VERIFICATION OF JAVA PROGRAMS IN TYPE THEORY WITH DEPENDENT
RECORD TYPES AND COERCIVE SUBTYPING

A THESIS SUBMITTED TO ROYAL HOLLOWAY, UNIVERSITY OF LONDON
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE DEPARTMENT OF COMPUTER SCIENCE

Author: Seokhyun Han

Supervisor: Prof. Zhaohui Luo

October 2010
Declaration of Authorship

I ______________________ hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:____________________
Date:____________________
Abstract

This thesis describes research on a functional interpretation of object-oriented programs in the intensional type theory UTT with dependent record types and coercive subtyping. In other words, it presents a functional analysis of constructing Java programs and their precise specifications in the Coq system. Here, we are using Coq only for the simulation of modeling Java programs in UTT, because Coq supports coercive subtyping and a macro for dependent record types.

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 of Java programs enjoys desirable subtyping relationships that correctly capture the important object-oriented features such as encapsulation, inheritance, reuse, 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 properties of Java programs in Coq.

The current object-oriented languages are closely linked to the underlying hardware, in the sense that programming is based on the idea of changing stored values. In contrast, functional programming languages promote a more abstract style of programming, based on the idea of applying functions to arguments. Moving to this higher-level leads to considerably simpler programs, and support a number of powerful new ways to structure and reason about programs. So, research on a functional interpretation of object-oriented programming languages can be in interest.

Most importantly, we will model Java programs by dwelling on intensional type theories without extensional features, and lead to the machine-supported verification of Java programs. When we have a suitable intensional notion of computational equality, it is easier to obtain a good operational understanding of the language. We shall use dependent record types and structural subtyping for them in the framework of coercive subtyping with the notion of intensional
manifest fields.

This thesis is partly theoretical study of the meaning of object-oriented programs, in other words, the functional model of object-oriented programs is the denotational semantics of them. So, one of the consequences about this study is to design a software which can translate Java programs into Coq codes, in order to verify specific properties of Java programs. Our aim is to add a new perspective to the debate how Java programs can be interpreted in type-theoretic models by modelling them in Coq, and how the properties of Java programs can be proved with regard to class invariants and pre-and-post conditions for methods.
I thank my supervisor Professor Zhaohui Luo for giving me the opportunity to work in this interesting project, for his prompt and continuous support, and well-founded and helpful advice.

I wish to acknowledge the helpful comments and advice of Types group members (Dr Robin Adams, Yangyue Feng, Tao Xue, and Ha V. Bui). I also thank Professor Qiang Shen and Dr Iman Poernomo for being my referees.

I am greatly indebted to my wife (Mira Choi) and my parents for their encouragement and praying for me.
# Contents

Declaration of Authorship .......................................................... 1  
Abstract .................................................................................. 2  
Acknowledgments ....................................................................... 4  

1 Introduction  ......................................................................... 7  
1.1 Main aims and motivations ............................................. 7  
1.2 Background work ............................................................. 10  
   1.2.1 Type theory and proof assistant Coq ......................... 10  
   1.2.2 Modeling object-oriented programs in type theory .... 12  
1.3 Major contribution ............................................................. 13  
1.4 Structure of the thesis ....................................................... 17  

2 Verification of myJava programs in Coq ................................. 19  
2.1 Modeling myJava programs in Coq ................................... 23  
   2.1.1 Modeling object state ............................................... 23  
   2.1.2 Modeling methods(member functions) of objects ....... 27  
   2.1.3 Modeling inheritance ............................................... 32  
   2.1.4 Modeling subtype polymorphism and dynamic dispatch . 37  
2.2 Verifying specifications of myJava programs in Coq ........... 53  

3 UTT .................................................................................... 57  
3.1 The logical framework LF ............................................... 57  
3.2 Specifying UTT in LF ....................................................... 60  
3.3 Coercive subtyping ........................................................... 62  
3.4 Dependent record types .................................................... 63  
   3.4.1 Terms ................................................................. 64
4 Semantics of functional interpretation 67
   4.1 The state of objects 67
   4.2 Modeling a class as a dependent record type 72
      4.2.1 Modeling statements 72
      4.2.2 Modeling methods 74
      4.2.3 Modeling classes 77
   4.3 Inheritance in coercive subtyping 77
   4.4 Subtype polymorphism and dynamic dispatch with interface-types 80
   4.5 Modeling fields of interface-type 81
   4.6 Modeling a while-loop statement 88

5 Case study 90

6 Conclusion 97
   6.1 Discussion and Future work 97
   6.2 Summary 101

Bibliography 103

A Coq codes for case study 109

B myLibrary.v 142

C The BNF syntax of myJava in JavaCC 144

D Inference rules in LF 147
Chapter 1

Introduction

1.1 Main aims and motivations

In this thesis we are taking up the theme [Han10] of constructing object-oriented programs and their precise specifications in UTT [Luo94]. The purpose is to take a closer look at whether we can interpret 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 [Luo09] and coercive subtyping [Luo97] [Luo99] [Luo05] [Bai99]. The dependent type theories such as 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 [Coq] [BC04] and Agda [Agd], which are used effectively for formalization of mathematics and verification of programs.

We define a new object-oriented language, myJava\textsuperscript{1}, which 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. Coq is here used for simulation of modeling myJava programs in UTT, because it supports coercive subtyping [Saï97] and a macro for dependent record types\textsuperscript{2}. In this overall perspective, we can say that

\footnotesize
\textsuperscript{1}The syntax of myJava is defined in Figure 1.2 and Figure 1.3. Programs written in myJava are compiled by Java Platform SE 6 [java]

\textsuperscript{2}In Coq, dependent record types are actually implemented as Σ-types with labels defined as global name.
this translation is to define the type-theoretical semantics of Java programs.

We describe a type-theoretic model of myJava 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. As our approach is based on [Luo09], 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, and between an interface-type and its sub-interface-type, not direct subtyping between a class and its subclass.

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, the functional programming paradigm [Bar84] [Bar92] [Tho91] [Mit96] [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.

There has been a lot of work on functional modeling of object-oriented programs, which has tremendously improved our understanding of object-oriented paradigm [GM94] [AC96] [Pie02]. Pierce and Turner [PT94] use impredicative existential types of $\mathcal{F}_{\leq}$ in order to model objects. However, the attempts have also met with some difficulties. For example, when modeling object-oriented programs in type systems, people were led to employ some powerful type-theoretic features such as bounded quantification [CW85] and singleton types [Asp95], which are meta-theoretically difficult and sometimes even lead to outright undecidability of the underlying type theory. Partly because they are arguably problematic and not well understood, they are not implemented in the current theorem provers based on intensional type theories. As a consequence, a model based on such features does not directly lead to the machine-supported verification of object-oriented programs.

Most importantly, we will model myJava programs by dwelling on intensional type theories without extensional features. When we have a suitable intensional notion of computational equality, it is easier to obtain a good operational understanding of the language. Being intensional, the underlying type theory does not have the strong extensional equality [ML84] or the $\eta$-like equality rules for inductive types. When record types are considered in this thesis, they do not have the manifest fields as studied in [HL94] [Ler94] [Pol02] [CPT05], which represent an extensional construction in the context of record types. Put in another way, we stay with the intensional type theory without employing the extensional features that are problematic or meta-theoretically difficult. Because of this, modeling classes as types require a special treatment, as sketched with a unit type. We shall use dependent record types and structural subtyping for them in the framework of coercive subtyping with the notion of intensional manifest fields [Luo09]. Furthermore, since the underlying type theory is implemented in the current proof assistants such as Coq and Agda, the machine supported verification of object-oriented programs can be done directly based on the type-theoretic models.

At the same time, 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
CHAPTER 1. INTRODUCTION

functions, class instances (or objects) as records, and method calls as record field-selection. 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 state.

