Verification of Java Programs in Coq

Seokhyun Han
Department of Computer Science
Royal Holloway, University of London
Surrey, UK
Email: seokhyun@cs.rhul.ac.uk

Abstract—This paper is a research on functional interpretation of object-oriented programs in the intensional type theory with dependent record types and coercive subtyping. We are here simulating a type-theoretic model of Java programs in Coq. Representing a class and its interface-type, which declares a set of methods and their signatures for code reuse, as dependent record types, the type-theoretic encoding enjoys desirable subtyping relationships that correctly capture the important object-oriented features such as inheritance, subtype polymorphism and dynamic dispatch. Furthermore, since the model is given in the intensional type theory, machine-supported verification of Java programs can be done by proving specifications that is satisfied by Java programs in Coq with regard to the state of objects.

I. INTRODUCTION

The purpose of the present paper is to offer a functional analysis of reconstructing object-oriented programs and their precise specifications in the Coq system [Coq] [BC04]. The author [Han10b] gave already the type-theoretical semantics of Java programs in UTT [Luo94] with dependent record types [Luo09] and coercive subtyping [Luo97] [Luo99] [Luo05] [Bai99]. UTT is an intensional and dependent type theory specified by a typed version of Martin-Löf’s logical framework [Luo94], which is a kind of meta-languages specifying object-languages by using type theories. Here, we are using Coq only for simulation of modeling Java programs in UTT, because Coq supports coercive subtyping [Sal97] and a macro for dependent record types, which means that dependent record types are actually implemented as Σ-types with labels defined as global names. The dependent type theories such as Calculus of Inductive Construction [CH88]. UTT and Martin-Löf’s intensional type theory [NPS90] are the modeling mechanism for formal development. They are a uniform language for programming, specification, and reasoning, i.e. programming and reasoning in a single formalism. They are implemented in the proof assistants such as Coq and Agda [Agd], which are used effectively for formalization of mathematics and verification of programs.

This paper is to revisit with a practical angle the theme originally explored in [Han10b], which is to interpret theoretically the core features of object-oriented programs such as Encapsulation, Inheritance(Reuse), Subtype polymorphism, and Dynamic dispatch as type-theoretic models with dependent record types and coercive subtyping. The object-oriented paradigm [Sal98] is not only closely linked to the underlying hardware, in the sense that programming is based on the idea of changing stored values, but also an approach to the solution of problems in which all computations are performed in the context of classes and their objects. In contrast, functional programming paradigm [Tho91] [Hin97] promotes a more abstract style of programming, based on the idea of introducing functions and applying them to arguments. Accordingly, research [GM94] [AC96] on functional interpretation of object-oriented programs leads to considerably simpler programs, and supports a number of powerful new ways to structure and reason about programs by formalizing the behavior of object-oriented programs in inductive data types which corresponds to computation structure they eventually terminate. The approach to functional modeling that we shall adopt originates from the ideas of the record model [Car88] and self-application semantics [Kam88] [SU94]. In the functional model, methods are interpreted as functions and class instances(or objects) as records. We shall also follow the state-application interpretation [PT94], considering field updates only without methods updates and separating fields and methods - only fields will be grouped into a record type of states.

We describe a type-theoretic model of Java programs in which a class and its interface-type are represented as a dependent record type whose component types may depend on the values of previous fields. In our model, a class entry that defines a method \( M \) to be a specific \( a \) of type \( A \) is represented by declaring a method \( M \) to be of type \( \text{Unit}(A,a) \), the inductive unit type parameterized by \( A \) and \( a \). Then, with a coercion that maps any object of \( \text{Unit}(A,a) \) to \( a \), \( M \) stands for \( a \) in any context that requires an object of type \( A \), as expected. Besides the coercion concerning the unit type, our model also employs structural subtyping between record types in the framework of coercive subtyping. Based on these subtyping relations, the model can capture subtyping relationships between a class and its interface-type \[Hor02\], and between an interface-type and its sub-interface-type, not direct subtyping between a class and its subclass.

