defined as:

$$ExRep^c\ o\ H \equiv Valid^c\ o\ H$$

The existence of an object in the store is represented by the validity of the object reference in the heap memory.

The function $CastRep^c_d$ is defined as:

$$CastRep^c_d\ o\ H \equiv$$
$$\begin{cases} if\ ExRep^c\ o\ H\ then\ Read^c_d\ o\ H\ else\ Null^d & (c \lhd d \text{ or } d \lhd c) \\ CastRep^e_d\ (CastRep^c_e\ o\ H)\ H & ((c \lhd e, e \lhd^+ d) \text{ or } (d \lhd^+ e, e \lhd c)) \end{cases}$$

In the case that the two classes c and d are in the direct super-sub relationship, the casting is represented by reading the d object in the tuple referenced by the c object. If the c object does not exists, it is cast to the null object $Null^d$. In the case that c and d are in the ancestor-descendant relationship but not in the direct super-sub relationship, the casting is applied transitively, i.e. first the c object is cast to the direct superclass e and then the e object is cast to the class d.

The functions $GetRep^c_a$ is defined as:

$$GetRep^c_a\ o\ H \equiv$$
$$\begin{cases} if\ ExRep^c\ o\ H\ then\ Read^c_a\ o\ H\ else\ Unknown^c_a & (a \in \mathcal{M}_{attr}(c)) \\ GetRep^d_a\ (CastRep^c_d\ o\ H)\ H & (d \lhd c, a \in attr(d)) \end{cases}$$

In the case that the attribute a is defined in the class c, getting a of a c object is represented by reading the element a in the tuple referenced by the c object. If the

c object does not exists, a constant $Unknown_a^c$ which represents the undefined value is returned. In the case that the attribute a is defined in the ancestor-class, the c object is cast to the superclass d and then $GetRep_a^d$ is applied.

The function Set_a^c is defined in the same way as $Read_a^c$:

$$SetRep_a^c\ o\ x\ H \equiv$$
$$\begin{cases} if\ ExRep^c\ o\ H\ then\ Write_a^c\ o\ x\ H\ else\ H & (a \in \mathcal{M}_{attr}(c)) \\ SetRep_a^d\ (CastRep_d^c\ o\ H)\ x\ H & (d \lhd c, a \in attr(d)) \end{cases}$$

If the c object does not exists, the heap is left unchanged.

The function $NewRep^c$ is defined as:

$$NewRep^c\ H \equiv \begin{cases} Add^c\ default_c\ H & (c \text{ is the root class}) \\ let\ (o_1, H_1) = NewRep^d\ H\ in & \\ \quad let\ (o_2, H_2) = Add^c\ default_c\ H_1\ in & \\ \quad\quad let\ H_3 = Link_c^d\ o_1\ o_2\ H_2\ in\ \ (o_2, H_3) & (d \lhd c) \end{cases}$$

$$\begin{aligned} &where \\ &\quad Link_c^d\ o_1\ o_2\ H \equiv Write_d^c\ o_2\ o_1\ (Write_c^d\ o_1\ o_2\ H) \\ &\quad default_c \equiv (\mathcal{V}(c, a_1), ..., \mathcal{V}(c, a_n), Null^d, Null^{e_1}, ..., Null^{e_m}) \\ &\quad\quad (a_i \in \mathcal{M}_{attr}(c), d \lhd c, c \lhd e_j) \end{aligned}$$

This function creates a linked-tuple structure recursively on the inheritance chain. As a base step, where the class c is the root class of the inheritance tree, the c instance is created by simply adding a new tuple to the sub-heap for c. As induction steps, first, the instance of the superclass d is created by $NewRep^d$ and then, a new tuple is added to the sub-heap for c, and finally, the newly obtained object o_1 and o_2 is linked by $Link_c^d$. The tuple value $default_c$ added to the sub-heap for c contains default values for attributes and null objects for the superclass and subclass objects.