In this respect, it would be also interesting to link and compare our work with the existing efforts in verification of object-oriented programs such as ESC/Java2 [FLL02], KeY [BHS07], Krakatoa [MPMU04], and LOOP [BJ01]. They have modeled object-oriented programs by deep embedding which translates from the concrete syntax that a programmer would actually write, to the abstract syntax and semantics tree that are defined in inductive data types.

1.2 Background work

1.2.1 Type theory and proof assistant Coq

Type theory can be seen as a programming language, a logical system, or a mixture of the two. In other words, a type theory can be seen as a programming language, with the terms as programs and the types as specification. Type theory is to study computational and logical languages for computer science with a coherent treatment, i.e. computation and logical inference in a single formalism.

<table>
<thead>
<tr>
<th>Type theory</th>
<th>Programming/Computation</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Specification</td>
<td>Propositions</td>
</tr>
<tr>
<td>Term(object)</td>
<td>Program</td>
<td>Proofs</td>
</tr>
<tr>
<td>Reduction</td>
<td>Execution of programs</td>
<td>→ elimination</td>
</tr>
<tr>
<td>Simple Type Theory</td>
<td></td>
<td>Propositional Logic</td>
</tr>
<tr>
<td>Dependent Type Theory</td>
<td></td>
<td>Predicate Calculus</td>
</tr>
</tbody>
</table>

Type theory (or Intuitionistic type theory, or constructive type theory) is a logical system and a set theory based on the principles of mathematical constructivism. Martin-Löf’s type theory was introduced by Per Martin-Löf, Swedish mathematician and philosopher, in 1972. Martin-Löf had modified his proposal a few times [ML75] [ML82] [ML84] [NPS94]: proposing first impredicative and then predicative, and first extensional and then intensional variants of type theory.

Intuitionistic type theory is based on a certain analogy or isomorphism between propositions and types: a proposition is identified with the type of its proofs. This identification is usually called the Curry-Howard isomorphism [How80],
which was originally formulated for propositional logic and simply typed lambda calculus. Just as a set can be empty, a singleton, finite, or infinite, a proposition may be inhabited by zero, one, many, or infinitely many proofs. Each inhabitant of a proposition $P$ is a different way of giving evidence for $P$. If there is none, then $P$ is not provable. If there are many, then $P$ has many different proofs, which is called proof irrelevance. Type theory extends this identification to predicate logic by introducing dependent types, which is types containing values. Type theory internalizes the interpretation of intuitionistic logic proposed by Brouwer, Heyting and Kolmogorov, the so called BHK interpretation [Hey71]. The types of type theory play a similar role as sets in set theory but functions definable in type theory are always computable. The normalization property is a key basis for understanding the internal logic and for the decidability results.

The Coq system [BC04], whose underlying theory is the Calculus of Inductive Constructions [CH88] [Hue87], is a computer tool for proving theorems which may concern mathematics or software verification. Coq language is very powerful and expressive both for reasoning and for programming, so that it can be seen as not only a functional programming language but also a set of tools for stating logical assertion about the behavior of programs and assembling evidence of their truth. In Coq’s programming language, almost nothing is built in - not even booleans or numbers, instead it provides powerful tools for defining new types of data and functions that process and transform them. Like its built-in programming language, Coq’s built-in logic is extremely small: universal quantification($\forall$) and implication($\to$) are primitive, but all other familiar logical connectives, which are conjunction, disjunction, negation, existential quantification, and even equality, can be defined using just these primitives together with the inductive definition facility. Now, Coq is an industrial-strength proof assistant: it is becoming more and more popular in both software and especially hardware industries.

Moreover, we can ask Coq to extract from a definition or a theorem, a program in a more conventional programming language such as OCaml [oca] with a high performance compiler. This facility is very important to computer scientists who have belief that correct programs can be equivalent to proofs of specification. It is because it gives them a way to construct fully certified programs in mainstream languages. Indeed, this is one of the main uses for which Coq was developed.
1.2.2 Modeling object-oriented programs in type theory

Some recent research has been directed at combining dependent type theory with object-oriented programming. Setzer [Set07] treated object-oriented programming in the context of predicative dependent type theory, which makes use of the expressive power of dependent types. Our type-theoretic model is given in the intensional type theory UTT [Luo94]. In our model construction, classes and their interface-types are interpreted as dependent record types. Since each method entry in a class is interpreted to define a specific function, it is not easy to interpret the class as a type in the intensional type theory, although this is not a problem in an extensional type theory. We shall review how this can be done in a type theory with strong extensional constructs.

To model a class as a record type, one of the key questions is whether we can interpret a method entry in a class as such a manifest field. Here are some examples of extensional constructs in type theory which can be used to do this.

**Manifest fields** Manifest fields in record types are studied in [HL94] [Ler94] [Pol02] [CPT05]. If we have a record type < ...m = a : A... > with a manifest field ‘m = a : A’, m is computationally equal to a. In such a case, one can simply interpret a method entry in a class as such a manifest field. However, manifest types represent a strong form of extensional η-equality in the context of record types and make the meta-theoretic studies rather difficult.

**Strong extensional equality** The extensional equality was first introduced in Martin-Löf’s extensional type theory(ETT) [ML84] and adopted by the NuPRL system [C+86]. In ETT, the propositional equality $Eq(A,a,b)$ is equivalent to the judgemental equality in the sense that the following is an inference rule

$$\Gamma \vdash p : Eq(A,a,b)$$

$$\Gamma \vdash a = b : A$$

In such a case, one can interpret a method in a class as two entries in a record type: $m : A$ followed by $p : Eq(A,m,a)$, where the second entry guarantees that $m$ is judgementally equal to $a$ [CH00]. As is well known,
the problems of ETT including its undecidability are due to the existence of such an unmanageable equality $Eq$.

**Singleton types** Singleton types are studied in [Hay94] [Asp95]. For singleton type $\{a\}_A$, where $a : A$, the following holds

$$
\Gamma \vdash M : \{a\}_A \\
\Gamma \vdash M = a : A
$$

In such a case, a method in a class can be modeled as an entry $m : \{a\}_A$ of a record type. Of course, singlton types are strong $\eta$-like constructs and difficult in meta-theory [Cou02].

Do we need such extensional constructs? Can a class be modeled as a type in an intensional type theory? This has been thought as impossible, partly because that in an intensional type theory, the propositional equality is not equivalent to the computational equality in a non-empty context. However, as shown below, with the help of coercive subtyping, we can interpret classes as types in an intensional type theory.

$$
\Gamma \vdash A : Type \\
\Gamma \vdash a : A \\
\Gamma \vdash Unit(A,a) \leq_{\xi(A,a)} A : Type
$$

where the coercion $\xi : (A:Type)(a:A)(Unit(A,a)).A$ is defined as $\xi(A, a, x) = a$ for any $x : Unit(A, a)$. Now, the key idea in modeling classes as types is to interpret a method as a field

$$
m : Unit(A,a)
$$

in a record type, where $A$ interprets the type of the method and $a$ the method itself. When $m$ occurs in a term where an object of type $A$ is asked, it will be coerced into $\zeta(A, a, m) = a$, as intended [Luo09].