We here define a new object-oriented language, myJava\(^1\) that is a subset of Java. It is also a specification language which can represent class invariants and pre-and-post conditions for methods. Moreover, we develop a verification software called JaCo, which translates automatically myJava programs and their specifications to Coq codes as shown in Figure 1.

\(^1\)The BNF syntax and features of myJava language is defined in the author’s website(http://www.cs.rhul.ac.uk/home/seokhyun). Programs written in myJava are compiled by Java Platform SE 6
Our investigation now stands in need of two things. First, we are modeling myJava programs by using Coq as a dependently typed functional programming language. Secondly, we are then verifying specifications being satisfied by those programs such as invariants and pre-and-post conditions by using Coq as a proof assistant.

II. MODELING JAVA PROGRAMS IN COQ

In the pages which follow we shall develop such an investigation based on three simple classes Storage, NewStorage, and Manager shown in Figure 2 and whose source codes are given in Appendix A. We are interpreting these classes as record types by means of separating fields from methods. All Coq codes given in this paper have been automatically generated by JaCo.

A. Modeling the state of objects

The state of an object is the set of values that determine how the object reacts to method calls. An object uses instance fields, which are the data that the object needs to execute its methods, in order to store its state. Each object of a class has its own set of instance fields.

First, the fields of the class Storage are separately grouped into the record type of state. So, the state of the class Storage is interpreted as objects of the record type SStorage.

The state of the class NewStorage is then modeled with the structural coercive subtyping of record projection $\triangleright:$ because the field stock of the class Storage is implicitly included in the class NewStorage according to inheritance relationship.

The following coercion graph is therefore defined with the coercion function $c \equiv \text{label}\_storage\_1$ such that for all $x: \text{SNewStorage}$, $c(x): \text{SStorage}$, which means that any object of SNewStorage can be regarded as an object of SStorage.

$\text{SNewStorage} \stackrel{\ll}{\hookrightarrow} \text{SStorage}$

Continuously, we define the state of the class Manager with the record type SManager.

Finally, we define the universal state Omega being able to capture all states of the classes Storage, NewStorage, and Manager simultaneously.

The non-dependent record type Omega is very essential for functional interpretation of myJava programs because functional programming does not have an idea of changing stored values on the hardware memory. After Omega is decided, we will start to model methods of the classes. All methods of the classes in the context are modeled as taking an object of Omega as an implicit parameter and returning a new object of Omega if the state is changed. But, each method is interested only in the state of the class to which it belongs. This realizes the important notion of encapsulation in object-oriented programming, which means that a method can directly access only a state of the object to which it belongs.

The more important reason that Omega is needed for our interpretation is to handle the contra-variant\(^2\) problem occurring when constructing component-wise structural coercive subtyping between record types representing interface-types which have inheritance relationship. For example, if we had interpreted the method $\text{GetStock()}$ in the class Storage as a function $\text{GetStock}\_\text{Storage}$ of type $\text{SStorage} \rightarrow \text{nat}$ and the method $\text{GetStock()}$ which is implicitly inherited to the class NewStorage as a function $\text{GetStock}\_\text{NewStorage}$ of type $\text{SNewStorage} \rightarrow \text{nat}$, we would have met the contra-variant problem while modeling coercive subtyping between $\text{SNewStorage} \rightarrow \text{nat}$ and $\text{SStorage} \rightarrow \text{nat}$.

\(^2\)We say that $\rightarrow$ is a contravariant operator in its left argument because $A \rightarrow B$ is in the opposite sense for $A$, i.e. $A \rightarrow B < A' \rightarrow B'$ provided that $A' < A$ and $B < B'$.
It is because that although \texttt{SNewStorage} is a subtype of \texttt{SStorage}, the function \texttt{GetStock\_NewStorage} of type \texttt{SNewStorage \rightarrow nat} cannot be an object of the type \texttt{SStorage \rightarrow nat} due to the contra-variant. However, if we model both methods as a function of type \texttt{Omega \rightarrow nat}, the contra-variant problem will never occur.