The predicate $IsRep_d^c$ is defined as:

$$IsRep_d^c\ o\ H \equiv \begin{cases} ExRep^c\ o\ H \wedge \bigwedge_j \neg ExRep^{e_j}\ (CastRep_{e_j}^c\ o\ H)\ H & (c = d, c \lhd e_j) \\ ExRep^c\ o\ H \wedge IsRep_e^d\ (CastRep_d^c\ o\ H)\ H & (c \lhd e, e \lhd^* d) \end{cases}$$

This predicate tests that the c object is the instance of the class d. This is tested by examining if the links are traversed from the c object reference up to a tuple in the sub-heap of d. Link traversing is realized by cast operators. If $c = d$, the c object is the very c instance, so there must not exist any links to any of the subclasses $e_1, ..., e_m$. If c is the ancestor-class of d, the c object is cast to the subclass e and the e object must be an instance of d. In both cases, the c object must exist in the store.

4.4 Abstracting ASOOT from the heap memory

Finally, we abstract ASOOT from the heap representation by creating the type *store*, defining ASOOT constants and deriving axioms.

The type *store* is created from a subset of the type $Heap$. In HOL, it takes the following steps to create a new type t_1 from an existing type t_2.

1. Define the predicate $p : t_2 \rightarrow bool$ which determines the subset of t_2.
2. Prove the theorem $\vdash \exists x.\ p\ x$, i.e. the subset is not an empty set.
3. Assert that there are bijections between t_1 and the subset of t_2 determined by p.

The predicate which determines the subset of $Heap$ is defined as $IsStoreRep$ as follows [1].

$$
\begin{aligned}
&IsStoreRep\ H \equiv \forall P.\ IsInv\ P \Rightarrow P\ H\\
&where\\
&\quad IsInv\ P \equiv P\ EmpRep \wedge\\
&\qquad \bigwedge_{c,a}(\forall o\ x\ H.\ P\ H \Rightarrow SetRep^c_a\ o\ x\ H) \wedge \bigwedge_{c}(\forall H.\ P\ H \Rightarrow Snd\ (NewRep^c\ H))
\end{aligned}
$$

The elements of the subset represented by $IsStoreRep$ are those which satisfy the predicate P which is an invariant proved by the following induction: as a base step, prove that $EmpRep$ satisfies P, and as induction steps, assume that P holds for a heap and prove that the heaps obtained by applying $SetRep^c_a$ and $NewRep^c$ maintain P. The existence of an element is proved as a theorem $th \equiv \vdash IsStoreRep\ EmpRep$. The existence of bijections between *store* and the subset is asserted automatically by calling the ML function $new\_type\_definition(store, th)$. Let us say the bijections are $RepStore : store \rightarrow Heap$ and $AbsStore : Heap \rightarrow store$.

ASOOT constants are defined by taking a map with their heap representations as follows.

$$
\begin{aligned}
&Emp \equiv AbsStore\ EmpRep, \quad Ex^c\ o\ s \equiv ExRep^c\ o\ (RepStore\ s)\\
&Get^c_a\ o\ s \equiv GetRep^c_a\ o\ (RepStore\ s)\\
&Set^c_a\ o\ x\ s \equiv AbsStore\ (SetRep^c_a\ o\ x\ (RepStore\ s))\\
&Cast^c_d\ o\ s \equiv CastRep^c_d\ o\ (RepStore\ s), \quad Is^c_d\ o\ s \equiv IsRep^c_d\ o\ (RepStore\ s)\\
&New^c\ s \equiv let\ (o, H) = NewRep^c\ (RepStore\ s)\ in\ (o, AbsStore\ H)
\end{aligned}
$$

All the ASOOT axioms are derived from the definition we presented so far. The axioms are divided into two groups according to how they are derived. One is those which are derived simply by expanding the definitions. The axioms 2, 4, 5, 8, 9, 10 and 11 are in this group. The other is those which are proved as invariants on the store. The axioms 1, 3, 6 and 7 are in this group. Invariants are proved by the induction given in $IsInv$. Let us consider the proof of the axiom 1 defined as Inv as follows.

$$Inv\ s \equiv \forall o.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee ... \vee Is^c_{d_n}\ o\ s \quad (d_i \in \{d \mid c \lhd^* d\})$$

[1] There is a logically equivalent implementation of the theory where the number of steps of the induction in $IsInv$ becomes only $1 + 2c$.