### 1.3 Major contribution

**Defining a new object-oriented language - myJava**

In this thesis, we define a new small-sized object-oriented language, myJava, which is a subset of Java language. myJava is one of class-based languages which form the mainstream of object-oriented languages, such as Simula, Smalltalk.
Program := (ClassDecl | InterfaceTypeDecl
| ClassDeclExt | InterfaceTypeDeclExt)^
ClassDecl := class id_class (ClassInvariant)?
{ (FieldVarDecl)* (MethodDecl)*
| class id_class implements id_interface
(ClassInvariant)? { (FieldVarDecl)* (MethodDecl)*
}
ClassDeclExt := class id_class extends id_class (ClassInvariant)?
{ (FieldVarDecl)* (MethodDecl)*
| class id_class extends id_class implements id_interface
(ClassInvariant)? { (FieldVarDecl)* (MethodDecl)*
}
InterfaceTypeDecl := interface id_interface { (MethodSignature)*
}
InterfaceTypeDeclExt := interface id_interface extends id_interface
{ (MethodSignature)*
}
FieldVarDecl := protected type field id_field;
MethodDecl := public type return id_method((Parameter)*
(Pre&PostConditions)?
{ (LocalVarDecl)* (Statement)* (return id ;)?
}
MethodSignature := public type return id_method((Parameter)*

LocalVarDecl := type local id_local ;
Parameter := type para id para

type field := int | boolean | id_interface
type local := int | boolean
type return := int | boolean | void | id_interface
type para := int | boolean | id_interface
ClassInvariant := /*INVARIANT [[ Expression ]] */
Pre&PostConditions := /*PRECONDITION [[ Expression ]] */
/* POSTCONDITION [[ Expression ]] */

Figure 1.2: The Extended BNF syntax of myJava (Part I)
\[
\begin{align*}
\text{Statement} & \quad := \quad \text{Assignment} \mid \text{IfThenElse} \mid \text{WhileLoop} \\
\text{Assignment} & \quad := \quad \text{id}_{\text{field}} = \text{Expression} ; \\
& \quad \mid \quad \text{id}_{\text{local}} = \text{Expression} ; \\
& \quad \mid \quad \text{id}_{\text{field}} = \text{id}_{\text{para}}\cdot\text{id}_{\text{method}}((\text{Expression})^*); \\
\text{IfThenElse} & \quad := \quad \text{if} \ (\text{Expression}) \ (\text{Statement})^* \\
& \quad \quad \quad \quad \quad \quad \text{(else} \ (\text{Statement})^*)? \\
\text{WhileLoop} & \quad := \quad \text{while} \ (\text{Expression}) \ (\text{Statement})^* \\
\text{MethodCallStatement} & \quad := \quad \text{ImplicitMethodCallState} \\
& \quad \quad \mid \quad \text{ExplicitMethodCallState} \\
& \quad \quad \mid \quad \text{FieldMethodCallState} \\
\text{ImplicitMethodCallState} & \quad := \quad \text{this}.\text{id}_{\text{method}}((\text{Expression})^*) ; \\
\text{ExplicitMethodCallState} & \quad := \quad \text{id}_{\text{para}}.\text{id}_{\text{method}}((\text{Expression})^*) ; \\
\text{FieldMethodCallState} & \quad := \quad \text{id}_{\text{field}}.\text{id}_{\text{method}}((\text{Expression})^*) ; \\
\text{Expression} & \quad := \quad \text{ArithExp} \mid \text{RelationExp} \mid \text{BooleanExp} \\
& \quad \quad \mid \quad \text{MethodCallExpression} \mid \text{id} \\
\text{MethodCallExpression} & \quad := \quad \text{ImplicitMethodCallExp} \\
& \quad \quad \mid \quad \text{ExplicitMethodCallExp} \\
& \quad \quad \mid \quad \text{FieldMethodCallExp} \\
\text{ImplicitMethodCallExp} & \quad := \quad \text{this}.\text{id}_{\text{method}}((\text{Expression})^*) \\
\text{ExplicitMethodCallExp} & \quad := \quad \text{id}_{\text{para}}.\text{id}_{\text{method}}((\text{Expression})^*) \\
\text{FieldMethodCallExp} & \quad := \quad \text{id}_{\text{field}}.\text{id}_{\text{method}}((\text{Expression})^*) \\
\text{ArithExp} & \quad := \quad \text{IntegerNumeral} \mid \text{ArithExp} \ast \text{ArithExp} \\
& \quad \quad \mid \quad \text{ArithExp} + \text{ArithExp} \mid \text{ArithExp} - \text{ArithExp} \\
\text{RelationExp} & \quad := \quad \text{ArithExp} < \text{ArithExp} \mid \text{ArithExp} \leq \text{ArithExp} \\
& \quad \quad \mid \quad \text{ArithExp} == \text{ArithExp} \\
\text{BooleanExp} & \quad := \quad \text{true} \mid \text{false} \mid \text{BooleanExp} \&\& \text{BooleanExp} \\
& \quad \quad \mid \quad \text{BooleanExp} \mid \text{BooleanExp} \mid \text{BooleanExp} \\
\text{IntegerNumeral} & \quad := \quad 0 \mid 1 \mid 2 \mid \ldots
\end{align*}
\]
[Kay93], C++ and Java, and are centered around the notion of class as descriptions of objects. The most different feature from other class-based languages is that myJava is also a specification language which supports the way of specifying properties of object-oriented programs. It can represent class invariants and pre- and postconditions for methods with the idea of Hoare logic [Hoa69] and JML [GY], which is a formal behavioral interface specification language for Java.

The syntactic specification of myJava is shown in Figure 1.2 and Figure 1.3, and is built in the BNF notation extended with the following quantifier syntax:

- $(\ )^*$: zero or more
- $(\ )^?$: zero or one
- $(\ )^+$: one or more

In this thesis, to be more informative, we shall use $id_{\text{class}}$, $id_{\text{interface}}$, $id_{\text{field}}$, $id_{\text{method}}$ and $id_{\text{local}}$ for identifiers indicating names of classes, interface-types, fields, methods and local variables, respectively. A few important restrictions$^3$ of using myJava can be made clear by a close reading of Appendix C.

**Defining the semantics of myJava programs in type theory**

We are to define the denotational semantics [G.W96] [Luo09] [Hen93] [CP94] [Fil03] of myJava programs in UTT with dependent record types and coercive subtyping, because this definition lays the groundwork for the automatic machine translation from myJava programs to Coq codes.

The central place of this definition is devoted to how to interpret object state, method entries of a class, inheritance relationship with interface-types, subtype polymorphism and dynamic dispatch in dependent record types with coercive subtyping. Particularly, much attention will be directed to modeling the method-calls such as *implicit-method-call statement and expression*, *explicit-method-call statement and expression*, and *field-method-call statement and expression*.

**Implementing the verification software - JaCo**

From the above type-theoretic semantics of myJava programs, we have designed a new verification tool, called JaCo, with the process given in Figure 1.4. First

$^3$For instance, a return-expression in the method must be written after the last statement, and is required to be only an identifier representing a state such as fields and local variables.
we need to convert the BNF syntax of myJava defined in Figure 1.2 and Figure 1.3 into the machine-readable BNF syntax by using the parser generator JavaCC [Cop07] [javb]. The implementation of JaCo is then achieved through employing the visitor algorithm [BJ01] [App97] with the help of Java API [Hor02] [java].

![Figure 1.4: Process of developing JaCo](image)

**Case study**

In Chapter 2, we will first visit with a practical angle the theme of interpreting the core features of myJava programs as type-theoretic models with dependent record types and coercive subtyping. In Chapter 5, we will provide another case for how our type-theoretical modeling is applied to computational geometry [O’R98].

### 1.4 Structure of the thesis

We provide summaries of the content of each of the chapters that follow. The structure of the thesis has been designed so that readers with different backgrounds or interests can find it a useful read without having to plough through all pages.

**Chapter 1** You are reading it. Introduces the areas of interest, explains the background study, and sets out some objectives.

**Chapter 2** Gives the motivation for a functional interpretation of object-oriented programs by modeling and verifying myJava programs in the Coq system,

---

4The BNF syntax written in JavaCC is given in Appendix C
adding a new perspective to the debate whether the key properties of object-oriented programs such as encapsulation, inheritance, reuse, subtype polymorphism and dynamic dispatch can be modeled in Coq.