\section*{B. Modeling member functions (methods) of objects}

We introduce useful terminologies for methods of a class. All methods written in myJava are classified as either an accessor or a mutator, or can be both, depending on what effect they have on their implicit parameter, Omega. Accessor methods access an object and return some information about it without changing the state of the object. In contrast, mutator methods modify and change the state of an object, but do not return a value - the return type should be void. It is needless to say that only mutator methods are modeled as returning a new object of \texttt{Omega}. For example, in the class \texttt{Storage}, the method \texttt{GetStock()} is an accessor and the method \texttt{Setting(int n)} is a mutator.

Each method consists of a sequence of statements\footnote{This paper does not mention while-loop and field-method-call statements, but those statements can be found in [Han10a]}. assignment statement, if-then-else statement, while-loop statement, implicit-method-call statement, explicit-method-call statement, or field-method-call statement. We are interpreting each statement as a function taking an object of \texttt{Omega} and returning a new object of \texttt{Omega}. Therefore, methods of a class are modeled as a composite function of a sequence of these statements. The implicit-method-call and explicit-method-call statements will be examined in detail later on.

Now, we are modeling methods of a class as parameterized inductive unit types and function fields in a dependent record type. A method \texttt{M} of a class is represented as an inductive unit type \texttt{Unit}(A,a) parameterized \texttt{A} and \texttt{a}, and then by a coercion that maps any object of \texttt{Unit}(A,a) to \texttt{a} in a context, which is of type \texttt{A}, stands for \texttt{M}.

\begin{align*}
\text{Unit} & : (A:\text{Type})(x:A)\text{Type} \\
\text{unit} & : (A:\text{Type})(x:A)\text{Unit}(A,x)
\end{align*}

where \text{unit}(A,a) is the only object of type \text{Unit}(A,a).

\begin{center}
\begin{tabular}{c}
\text{Inductive Unit (A:Set)(a:A):Set := unit : Unit A a.} \\
\text{Implicit Arguments Unit.} \\
\text{Implicit Arguments a.}
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c}
$\Gamma \vdash A : \text{Type}$ \\
$\Gamma \vdash a : A$ \\
$\Gamma \vdash \text{Unit}(A,a) <_{\xi(A,a)} A : \text{Type}$
\end{tabular}
\end{center}

where the non-structural coercion function \(\xi : (A:\text{Type})(a:A)(\text{Unit}(A,a))A\) is defined as \(\xi(A,a,x) = a\) for any \(x : \text{Unit}(A,a)\). We need to define the identity \(\text{Id}(A) = A\) on types to force Coq to accept \(\xi\)-coercion and to use type-casting as a trick to make it happen. We assume as a trick only for Coq that the identity is a coercion, where \(\text{Id}_A \equiv [x:A|x]

\begin{center}
\begin{tabular}{c}
\text{Definition ID (A:Set) : Set := A.} \\
\text{Coercion unit\_coercion (A:Set)(a:A) (_:Unit a) := a : ID a.}
\end{tabular}
\end{center}

Now, we are interpreting the class \texttt{Storage}\footnote{In Coq codes, \texttt{LessThan} is the boolean comparison operator of type \texttt{nat \rightarrow nat \rightarrow bool} on natural numbers.} where we note that unit types are defined implicitly, i.e. parameters can be omitted.