First, we define the heap representation of the axiom as follows.

$$InvRep\ H \equiv \forall o.\ ExRep^c\ o\ H = IsRep^c_{d_1}\ o\ H \vee ... \vee IsRep^c_{d_n}\ o\ H$$

Then, we prove the theorem $\vdash IsInv\ InvRep$ based on the structural induction. If this holds, the theorem $\vdash \forall H.\ IsStoreRep\ H \Rightarrow InvRep\ H$ is derived from the definition of *IsStoreRep*. And as $IsStoreRep\ (RepStore\ s)$ holds (from the bijection theorem not presented here), we obtain the theorem $\vdash \forall s.\ InvRep\ (RepStore\ s)$. From this theorem and the definitions of Ex^c and Is^c_d, we obtain $\vdash \forall s.\ Inv\ s$.

5 A verification example

In this section, we show an example verification using ASOOT, where a UML collaboration diagram is verified to satisfy an invariant written in OCL.

The UML class diagram and collaboration diagram of the library system are shown in Fig.4. The system consists of four classes. The class `library` is the main class of the system and has the methods for operations such as item lending and customer registration. It has association with the classes `customer` and `item` which represent the customers and items registered in the library, respectively. There are two kinds of items: books and CDs. They are represented as subclasses `book` and `cd`. The class `lend` keeps the lending information between a customer and an item. In the class model, an association is defined as an attribute whose type is a list of objects, e.g. the association for `library` with `customer` is defined as an attribute `customerlist` of type `customer list`.

The lending operation is defined as a method `lend` of the class `library` and its collaboration proceeds as follows. First, the method is applied to an library object `lib` with two inputs: a customer ID (`cid`) and an item ID (`iid`). Then, it checks if the customer is qualified to lend the item (1.1). The conditions to check are: if the IDs are valid, if the customer currently keeps at most the maximum number of items specified by the library (`max`) and if the item is available. If the check is passed, the `customer` object (`cst`) and the `item` object (`itm`) corresponding to the IDs are obtained (1.2, 1.3) and the maximum number of days for the lent specified by the library (`days`) is obtained (1.4). Then, a new `lend` object (`lnd`) is created by the creation method `new_lend` (1.5). In this method, the `lend` object is set the remaining days for the lent (1.5.1) and linked to the `customer` object and the `item` object (1.5.2-1.5.5). Finally, the `lend` object is linked to the `library` object (1.6).

One of the invariants which must be met by the systems is: "The total number of items lent by all the customers is equal to the number of items unavailable." The OCL expression of this invariant is written as follows.

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

The method and the invariant are translated into a function `library_lend` and a predicate `Inv1`, respectively, as shown in Fig.5. We have not defined the formal translation, but it is our future work.