Chapter 3 Gives a brief formal presentation of UTT with dependent record types and coercive subtyping, which can capture exactly the core features of object-oriented programs.

Chapter 4 Defines the type-theoretical semantics for constructing models of my- Java programs in UTT.

Chapter 5 Shows case study with modeling classes and proving their properties.

Chapter 6 Concludes the thesis with discussion and summary.
Chapter 2

Verification of myJava programs
in Coq

Central to this chapter is to give the motivation for functional interpretation of object-oriented programs by modeling and verifying myJava programs in the Coq system [BC04] [Coq]. Therefore, the present study primarily sets out to investigate whether the key properties of object-oriented programs such as encapsulation, inheritance, reuse, subtype polymorphism and dynamic dispatch can be modeled in Coq, which supports coercive subtyping [Saı97] and a macro for dependent record types.

To put the point another way, we begin to interpret myJava programs as type-theoretic models with a practical angle. It is because it provides necessary background for understanding the formal semantics of a functional interpretation with dependent record types and coercive subtyping given in Chapter 4. Our investigation now stands in need of two things. First, we are modeling myJava programs by using Coq as a functional programming language. Secondly, we are then verifying specifications being satisfied by those programs such as class invariants and pre-and-post conditions for methods by using Coq as a proof assistant.

In the pages which follow we shall develop such an investigation with the four classes \textit{Storage}, \textit{NewStorage}, \textit{Supplier} and \textit{Manager}, and their interface-types shown in Figure 2.1. We are interpreting these classes as record types by means of separating fields from methods. All Coq codes given in this chapter have been automatically generated by JaCo.
CHAPTER 2. VERIFICATION OF MYJAVA PROGRAMS IN COQ

Figure 2.1: UML class diagram

```
interface InterfaceStorage
{
    public void Storage(const(int x));
    public int GetStock();
    public void Setting(int n);
}

class Storage implements InterfaceStorage
{
    protected int stock;

    public void Storage(const(int x)) //like a constructor
    {
        stock = x;
    }

    public int GetStock()
    {
        return stock;
    }

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

interface InterfaceNewStorage extends InterfaceStorage
{
    public void NewStorage(const());
    public void Decreasing();
    public void Setting(int n);
    public void Restore();
}
```
class NewStorage extends Storage implements InterfaceNewStorage
/* INVARIANT [[ saleProduct <= stock ]] */
{
  protected int saleProduct;

  public void NewStorageCONST() //like a constructor
  {
    saleProduct = 0;
  }

  public void Decreasing()
  {
    while (saleProduct < stock)
    {
      stock = stock - 1;
    }
  }

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

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

interface InterfaceSupplier
{
  public void SupplierCONST(int x, int y);
  public InterfaceStorage GetCell();
  public void ClearExtra();
  public void Supply(InterfaceStorage x);
  public void GetTotalStock(InterfaceStorage x);
  public void FieldSetting(int n);
}

class Supplier implements InterfaceSupplier
{
  protected int sum;
  protected int extra;
  protected InterfaceStorage cell;

  public void SupplierCONST(int x, int y)
  {
    sum = x;
    extra = y;
  }
cell = new NewStorage(); //generating a new object
cell.StorageCONST(10); //field-method-call statement

//type-casting and field-method-call statement
((NewStorage)cell).NewStorageCONST();

}

public InterfaceStorage GetCell()
{
    return cell; // returning an object of interface-type
}

public void ClearExtra()
{
    extra = 0;
}

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

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

public void FieldSetting(int n)
{
    cell.Setting(n); // field-method-call statement
}

interface InterfaceManager
{
    public void ManagerCONST(int x, InterfaceSupplier y);
    public void FindBalance(InterfaceStorage x);
    public void SettingCopyCell(int n);
}

class Manager implements InterfaceManager
{
    protected int balance;
    protected InterfaceStorage copyCell;

    public void ManagerCONST(int x, InterfaceSupplier y)
    {
        balance = x;
        copyCell = y.GetCell(); //explicit-method-call expression
2.1 Modeling myJava programs in Coq

Before moving to the central part of our argument, we shall briefly discuss some of the major features of myJava language. First, myJava does not have a constructor that contains instructions to initialize objects. In the object-oriented programming, a constructor has the same name as that of the class to which it belongs, and does not return a value. Moreover, as a constructor is treated as not being a method, we cannot invoke it on an existing object. Instead, we are here defining the methods `StorageCONST`, `NewStorageCONST`, `SupplierCONST`, and `ManagerCONST` which initialize the state of objects like a constructor and can be invoked on objects.

Secondly, in myJava programs, methods can be declared only as public and fields as protected. The public methods of a class form the public interface of the class, which are the operations that an object in the context can access to manipulate instances of the class. Protected fields can be accessed only by methods of the class to which they belong or by methods of the subclass to which the class is inherited.

2.1.1 Modeling object state

The state of an object is the set of values that determine how an object reacts to method calls. An object uses instance fields, which are the data that the object needs to execute its methods, for the sake of storing its state. Each object of a class has its own set of instance fields. We are modeling the state of objects in the context by grouping the set of fields of each class into a non-dependent record
type. We will then define the Ω-state consisting of all these record types of state. After the Ω-state is decided, we will be able to start to model the methods of classes. All methods of classes are modeled as taking an object of the Ω-state as an implicit parameter and returning a new object of the Ω-state if the state is changed.

First, the fields of the class Storage are separately grouped to the record type of state, so that the state of the class Storage is interpreted as objects of the record type SStorage.

\[
\text{Record SStorage: Set := mk_SStorage \{ stock : nat \}.}
\]

where the field stock is used as a label in the record type. In Coq, labels are defined as global functions which are required to be different from others. Therefore, field names in classes written in myJava should be regretfully different each other even though they are members of different classes.

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

\[
\text{Record SNewStorage : Set := mk_SNewStorage \{label_storage_1 :> SStorage ; saleProduct : nat \}.}
\]

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

\[
SNewStorage \leq_c SStorage
\]

Continuously, we define the state of the class Supplier

\[
\text{Record SSupplier : Set := mk_SSsupplier \{ sum : nat ; extra : nat ; cell :> SNewStorage \}.}
\]

Much attention is given to the field cell which is declared of the interface-type InterfaceStorage. One thing to note is that an instance of the class Storage implementing InterfaceStorage or an instance of the class NewStorage implementing InterfaceNewStorage which inherits from InterfaceStorage can be assigned to cell.
In spite of the fact that an instance of a class includes methods and fields, we are here modeling the field \textit{cell} only with the state of the object which is assigned to the field \textit{cell}. In the method \texttt{Supplier\textsc{Const}(int x, int y)}, an instance of the class \texttt{NewStorage} is assigned to the field \textit{cell}. For that reason, we are interpreting the field \textit{cell} as an object of the record type \texttt{S\textsc{NewStorage}} instead of \texttt{S\textsc{Storage}} with the coercion function \( d \equiv \text{cell} \) such that all \( x : \texttt{SSupplier}, \ d(x) : \texttt{S\textsc{NewStorage}} \).

\[
\texttt{SSupplier} \leq_{d} \texttt{S\textsc{NewStorage}}
\]

Furthermore, by coercion transitivity the following coercions graph is defined with the composite function \( d \circ c \)

\[
\texttt{SSupplier} \leq_{\text{doc}} \texttt{S\textsc{Storage}}
\]

Next, the state of the class \texttt{Manager} is defined as following,

\begin{verbatim}
Record \texttt{SManager} : Set := mk_{\texttt{SManager}} { balance : nat ;
  copyCell :> \texttt{S\textsc{NewStorage}} }.
\end{verbatim}

Ostensibly, the field \texttt{copyCell} is modeled as an object of the record type \texttt{S\textsc{NewStorage}} because in the method \texttt{Manager\textsc{Const}(int x, Interface-Supplier y)}, the object that the statement \( y.\texttt{GetCell()} \) returns is assigned to \texttt{copyCell}. However, there is more here than meets the eye: we can shrewdly observe that the field \texttt{copyCell} is just a reference for the field object \textit{cell} in the class \texttt{Supplier}. In other words, the fields \textit{cell} and \texttt{copyCell} are actually the same object. As a consequence, the state of the two fields should always be the same while the classes \texttt{Supplier} and \texttt{Manager} are being instantiated. For this reason, the record field \texttt{copyCell} in the record type \texttt{SManager} must synchronize with the record field \textit{cell} in the record type \texttt{SSupplier}. This synchronization will be explained more later on.