\begin{center}
\begin{tabular}{c}
\text{(* Modeling the method Storage\_CONST *)} \\
\text{Let Storage\_CONST\_Storage} \\
\text{: nat->Omega->Omega := fun(x : nat)(w : Omega) =>} \\
\text{let Assign1\_Storage\_CONST := fun (awl : Omega) =>} \\
\text{mk\_Omega ( mk\_SStorage x) } \\
\text{awl.(label\_newstorage) } \\
\text{awl.(label\_manager) ) } \\
\text{in ( Assign1\_Storage\_CONST w) .}
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c}
\text{(* Modeling the method GetStock *)} \\
\text{Let GetStock\_Storage} \\
\text{: Omega->nat:= fun(w : Omega) =>} \\
\text{w.(label\_storage).(stock)).}
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c}
\text{(* Modeling the method Setting *)} \\
\text{Let Setting\_Storage ( GetStock : Omega->nat ) :} \\
\text{nat->Omega->Omega= fun(n : nat)(w : Omega) =>} \\
\text{let If1\_Setting := fun (iw1:Omega) =>} \\
\text{if ( LessThan ( GetStock iw1) n) then } \\
\text{let Assign1\_Setting := fun (awl : Omega) =>} \\
\text{mk\_Omega ( mk\_SStorage n ) } \\
\text{awl.(label\_newstorage) } \\
\text{awl.(label\_manager) ) } \\
\text{in Assign1\_Setting iw1 } \\
\text{else iw1 } \\
\text{in ( If1\_Setting w ).}
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c}
\text{(* Define a record type representing all methods of} \\
\text{the class Storage *)} \\
\text{Record MStorage : Set := mk\_MStorage {}.
\text{label\_Storage\_CONST\_Storage : Unit Storage\_CONST\_Storage;}
\text{label\_GetStock\_Storage : Unit GetStock\_Storage;}
\text{label\_Setting\_Storage : Unit Setting\_Storage (label\_GetStock\_Storage : ID(Omega->nat))}.}
\end{tabular}
\end{center}

Here, we return to explaining \textit{implicit-method-call statement} and \textit{implicit-method-call expression} with the notion of an implicit parameter. The implicit parameter of a method is the object on which the method is invoked. The ‘this’ reference denotes the implicit parameter: for instance in the given example, \texttt{this.NewStorage\_CONST()} in the method \texttt{Restore()} of the class \texttt{Storage} is called an \textit{implicit-method-call statement} where ‘this’ refers an object of the class \texttt{NewStorage}, and \texttt{this.GetStock()} in the method \texttt{Setting(int n)} of the class \texttt{Storage} is called an \textit{implicit-method-call expression} where ‘this’ refers an object of the class \texttt{Storage}. These objects of the class \texttt{Storage} and \texttt{NewStorage} are not given explicitly by parameters in the method definition, so that they are called an implicit parameter of the methods. We note that implicit-method-call statement should be a mutator, and implicit-method-call expression an accessor.

Now, we are modeling the set of all methods in a class as a record type. In particular, if a method includes implicit-method-call statement or implicit-method-call expression, the set of methods is modeled as a dependent record type with
the help of the non-structural coercion function $\xi$ defined previously. As we have seen in the above, the method
\textbf{Setting}(int n) is modeled as a function $\text{Setting\_Storage}$ being parameterized by another function of type $\Omega \rightarrow \text{nat}$ in order to model the implicit-method-call expression $\text{this.GetStock()}$. This parameterized function is integrated in a unit type by applying the function $\text{Setting\_Storage}$ to the field $\text{label\_GetStock\_Storage}$ defined earlier in the record type $\text{MStorage}$. However, here the field $\text{label\_GetStock\_Storage}$ in the record type $\text{MStorage}$ is not a function since it is of type $(\text{Unit GetStock\_Storage})$. So, we should have a doubt of how this application can be well-typed.

The reason of this application being well-typed is that $\text{label\_GetStock\_Storage}$ is an object of type $(\text{Unit GetStock\_Storage})=\text{GetStock\_Storage}$ of type $\Omega \rightarrow \text{nat}$.