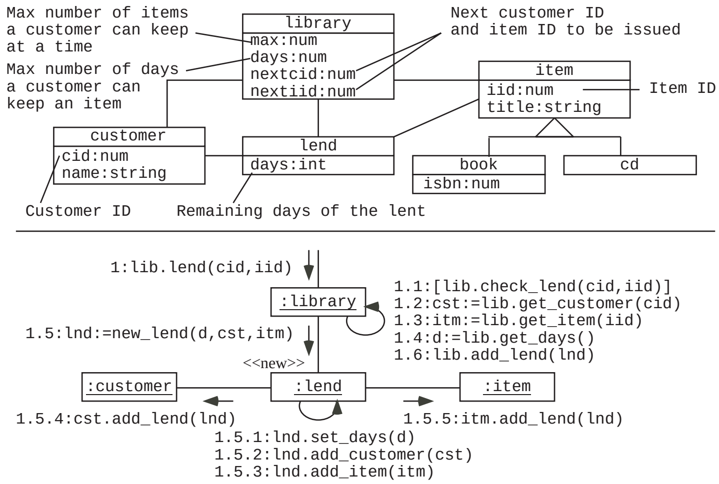


Fig. 4. The class diagram and the lending collaboration

Collaboration

```
library_lend : library -> num -> num -> store -> string # store
library_lend lib cid iid s =                      (* 1 *)
  if library_check_lend lib cid iid s then        (* 1.1 *)
    let cst = library_get_customer lib cid s in   (* 1.2 *)
    let itm = library_get_item lib iid s in       (* 1.3 *)
    let d = library_get_days lib s in             (* 1.4 *)
    let (lnd,s1) = new_lend d cst itm s in        (* 1.5 *)
      library_add_lend lib lnd s1                 (* 1.6 *)
  else s

new_lend : num -> customer -> item -> store -> lend # store
new_lend d cst itm s =
  let (lnd,s1) = lend_new s in
  let s2 = lend_set_days lnd d s1 in            (* 1.5.1 *)
  let s3 = lend_add_customer lnd cst s2 in      (* 1.5.2 *)
  let s4 = lend_add_item lnd itm s3 in          (* 1.5.3 *)
  let s5 = customer_add_lend cst lnd s4 in      (* 1.5.4 *)
  let s6 = item_add_lend itm lnd s5 in          (* 1.5.5 *)
    (lnd,s6)

library_get_customer : library -> num -> store -> customer
library_get_customer lib cid s =
  let l = library_get_customerlist lib s in
    HD (FILTER (\x. customer_get_cid x s = cid) l)

lend_add_customer : lend -> customer -> store -> store
lend_add_customer lnd cst s =
  let l = lend_get_customerlist lnd s in
    lend_set_customerlist lnd (cst::l) s
```

Invariant

```
Inv1 : library -> store -> bool
Inv1 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_customerlist lib s in
    LENGTH (FLATTEN (MAP (\x. customer_get_lendlist x s) l))

library_get_item_lendsum lib s =
  let l = library_get_itemlist lib s in
    LENGTH (FILTER (\x. 0 < LENGTH (item_get_lendlist x s)) l)
```

Fig. 5. Definitions of the collaboration (partially) and the invariant in HOL

The methods in the collaboration is defined as HOL functions and the whole collaboration is represented as their application sequence. This is a merit of ASOOT compared to the UML/OCL verification based on B [13][14] where methods are defined only as as pre- and post-conditions. ASOOT enables to define even the internal operation of the methods using HOL functions. For example, the method call at 1.2 is defined as a function `library_get_customer`. This method returns a `customer` object which has the ID equals to `cid` and is defined making use of the list function `FILTER`. The method call at 1.5 is defined as a function `new_lend` and the collaboration proceeds to the next depth. The method call at 1.5.2 is defined as a function `lend_add_customer`. This function adds the object `cst` to the attribute `customerlist` using the *Get* and *Set* operators. As for the invariant, the left-hand-side is defined as a function `library_get_customer_lendsum`. The navigation `customer.lend` is represented by getting the `lendlist` of all the `customer` object using `MAP`, and then, flattening the nested list using `FLATTEN`. The set operation `size` is represented by `LENGTH`. The predicate `Inv1` takes the `library` object as its first argument. This represents the context object.

The fact that the invariant is maintained by application of the collaboration is proved as the following theorem.

```
|- !lib cid iid s.
   Inv1 lib s /\ Inv2 lib s ==> Inv1 lib (library_lend lib cid iid s)
```

The predicate `Inv2` is another invariant required as lemma which we omit to explain the details. The whole proof proceeds on the abstract level of ASOOT (without expanding the definition of ASOOT constants).

6 Related work

J. Berg et al. [9] and Claude Marché et al. [10] define memory models for reasoning Java programs annotated with JML specifications. The first work defines the memory with untyped blocks, so that it can store arbitrary Java objects. The second work introduces multiple heap memories for different types in order to statically tell the types of each memory contents. Our memory model differs from them in that it can store values of arbitrary types not limited to the primitive ones in Java. This is important when it comes to the verification on the analysis level as the models are abstracted with high-level types such as list, set, and tree. We made this possible by constructing the memory depending on the type information of the application. Moreover, we can take advantage of the plentiful mathematical libraries and the powerful type definition package provided by HOL to define high-level types. Actually, those types can be implemented using Java classes with primitive types, but it will take additional proof steps to derive type properties from those class implementations compared to use HOL types directly.