Finally, we are defining the universal state \texttt{Omega} which is able to capture all states of the classes \texttt{Storage, NewStorage, Supplier,} and \texttt{Manager} simultaneously.

\begin{verbatim}
Record \texttt{Omega} : Set := mk_{\texttt{Omega}} { label_storage : \texttt{S\textsc{Storage}} ;
  label_newstorage : \texttt{S\textsc{NewStorage}} ;
  label_supplier : \texttt{SSupplier} ;
  label_manager : \texttt{SManager} }.
\end{verbatim}
The non-dependent record type \texttt{Omega} is very essential for a functional interpretation of myJava programs because functional programming does not have an idea of changing stored values on the hardware memory. Regrettably, we should assume in our modeling that the \texttt{Omega} state can capture only one instance of each class, i.e. there exists only one instance for each class in the context. It is because we are just modeling classes written in myJava which can not declare a class with the main method\footnote{In Java programming, when the application starts, the instructions in the main method are executed: \texttt{public static void main(String[] args) \{ \ldots \}}}, so that we do not know how many instances are acting in the context.

After \texttt{Omega} is decided, we will start to model the methods of the classes \texttt{Storage, NewStorage, Supplier}, and \texttt{Manager}. All methods are modeled as taking an object of \texttt{Omega} as an implicit parameter and returning a new object of \texttt{Omega} if the \texttt{Omega} state is changed. However, each method is modeled as having to access only the state of the class to which it belongs. This realizes the important notion of \textit{encapsulation} in object-oriented programming, which is the process of hiding the data and providing methods for data access to guarantee that an object can not accidently be put into an incorrect state.

The more important reason that \texttt{Omega} is needed for our interpretation is to handle the \textit{contra-variant}\footnote{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. $A \rightarrow B \leq A' \rightarrow B'$ provided that $A' \leq A$ and $B' \leq B'$.} 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 \texttt{GetStock()} in the class \texttt{Storage} as a function \texttt{GetStock\_Storage} of type \texttt{SStorage \rightarrow nat} and the method \texttt{GetStock()} which is implicitly inherited to the class \texttt{NewStorage} as a function \texttt{GetStock\_NewStorage} of type \texttt{SNewStorage \rightarrow nat}, we would have met the contra-variant problem while modeling coercive subtyping between \texttt{SNewStorage \rightarrow nat} and \texttt{SStorage \rightarrow nat}. It is because that although \texttt{SNewStorage} is a subtype of \texttt{SStorage}, the function \texttt{GetStock\_NewStorage} of type \texttt{SNewStorage \rightarrow nat} can not be an object of the type \texttt{SStorage \rightarrow nat} due to the contra-variant. However, if we model the both methods as a function of type \texttt{Omega \rightarrow nat}, the contra-variant problem can be avoided.

More remarkable is for the \texttt{Omega}-state even to capture tree graphs of inheritance relationships Java supports, as we will see it in Chapter 4.
2.1.2 Modeling methods (member functions) of objects

We introduce useful terminologies for modeling methods of a class written in myJava. All methods are classified as either an accessor or a mutator, or can be both, depending on what effect they have on object state. An accessor method accesses an object and returns some information about it without changing the state of the object. In contrast, a mutator method modifies and changes the state of an object, but does not return a value - the return type should be void. For example, in the class Storage, the method GetStock() is an accessor and the method Setting(int n) is a mutator. It is needless to say that only mutator methods are modeled as returning a new object of Omega.

Each method consists of a sequence of statements: assignment statement, if-then-else statement, while-loop statement, implicit-method-call statement, explicit-method-call statement, and field-method-call statement. We are interpreting each statement as a function taking an object of Omega and returning a new object of Omega except for while-loop statements. Therefore, methods of a class are modeled as a composite function of these statements. The formal definition of each statement will be given in Chapter 4.

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

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

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

\[
\Gamma \vdash A:Type \quad \Gamma \vdash a:A \\
\Gamma \vdash \text{Unit}(A,a) \leq \xi(A,a) A : Type
\]

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 \( x : \text{Unit}(A,a) \). Additionally, we need to define the identity \( ID(A) = A \) on types to force Coq to accept \( \xi \)-coercion, and 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 \( \text{Id}_A \equiv [x:A]x \)

\[
\Gamma \vdash A : \text{Type} \\
\Gamma \vdash A \leq \text{Id}_A \quad A : \text{Type}
\]

Definition ID (A:Set) : Set := A.
Coercion unit_coercion (A:Set)(a:A)(_:Unit a) := a : ID A.

Now, we are interpreting the class \textit{Storage}\(^3\), noting that unit types are defined implicitly, i.e. parameters can be omitted. As we said earlier, the methods of the class \textit{Storage} have accessed only the state of \textit{SStorage} of the \textit{Omega} state

\[
\text{(* Modeling the method StorageCONST of the class Storage *)}
\]

Let StorageCONST_Storage : nat->Omega->Omega := fun(x:nat)(w:Omega) =>
let w1 := w
in let Assign1_StorageCONST := fun (aw1 : Omega) =>
  (mk_Omega (mk_SStorage x)
   aw1.(label_newstorage)
   aw1.(label_supplier)
   aw1.(label_manager))
in let w2 := Assign1_StorageCONST w1
in w2.

\[
\text{(* Modeling the method GetStock of the class Storage *)}
\]

Let GetStock_Storage : Omega->nat := fun(w:Omega) => w.(label_storage).(stock).

\[
\text{(* Modeling the method Setting of the class Storage *)}
\]

Let Setting_Storage ( GetStock : Omega->nat ) : nat -> Omega-> Omega := fun(n : nat) (w : Omega) =>
let w1 := w
in let If1_Setting := fun (iw1:Omega) =>
  if ( LessThan ( GetStock iw1) n)
  then let Assign1_Setting := fun (aw1 : Omega) =>
    (mk_Omega (mk_SStorage n)
     aw1.(label_newstorage)
     aw1.(label_supplier)
     aw1.(label_manager))
in Assign1_Setting iw1
  else iw1
in let w2 := If1_Setting w1
in w2.

\(^3\)In Coq codes, \texttt{LessThan} is the boolean comparison operator of type \texttt{nat -> nat -> bool} on natural numbers. The definition of this function is given in Appendix B.
CHAPTER 2. VERIFICATION OF MYJAVA PROGRAMS IN COQ

(* Define a record type representing all methods of the class Storage *)
Record MStorage : Set := mk_MStorage {
  label_StorageCONST_Storage : Unit StorageCONST_Storage ;
  label_GetStock_Storage : Unit GetStock_Storage ;
  label_Setting_Storage : Unit (Setting_Storage(label_GetStock_Storage : ID(Omega->nat)) ) }.

Implicit-method-call Statement and Expression

First, we need to explain intuitively what *implicit-method-call statement* and *implicit-method-call expression* are 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 in myJava codes denotes the implicit parameter: for instance,\texttt{this.NewStorageCONST() }in the method \texttt{Restore()} of the class \texttt{NewStorage} is called an *implicit-method-call statement* where ‘this’ refers to 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 *implicit-method-call expression* where ‘this’ refers to 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. One point is worth noting that an implicit-method-call statement should be a mutator, and an implicit-method-call expression an accessor.