Finally, we are modeling the class $\text{Storage}$ as merging the record type of state $\text{SStorage}$ and the record type of methods $\text{MStorage}$ with the structural coercion $c_1 \equiv \text{label\_mstorage}$. $\text{Storage} \triangleleft c_1 \text{MStorage}$

\begin{verbatim}
/* Modeling the class Storage as a record type */
Record Storage : Set := mk_Storage { 
  label_mstorage : SStorage ;
  label_sstorage : SStorage ;
  label_getstock_newstorage : SStorage ;
}.

C. Modeling inheritance relationship

Inheritance is a way of extending existing classes for code reuse. The following is interpretation of the class $\text{NewStorage}$ where we can see that the methods $\text{InterfaceStorage}$ and $\text{GetStock()}$ defined in the class $\text{Storage}$ are implicitly included in the class $\text{NewStorage}$, and the method $\text{Setting}(int n)$ is overridden.

\begin{verbatim}
/* Modeling the method $\text{Setting\_CONST}(int x)$ inherited from the class $\text{Storage}*/
Let Storage\_CONST\_NewStorage := fun (w : Omega) => 
  (mk_Storage awl.(label_storage) 
    (mk\_NewStorage (mk\_SNewStorage x) 
      awl.(label\_newstorage),(saleProduct)) 
    awl.(label\_manager) 
  ) .

/* Modeling the method $\text{GetStock\_NewStorage}$ inherited from the class $\text{Storage}*/
Let GetStock\_NewStorage := fun (w : Omega) => 
  (wl.(label\_newstorage),(stock)) .

/* Modeling the new method $\text{NewStorage\_CONST}$ */
Let NewStorage\_CONST\_NewStorage := fun (awl : Omega) => 
  (mk_Storage awl.(label\_storage) 
    (mk\_NewStorage (mk\_SNewStorage) x 
      awl.(label\_newstorage),(saleProduct)) 
    awl.(label\_manager) 
  ) .

/* Modeling the overridden method $\text{Setting\_NewStorage}$ */
Let Setting\_NewStorage := fun (GetStock : Omega->nat) => 

let If1_Setting := fun (iw1:Omega) =>
  if (LessThan (GetStock iw1) n) then
    let Assign1_Setting := fun (aw1 : Omega) =>
      (mk\_Omega awl.(label\_storage) 
        (mk\_SNewStorage (awl.(label\_newstorage),(stock)) 
          (GetStock aw1) ) 
        awl.(label\_manager) )
    in let Assign2_Setting := fun(aw2 : Omega) =>
      (mk\_Omega aw2.(label\_storage) 
        (mk\_NewStorage (mk\_SStorage n) aw2.(label\_newstorage),(saleProduct)) 
        aw2.(label\_manager) ) 
    in (Assign2_Setting (Assign1_Setting iw1)) else iw1 
  in (If1_Setting w).

 Now we are modeling inheritance between the classes $\text{Storage}$ and $\text{NewStorage}$ as subtyping relation. We should note that we are here not modeling direct inheritance relationship between the class $\text{Storage}$ and the class $\text{NewStorage}$, but modeling indirect inheritance relationship between them by way of their interface-types which have inheritance relationship. It means that if interface-types have inheritance relationship, then the classes implementing these interface-types have also inheritance relationship. The purpose is here to outline and elaborate indirect inheritance relationship in the frame of coercive subtyping.

 The interface-type of a class can be defined from the signature of the methods which the class has. The signature includes information about a method: its name, its parameter types, and its return-type. For instance, as we see the class diagram in Figure 2, we have defined the classes $\text{Storage}$ and $\text{NewStorage}$ implementing the interface-types $\text{InterfaceStorage}$ and $\text{InterfaceNewStorage}$ respectively. We interpret the interface-type $\text{InterfaceStorage}$ as the record type $\text{InterfaceStorage}$, and then define the component-wise coercion function between the record type $\text{MStorage}$ and the
record type InterfaceStorage.