A. Poetzsch-Heffer et al. [8] defines a Hoare-style logic for the verification of OO programs. As a logical foundation of the logic, it defines an OO theory based on the store model in HOL. The operators on the store are *get*, *set*, *new*, *alive*. The last one corresponds to Ex in our theory. It does not have the operators concerning subtyping like $Cast$ and Is in our theory, and the axioms about subtyping are defined on the Hoare-logic level. In our theory, we included the axioms about subtyping on the store level by introducing the subtyping operators $Cast$ and Is. As a result, the store theory becomes independent of the Hoare logic.

W. Naraschewski et al. [11] defines an object as an *extensible record* in Isabelle/HOL. This is a record in which a type variable is embedded as one of its element. Although this record enables structural subtyping of objects, it does not work as a reference. To allow object referencing, we defined our theory based on the store. With the referencing mechanism, verification of object collaboration becomes possible.

T. Aoki et al. [12] defines a semantics for the statechart-based verification of invariants about object attributes in HOL. The semantics is constructed by directly introducing axioms in HOL. The advantage of this axiomatic theory construction is that the mapping between the model elements and the theory element becomes clear, but the problem is that it may weaken the reliability of the theory. On the other hand, the definitional construction adopted in this paper guarantees the soundness of the theory.

7 Conclusion and future work

In this paper, we presented an OO theory for the verification of analysis models which we implemented in HOL. In order to allow arbitrary types in object attributes, the theory is automatically constructed depending on the class model of the system. The theory is derived from the operational semantics of a heap memory model and is guaranteed to be sound by definitional extension mechanism. Using the theory, a UML collaboration diagram is verified to satisfy an OCL invariant.

Future work includes the formalization of the UML collaboration diagram and its translation to the theory. We are considering to develop a Hoare-style logic for the verification of collaborations and implementation of a verification condition generator to make proof efficient. One of the future goal is to apply the verification method to collaboration-based designs [15] [16].

References

1. OMG. Unified Modeling Language.
   URL: http://www.omg.org/.
2. The HOL system.
   URL: http://hol.sourceforge.net/.
3. J. Warmer and A. Kleppe. The object constraint language: precise modeling with UML. Addison-Wesley, 1999.

4. Tobias Nipkow, David von Oheimb and Cornelia Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In Foundations of Secure Computation. IOS Press, 2000.
5. Bart Jacobs et al. LOOP project, http://www.cs.kun.nl/ bart/LOOP/
6. David von Oheimb. Hoare Logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience, vol.13 pp.1173-1214, 2001.
7. A. Poetzsch-Heffer and P. Muller. A programming logic for sequential Java. Programming Languages and Systems (ESOP'99), vol.1576 LNCS Springer-Verlag, 1999.
8. A. Poetzsch-Heffer and P. Muller. Logical Foundations for Typed Object-Oriented Languages. Programming Concepts and Methods (PROCOMET), 1998.
9. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
10. Claude Marché and Christine Paulin-Mohring. Reasoning on Java programs with aliasing and frame conditions. In 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS, August 2005.
11. W. Naraschewski and M. Wenzel. Object-Oriented Verification based on Record Subtyping in Higher-Order Logic. Tecnische Universitat Munchen, 1998.
12. Toshiaki Aoki, Takaaki Tateishi, and Takuya Katayama. An Axiomatic Formalization of UML Models. Practical UML-based Rigorous Development Methods, pp.13-28 2001.
13. Using B formal specifications for analysis and verification of UML/OCL models. Marcano, R. and N. Levy. Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language. Dresden, Germany, October 2002.
14. K. Lano, D. Clark and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004.
15. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1998.
16. Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In Symposium on the Foundation of Software Engineering, 2001.