We are next modeling the set of all methods in the class \texttt{Storage} 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 non-structural coercion function $\xi$ defined previously. As we have seen in the above, the method \texttt{Setting(int n)} is modeled as a function \texttt{Setting_Storage} being parameterized by another function of type $\Omega \to \text{nat}$ in order to model the implicit-method-call expression \texttt{this.GetStock()}. This parameterized function is integrated in a unit type by applying the function \texttt{Setting_Storage} to the field \texttt{label_GetStock_Storage} defined earlier in the record type \texttt{MStorage}. However, the field \texttt{label_GetStock_Storage} in the record type \texttt{MStorage} is not a function since it is of type (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 \texttt{label_GetStock_Storage} being an object of type (Unit GetStock_Storage) is coerced into $\xi((\text{unit GetStock_Storage})) = \text{GetStock_Storage}$ of type $\Omega \to \text{nat}$. 
Finally, we are modeling the class *Storage* as the record type *Storage* merging the record type of state *SStorage* and the record type of methods *MStorage* with the structural coercion \( c_1 \equiv \text{label}_\text{mstorage} \).

\[
\text{Storage} \leq \text{c}_1 \text{ MStorage}
\]

```coq
(** Model the class Storage as a record type *)
Record Storage : Set := mk_Storage { label_sstorage : SStorage ; label_mstorage := MStorage }.
```

**Modeling While-loop statement**

Before moving to the next section, we are dealing with how a terminating while-loop statement in the method *Decreasing()* of the class *NewStorage* can be modeled with a primitive recursive function.

```java
public void Decreasing()
{
    while (saleProduct < stock)
    {
        stock = stock - 1;
    }
}
```

In our interpretation, all statements except for a while-loop statement are interpreted as a function that maps an object \( w \) of \( \Omega \) to the expression where a term corresponding to a field variable appearing in a statement is built by using \( w \) with record field selection and coercion transitivity. For example, the field *stock* is modeled as \( w.(\text{label}_\text{storage}).(\text{stock}) \) in the function *GetStock_Storage*, or as \( w.(\text{label}_\text{newstorage}).(\text{stock}) \) in the function *GetStock_NewStorage*.

However, we cannot follow this approach in modeling a while-loop statement as a primitive recursive function, because an object of \( \Omega \) cannot be used in the recursive function definition as a primary argument which plays a main role in building a pattern matching construct, i.e. we can not perform a structural recursion over the argument \( \omega \) of \( \Omega \). In Coq, a primary argument is denoted with the directive \{ \text{struct } \text{var}_1 \ldots \text{var}_n \}.

Let’s have a look at how JaCo interprets the above while-loop statement. JaCo first goes through the while-loop statement to collect variables whose state are changed or not changed. The collected variable whose state is changed will be used as a primary argument in the recursive function definition. The
while-loop statement is interpreted as a function `While1_Decreasing` taking the variables, which JaCo has collected, as arguments and applying the primitive recursive function `While_Decreasing_NewStorage1` to them. The function `While_Decreasing_NewStorage1` should give the same semantical result as the while-loop statement. Unfortunately, the actual definition of this recursive function is left to users. However, JaCo can reconstruct a new object of `Omega` after getting a new state of each field variable from `While_Decreasing_NewStorage1`.

Definition `UserDefn {T : Type} : T. Admitted.`

The above lines of magic define a `UserDefn` value that can fill a hole in an incomplete definition or proof. The following is the interpretation corresponding to the method `Decreasing()`.

```
Fixpoint While_Decreasing_NewStorage1 (STOCK : nat)(SALEPRODUCT : nat) { struct STOCK }
Let Decreasing_NewStorage :
  Omega->Omega := fun(w : Omega) =>
  let w1 := w
  in let While1_Decreasing := fun (STOCK : nat)(SALEPRODUCT : nat)=>
      While_Decreasing_NewStorage1 STOCK SALEPRODUCT
  in let before_stock := w1.(label_newstorage).(stock)
  in let before_saleProduct := w1.(label_newstorage).(saleProduct)
  in let after_stock := While1_Decreasing before_stock before_saleProduct
  in let w2 := ( mk_Omega w1.(label_storage)
      ( mk_SNewStorage ( mk_SStorage after_stock )
        w1.(label_newstorage).(saleProduct) )
    w1.(label_supplier)
    w1.(label_manager) )
in w2.
```

The function `While_Decreasing_NewStorage1` left to users has been manually defined by the author.

```
Fixpoint minus (n m : nat) : nat :=
  match n with
  | 0 => n
  | S k => match n with
      | 0 => n
      | S l => minus k l
    end
end.
```
Fixpoint While_Decreasing_NewStorage1
(STOCK : nat)(SALEPRODUCT : nat) { struct STOCK } : nat :=
if (LessThan SALEPRODUCT STOCK )
then let diff := (minus STOCK SALEPRODUCT)
in (minus STOCK diff)
else STOCK.

2.1.3 Modeling inheritance

Inheritance is a way of extending existing classes for code reuse. The following
is interpretation of the class \textit{NewStorage} where we can see that the methods
\textit{StorageCONST}(int x) and \textit{GetStock()} defined in the class \textit{Storage} are
implicitly inherited to the class \textit{NewStorage}, the method \textit{Setting}(int n) is
overridden, and the methods \textit{Decreasing()} and \textit{Restore()} are added as a new
member function.

\begin{verbatim}
(* Modeling the method StorageCONST of the class NewStorage *)
Let StorageCONST_NewStorage :
at->Omega->Omega := fun(x : nat)(w : Omega) =>
let w1 := w
in let Assign1_StorageCONST := fun (aw1 : Omega) =>
  ( mk_Omega aw1.(label_storage)
    ( mk_SNewStorage ( mk_SStorage x )
      aw1.(label_newstorage).(saleProduct) )
    aw1.(label_supplier)
    aw1.(label_manager) )
in let w2 := Assign1_StorageCONST w1
in w2.

(* Modeling the method GetStock of the class NewStorage *)
Let GetStock_NewStorage :
Omega->nat:= fun(w : Omega) => w.(label_newstorage).(stock).

(* Modeling the method NewStorageCONST of the class NewStorage *)
Let NewStorageCONST_NewStorage :
Omega->Omega:= fun(w : Omega) =>
let w1 := w
in let Assign1_NewStorageCONST := fun (aw1 : Omega) =>
  (mk_Omega aw1.(label_storage)
    (mk_SNewStorage (mk_SStorage aw1.(label_newstorage).(stock))
    0)
    aw1.(label_supplier)
    aw1.(label_manager) )
in let w2 := Assign1_NewStorageCONST w1
in w2.

(* Modeling the method Decreasing of the class NewStorage *)
Fixpoint minus (n m : nat) : nat :=
\end{verbatim}
match n with
| 0 => n
| S k => match m with
| 0 => n
| S l => minus k l
end
end.

Fixpoint While_Decreasing_NewStorage1
(STOCK : nat)(SALEPRODUCT : nat) { struct STOCK } : nat :=
if (LessThan SALEPRODUCT STOCK )
then let diff := (minus STOCK SALEPRODUCT)
    in (minus STOCK diff)
else STOCK.

Let Decreasing_NewStorage :
Omega->Omega
:= fun(w : Omega) =>
let w1 := w in
let While1_Decreasing := fun (STOCK : nat)(SALEPRODUCT : nat)
    => While_Decreasing_NewStorage1 STOCK SALEPRODUCT
in let before_stock := w1.(label_newstorage).(stock)
in let before_saleProduct := w1.(label_newstorage).(saleProduct)
in let after_stock := While1_Decreasing before_stock before_saleProduct
in let w2 := (mk_Omega w1.(label_storage)
    ( mk_SNewStorage ( mk_SStorage after_stock )
        w1.(label_newstorage).(saleProduct) )
    w1.(label_supplier)
    w1.(label_manager) )
in w2.