\[
\begin{align*}
\text{Record InterfaceStorage : Set := mk\_InterfaceStorage (}
& \quad \text{i\_label\_Storage\_CONST\_Storage : nat->Omega->Omega;}
& \quad \text{i\_label\_GetStock\_Storage : Omega->nat;}
& \quad \text{i\_label\_Setting\_Storage : nat->Omega->Omega .}
\end{align*}
\]

As a result, we can see that the relation between the class Storage and the interface-type InterfaceStorage is interpreted as subtyping relation by the composite function \(c_2 \circ c_1\) where \(c_1 = \text{label}\_m\text{storage},\) and \(c_2 = \text{MStorage}\_\text{InterfaceStorage}.

\[
\text{Storage} \triangleleft c_1 \text{MStorage} \triangleleft c_2 \text{InterfaceStorage} \triangleleft c_2 \circ c_1 \text{InterfaceStorage}
\]

Likewise, we are modeling the relation between the class NewStorage and the interface-type InterfaceNewStorage as subtyping relation by the composite function \(c_4 \circ c_3\) where \(c_3 = \text{label}\_m\text{newstorage},\) and \(c_4 = \text{MNewStorage}\_\text{InterfaceNewStorage}.

\[
\text{NewStorage} \triangleleft c_3 \text{MNewStorage} \triangleleft c_4 \text{InterfaceNewStorage} \triangleleft c_4 \circ c_3 \text{InterfaceNewNewStorage}
\]

Subsequently, we can interpret inheritance relationship between the interface-types InterfaceNewStorage and InterfaceStorage as subtyping relation with the coercion function \(c_5\) where \(c_5 = \text{InterfaceNewStorage}\_\text{InterfaceStorage}.

\[
\text{InterfaceNewStorage} \triangleleft c_5 \text{InterfaceStorage}
\]

Proceeding from what we have said above, indirect inheritance relationship shown in the class diagram of Figure 2 is eventually modeled as the coercion graph shown in Figure 3.

\[
\text{Storage} \triangleleft c_1 \text{MStorage} \triangleleft c_2 \text{InterfaceStorage} \triangleleft \sqrt{3}
\]

Fig. 3: Coercion graphs

**Remark** We should note that we could not define a coercion function between the record types Storage and NewStorage, or between the record types MStorage and MNewStorage. In other words, we could not model the direct inheritance relationship between the classes Storage and NewStorage in the frame of coercive subtyping without their interface-types. If we were to define coercion functions, we had confronted a coercion coherence problem.

**D. Modeling Subtype polymorphism and Dynamic dispatch**

Subtype polymorphism and dynamic dispatch with interface-types provide one of ways of accessing other objects with possible side effects, which means that the state of the referred object may be changed. We will capture these mechanisms in type theoretic models with the frame of coercive subtyping shown in Figure 3. As we see the class Manager in Figure 2 and its source codes in the appendix A, it would exchange messages with the classes Storage and NewStorage.

Here, we need to define the notion of the explicit-method-call expression and explicit-method-call statement with a explicit parameter whose type is an interface-type. This parameter is an object which a method is explicitly taking as an argument when the object is invoked. For example, \(x.\text{GetStock}()\) in the method \(\text{GetTotalStock(InterfaceStorage} x)\) of the class Manager is an explicit-method-call expression, and \(x.\text{Setting(extra)}\) in the method \(\text{Supply(InterfaceStorage} x)\) of the class Manager is an explicit-method-call statement. Consider the following assignment statements and function applications in Java programing.

First, two variables \(i\text{Storage}_A\) and \(i\text{Storage}_B\) are declared as objects of the interface-type InterfaceStorage. Accordingly, an instance of class Storage can be assigned to the

5 Coherence essentially says that the coercions between any two types are unique up to the computational equality. Coherence is a crucial property of coercion that guarantees the logical consistency and the nice meta-theoretic properties of the subtyping extensions.
variable \(iStorage_A\) because the class \(Storage\) implements \(InterfaceStorage\). Likewise, an instance of class \(NewStorage\) can be assigned to the variable \(iStorage_B\) because the class \(NewStorage\) implements \(InterfaceNewStorage\) inheriting from \(InterfaceStorage\).