(* Modeling the method Setting of the class NewStorage (*)

Let Setting_NewStorage
( GetStock : Omega->nat ) : nat->Omega->Omega:=
fun(n : nat)(w : Omega) =>
let w1 := w in
let If1_Setting := fun (iw1:Omega) =>
    if ( LessThan ( GetStock iw1) n)
    then let Assign1_Setting := fun (aw1 : Omega) =>
        (mk_Omega aw1.(label_storage)
            ( mk_SNewStorage ( mk_SStorage aw1.(label_newstorage).(stock))
                ( GetStock aw1) )
            aw1.(label_supplier)
            aw1.(label_manager) )
        in let Assign2_Setting := fun (aw2 : Omega) =>
            ( mk_Omega aw2.(label_storage)
                ( mk_SNewStorage ( mk_SStorage n )
                    aw2.(label_newstorage).(saleProduct) )
            aw2.(label_supplier)
            aw2.(label_manager) )
in (Assign2_Setting (Assign1_Setting iw1) )
else iw1
in let w2 := If1_Setting w1
in w2.
(* Modeling the method Restore of the class NewStorage *)
Let Restore_NewStorage
( NewStorageCONST : Omega->Omega ) : Omega->Omega := fun (w : Omega) =>
  let w1 := w
  in let Assign1_Restore := fun (aw1 : Omega) =>
      ( mk_Omega aw1.(label_storage)
        (mk_SNewStorage (mk_SStorage aw1.(label_newstorage).(saleProduct))
          aw1.(label_newstorage).(saleProduct))
      aw1.(label_supplier)
      aw1.(label_manager) )
  in let w2 := Assign1_Restore w1
  in let CallStatement1_Restore := fun (cw1 : Omega) => NewStorageCONST cw1
  in let w3 := CallStatement1_Restore w2
  in w3.

(* Define a record type representing all methods of the class NewStorage *)
Record MNewStorage : Set := mk_MNewStorage {
  label_StorageCONST_NewStorage : Unit StorageCONST_NewStorage ;
  label_GetStock_NewStorage : Unit GetStock_NewStorage ;
  label_NewStorageCONST_NewStorage : Unit NewStorageCONST_NewStorage ;
  label_Decreasing_NewStorage : Unit Decreasing_NewStorage ;
  label_Setting_NewStorage : Unit (Setting_NewStorage
    (label_GetStock_NewStorage : ID(Omega->nat))) ;
  label_Restore_NewStorage : Unit (Restore_NewStorage
    (label_NewStorageCONST_NewStorage : ID(Omega->Omega)))}.

(* Model the class NewStorage as a record type *)
Record NewStorage : Set := mk_NewStorage {
  label_snewstorage : SNewStorage ;
  label_mnewstorage :> MNewStorage }.

Now we are modeling inheritance relationship between the classes Storage and NewStorage as subtyping. We should note that we are here not modeling direct inheritance relationship between the classes Storage and 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 2.1, we have defined the classes Storage and NewStorage implementing the interface-types InterfaceStorage and InterfaceNewStorage.
respectively. We interpret the interface-type \textit{InterfaceStorage} as the record type \textit{InterfaceStorage}, and then define the component-wise coercion function between the record type \textit{MStorage} and the record type \textit{InterfaceStorage}.

\begin{verbatim}
(* Model the interface-type InterfaceStorage as a record type *) Record InterfaceStorage : Set := mk_InterfaceStorage { i_label_StorageCONST_Storage : nat->Omega->Omega ;
i_label_GetStock_Storage : Omega->nat ;
i_label_Setting_Storage : nat->Omega->Omega }.

(* Define a coercion between MStorage and InterfaceStorage *) Coercion MStorage_InterfaceStorage (x : MStorage) : InterfaceStorage := mk_InterfaceStorage (label_StorageCONST_Storage x : ID (nat->Omega->Omega) )
(label_GetStock_Storage x : ID (Omega->nat) )
(label_Setting_Storage x : ID (nat->Omega->Omega) ) .
\end{verbatim}

As a result, we can see that the relation between the class \textit{Storage} and the interface-type \textit{InterfaceStorage} is interpreted as subtyping relation by the composite function \( c_2 \circ c_1 \) where

\[
c_1 \equiv \text{label\_mstorage}, \text{ and } c_2 \equiv \text{MStorage\_InterfaceStorage}.
\]

\[
\text{Storage} \leq_{c_1} \text{MStorage} \leq_{c_2} \text{InterfaceStorage}
\]

\[
\text{Storage} \leq_{c_2 \circ c_1} \text{InterfaceStorage}
\]

\begin{verbatim}
(* Model the interface-type InterfaceNewStorage as a record type *) Record InterfaceNewStorage : Set := mk_InterfaceNewStorage { i_label_StorageCONST_NewStorage : nat->Omega->Omega ;
i_label_GetStock_NewStorage : Omega->nat ;
i_label_NewStorageCONST_NewStorage : Omega->Omega ;
i_label_Decreasing_NewStorage : Omega->Omega ;
i_label_Setting_NewStorage : nat->Omega->Omega ;
i_label_Restore_NewStorage : Omega->Omega }.

(* Define a coercion between MNewStorage and InterfaceNewStorage *) Coercion MNewStorage_InterfaceNewStorage (x : MNewStorage) : InterfaceNewStorage := mk_InterfaceNewStorage (label_StorageCONST_NewStorage x : ID (nat->Omega->Omega) )
(label_GetStock_NewStorage x : ID (Omega->nat) )
(label_NewStorageCONST_NewStorage x : ID (Omega->Omega) )
(label_Decreasing_NewStorage x : ID (Omega->Omega) )
(label_Setting_NewStorage x : ID (nat->Omega->Omega) )
(label_Restore_NewStorage x : ID (Omega->Omega) ) .
\end{verbatim}

Likewise, we are modeling the relation between the class \textit{NewStorage} and the interface-type \textit{InterfaceNewStorage} as subtyping relation by the composite function \( c_4 \circ c_3 \) where

\[
\text{NewStorage} \leq_{c_3} \text{MNewStorage} \leq_{c_4 \circ c_3} \text{InterfaceNewStorage}
\]
\[ c_3 \equiv \text{label\_mnewstorage}, \text{ and } c_4 \equiv \text{MNewStorage\_InterfaceNewStorage}. \]

\[
\text{NewStorage} \leq c_3 \text{ MNewStorage} \leq c_4 \text{ InterfaceNewStorage}
\]

\[
\text{NewStorage} \leq c_4 \circ c_3 \text{ InterfaceNewStorage}
\]

(* Define a coercion between InterfaceNewStorage and InterfaceStorage *)

Coercion InterfaceNewStorage\_InterfaceStorage (x : InterfaceNewStorage) : InterfaceStorage :=

\[
\text{mk\_InterfaceStorage}
\]

\[
x. (\text{i\_label\_StorageCONST\_NewStorage})
\]

\[
x. (\text{i\_label\_GetStock\_NewStorage})
\]

\[
x. (\text{i\_label\_Setting\_NewStorage}).
\]

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

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

Proceeding from what we have said above, indirect inheritance relationship is eventually modeled as the coercion graph in Figure 2.2.

Figure 2.2: Coercion graphs for indirect inheritance

**Remark** We could not define a coercion function between the record types \text{Storage} and \text{NewStorage}, or between the record types \text{MStorage} and \text{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 such coercion functions, we had confronted a coercion coherence\(^4\) problem.