Secondly, the objects \(iStorage_A\) and \(iStorage_B\) can be passed to the method \(Supply(InterfaceStorage x)\) of the class \(Manager\) that expects an instance of a class implementing \(InterfaceStorage\) or an instance of a class implementing \(InterfaceNewStorage\) which inherits from \(InterfaceStorage\). These codes would be illegal in functional programming language, because the interface-types \(InterfaceStorage\) and \(InterfaceNewStorage\) do not match. However, in object-oriented programming, these codes are made legal by \(Subtype Polymorphism\):

If \(NewStorage\) is a subclass of \(Storage\) by way of the interface-type \(InterfaceNewStorage\) being a subtype of the interface-type \(InterfaceStorage\), and \(o\) is an object of \(NewStorage\), then \(o\) is an object of \(Storage\).

With regard to subtype polymorphism, we introduce the notion of Dynamic Dispatch by determining the meaning of the explicit-method-call statement \(x.Setting(extra)\) during the invocation of the method \(Supply(InterfaceStorage x)\) of the class \(Manager\). If the method \(Supply(InterfaceStorage x)\) is applied to an object of class \(Storage\), then \(x.Setting(extra)\) executes the code of \(Setting(int n)\) from the class \(Storage\). Analogously if the method \(Supply(InterfaceStorage x)\) is applied to an object of the class \(NewStorage\), then \(x.Setting(extra)\) executes the overridden method \(Setting(int n)\) from the class \(NewStorage\).

The interpretation for the class \(Manager\) is given as follows.

<table>
<thead>
<tr>
<th>Method</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GetTotalStock)</td>
<td>(f \in \text{Manager}.GetTotalStock)</td>
</tr>
<tr>
<td>(GetCheckTotal)</td>
<td>(f \in \text{Manager}.GetCheckTotal)</td>
</tr>
<tr>
<td>(ClearExtra)</td>
<td>(f \in \text{Manager}.ClearExtra)</td>
</tr>
<tr>
<td>(Manager_CONST)</td>
<td>(f \in \text{Manager}.Manager_CONST)</td>
</tr>
</tbody>
</table>

We can see that the methods \(GetTotal\_Stock(InterfaceStorage x)\) and \(Supply(InterfaceStorage x)\) are modeled as the functions \(GetTotal\_StockManager\) and \(Supply\_Manager\) taking an object of the record type \(InterfaceStorage\) respectively. All explicit-method-call expressions and explicit-method-call statements in these methods are modeled as record field-selection.

### III. VERIFYING SPECIFICATIONS OF JAVA PROGRAMS IN COQ

The verification of myJava programs \(Storage\), \(NewStorage\) and \(Manager\) has been done in the proof assistant Coq based on the previously defined type-theoretic models corresponding to those programs. For instance, we can show that, for the class \(NewStorage\), it is an invariant that the \(stock\) value is always greater than the \(saleProduct\) value. To put it more concretely, a class invariant is a statement about an object that is preserved by every mutator provided that a caller respects all preconditions.

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Invariant_Setting_NewStorage)</td>
<td>forall ((r : NewStorage) (n)), True -&gt; ((s.(label_newstorage).saleProduct) &lt; (s.(label_newstorage).stock))</td>
</tr>
</tbody>
</table>

\[\text{Proof:}\]

<table>
<thead>
<tr>
<th>Proof</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{intro})</td>
<td>(r \in n \text{H1})</td>
</tr>
<tr>
<td>(\text{elim})</td>
<td>((\text{le}_lt_dec n s).(label_newstorage).stock))</td>
</tr>
<tr>
<td>(\text{apply})</td>
<td>(\text{if}_false)</td>
</tr>
<tr>
<td>(\text{exact})</td>
<td>(\text{H1})</td>
</tr>
<tr>
<td>(\text{apply})</td>
<td>(\text{LessThan_false})</td>
</tr>
<tr>
<td>(\text{exact})</td>
<td>(\text{H2})</td>
</tr>
<tr>
<td>(\text{intro})</td>
<td>(\text{H2})</td>
</tr>
<tr>
<td>(\text{apply})</td>
<td>(\text{if}_true)</td>
</tr>
<tr>
<td>(\text{apply})</td>
<td>(\text{LessThan_true})</td>
</tr>
<tr>
<td>(\text{exact})</td>
<td>(\text{H2})</td>
</tr>
</tbody>
</table>

Qed.

\[\text{Theorem Invariant\_Restore\_NewStorage}\]

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(forall)</td>
<td>((r : NewStorage) (s)), True -&gt; ((s.(label_newstorage).saleProduct) &lt; (s.(label_newstorage).stock)))</td>
</tr>
</tbody>
</table>
Furthermore, we formulate pre-and-post conditions which can be regarded as contractual terms between a method and its caller. The method promises to fulfill the post-condition for all states that fulfill the pre-condition. The caller promises never to call the method with states which do not satisfy the pre-condition. For example, if we call the method Supply(InterfaceStorage x) of the class Manager with the state extra that is greater than zero, the method should definitely return a new state where extra is equal to zero.

IV. CONCLUSION

This article has attempted to sketch out how to represent object-oriented programs in the intensional type theory and Coq. As we have seen, the main purpose has been to explore that the combination of dependent record types and coercive subtyping provides a strong modeling mechanism for verification of Java programs in type theory. We have used the proof assistant Coq as a platform of implementing domain-specific reasoning tools for studying automated generations of Coq-models and specifications of Java programs. With such a tool support, a case study has been done with the myJava programs given in Appendix A in order to show that the type-theoretic encoding can correctly capture the important object-oriented features such as encapsulation, inheritance, subtype polymorphism and dynamic dispatch. I hope the outcome of the present paper will be a step toward a richer and more inclusive understanding of object-oriented programs in type-theoretic models.

REFERENCES


interface InterfaceStorage
{
    public void Storage_CONST(int x);
    public void Setting(int n);
}

class Storage implements InterfaceStorage
{
    protected int stock;

    //Initializing the field like a constructor
    public void Storage_CONST(int x) {
        stock = x;
    }

    public int GetStock() {
        return stock;
    }

    public void Setting(int n) {
        //implicit-method-call expression
        if (this.GetStock() < n) 
            stock = n;
    }

    interface InterfaceNewStorgre extends InterfaceStorage
    {
        public void NewStorage_CONST();
        public void GetStock();
        public void Restore();
    }

    class NewStorgre extends Storage
    implements InterfaceNewStorgre
    
    /* INVARIANT [[ saleProduct < stock ]] */
    protected int saleProduct;

    public void NewStorage_CONST()
public void Setting(int n) //overriding the method
{
    if (this.GetStock() < n)
    {
        //implicit-method-call expression
        saleProduct = this.GetStock();
        stock = n;
    }
}

public void Restore()
{
    stock = saleProduct;
    this.NewStorage_CONST(); //implicit-method-call statement
}

class Manager
{
    protected int checkTotal;
    protected int extra;

    public void Manager_CONST(int x, int y)
    {
        checkTotal = x;
        extra = y;
    }

    public void ClearExtra()
    {
        extra = 0;
    }

    public int GetCheckTotal()
    {
        return checkTotal;
    }

    public void GetTotalStock(InterfaceStorage x)
    {
        //explicit-method-call expression
        checkTotal = checkTotal + x.GetStock();
    }

    public void Supply(InterfaceStorage x)
    { /* PRECONDITION [[ 0 < extra ]] */
        /* POSTCONDITION [[ extra == 0 ]] */
        x.Setting(extra); //explicit-method-call statement
        this.ClearExtra(); //implicit-method-call statement
    }
}