### 2.1.4 Modeling subtype polymorphism and dynamic dispatch

Subtype polymorphism and dynamic dispatch with interface-types provide one of the ways of accessing other objects with possible side effects, which means that the state of the referred object can be changed. We will capture these mechanisms in type theoretic models with the frame of coercive subtyping shown in Figure 2.2. As we see the class `Manager` in Figure 2.1, it would exchange messages with the classes `Storage`, `NewStorage`, and `Supplier`.

**Explicit-method-call Statement and Expression**

We need to define the notion of the *explicit-method-call expression* and *explicit-method-call statement* with an explicit parameter whose type is an interface-type. This parameter is an object which a method is explicitly taking as an argument. For example, `x.GetStock()` in the method `GetTotalStock(InterfaceStorage x)` of the class `Supplier` is an explicit-method-call expression, and `x.Setting(extra)` in the method `Supply(InterfaceStorage x)` of the class `Supplier` is an explicit-method-call statement. Consider the following assignment statements and function applications in Java programing.

```java
InterfaceStorage iStorage_A, iStorage_B;
iStorage_A = new Storage();
iStorage_B = new NewStorage();
Supplier mySupplier = new Supplier();
mySupplier.Supply(iStorage_A);
mySupplier.Supply(iStorage_B);
```

First, two variables `iStorage_A` and `iStorage_B` are declared as objects of type `InterfaceStorage`. Accordingly, an instance of class `Storage` can be

---

\(^4\) 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.
assigned to the variable $i\text{Storage}_A$ because the class $\text{Storage}$ implements the interface-type $\text{InterfaceStorage}$. Likewise, an instance of class $\text{NewStorage}$ can be assigned to the variable $i\text{Storage}_B$ because the class $\text{NewStorage}$ implements $\text{InterfaceNewStorage}$ inheriting from $\text{InterfaceStorage}$.

Secondly, the objects $i\text{Storage}_A$ and $i\text{Storage}_B$ can be passed to the method $\text{Supply(InterfaceStorage }x\text{)}$ of the class $\text{Supplier}$ that expects an instance of a class implementing $\text{InterfaceStorage}$ or an instance of a class implementing $\text{InterfaceNewStorage}$ which inherits from $\text{InterfaceStorage}$. These codes would be illegal in functional programming language, because the types $\text{InterfaceStorage}$ and $\text{InterfaceNewStorage}$ do not match. However, in object-oriented programming, these codes are made legal by Subtype Polymorphism: if $\text{NewStorage}$ is a subclass of $\text{Storage}$ by way of the interface-type $\text{InterfaceNewStorage}$ being a subtype of the interface-type $\text{InterfaceStorage}$, and $o$ is an object of $\text{NewStorage}$, then $o$ is an object of $\text{Storage}$.

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

The interpretation for the class $\text{Supplier}$ is given as follows\(^5\),

\[
\begin{aligned}
\text{Let } \text{SupplierCONST\_Supplier} &\text{ : nat}\to\text{nat}\to\text{Omega}\to\text{Omega} := \text{fun } (x : \text{nat})(y : \text{nat})(w : \text{Omega}) \Rightarrow \\
&\text{let } w1 := w \\
&\text{in let } \text{Assign1\_SupplierCONST} := \text{fun } (aw1 : \text{Omega}) \Rightarrow \\
&\quad (\text{mk\_Omega } aw1.(\text{label\_storage}) \\
&\quad \quad aw1.(\text{label\_newstorage}) \\
&\quad \quad (\text{mk\_SSupplier } x \\
&\quad \quad \quad aw1.(\text{label\_supplier}).(\text{extra}) \\
&\quad \quad \quad aw1.(\text{label\_supplier}).(\text{cell}) ) \\
&\quad \quad aw1.(\text{label\_manager}) ) \\
&\text{in let } w2 := \text{Assign1\_SupplierCONST } w1 \\
&\text{in let } \text{Assign2\_SupplierCONST} := \text{fun } (aw2 : \text{Omega}) \Rightarrow 
\end{aligned}
\]

\(^5\)Here, readers might be confused with interpretation of the class $\text{Supplier}$ due to lots of new identifiers, which have been used for modeling field-method-call statements and field-method-call expressions. We will come back to them in a bit.
( mk_Omega aw2.(label_storage)
  aw2.(label_newstorage)
  ( mk_SSupplier aw2.(label_supplier).(sum)
    y
    aw2.(label_supplier).(cell) )
  aw2.(label_manager) )
in let w3 := Assign2_SupplierCONST w2
in let Assign3_SupplierCONST := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_storage)
    aw3.(label_newstorage)
    ( mk_SSupplier aw3.(label_supplier).(sum)
      aw3.(label_supplier).(extra)
      ( mk_SNewStorage ( mk_SStorage 0 ) 0 )
    aw3.(label_manager) )
in let w4 := Assign3_SupplierCONST w3
in let CallStatement1_SupplierCONST := fun (cw1 : Omega_InterfaceNewStorage) =>
  cw1.(label_interfacenewstorage).(i_label_StorageCONST_NewStorage ) 10 cw1
in let w5 := mk_Omega_InterfaceNewStorage w4
  ( mk_InterfaceNewStorage
    StorageCONST_Supplier
    GetStock_Supplier
    NewStorageCONST_Supplier
    Decreasing_Supplier
    ( Setting_Supplier GetStock_Supplier )
    ( Restore_Supplier NewStorageCONST_Supplier )))
in let w6 := CallStatement1_SupplierCONST w5
in let CallStatement2_SupplierCONST := fun (cw2 : Omega_InterfaceNewStorage) =>
  cw2.(label_interfacenewstorage).(i_label_NewStorageCONST_NewStorage ) cw2
in let w7 := mk_Omega_InterfaceNewStorage w6
  ( mk_InterfaceNewStorage
    StorageCONST_Supplier
    GetStock_Supplier
    NewStorageCONST_Supplier
    Decreasing_Supplier
    ( Setting_Supplier GetStock_Supplier )
    ( Restore_Supplier NewStorageCONST_Supplier )))
in let w8 := CallStatement2_SupplierCONST w7
in w8.

(* Modeling the method GetCell of the class Supplier *)
Let GetCell_Supplier
  : Omega->SNewStorage := fun(w : Omega) =>
  w.(label_supplier).(cell).

(* Modeling the method ClearExtra of the class Supplier *)
Let ClearExtra_Supplier
  : Omega->Omega:= fun(w : Omega) =>
  let w1 := w
  in let Assign1_ClearExtra := fun (aw1 : Omega) =>
    ( mk_Omega aw1.(label_storage)
      aw1.(label_newstorage)
      ( mk_SSupplier aw1.(label_supplier).(sum)
        0
      aw1.(label_supplier).(cell) )
Let \texttt{Supply\_Supplier}:

\begin{verbatim}
let \texttt{CallStatement1\_Supply} := fun (\texttt{cw1} : \texttt{Omega}) =>
  \texttt{x.\_label\_Setting\_Storage} \texttt{cw1.\_label\_supplier}.\_extra \texttt{cw1}
\end{verbatim}

Let \texttt{GetTotalStock\_Supplier}:

\begin{verbatim}
let \texttt{Assign1\_GetTotalStock} := fun (\texttt{aw1} : \texttt{Omega\_InterfaceNewStorage}) =>
  \texttt{mk\_Omega} \texttt{aw1.\_label\_storage}
  \texttt{aw1.\_SSupplier}
  \texttt{aw1.\_label\_supplier}.\_sum +
  \texttt{(x.\_label\_GetStock\_Storage) aw1)}
  \texttt{aw1.\_label\_supplier}.\_extra
  \texttt{aw1.\_label\_supplier}.\_cell)
\end{verbatim}

Let \texttt{FieldSetting\_Supplier}:

\begin{verbatim}
let \texttt{CallStatement1\_FieldSetting} := fun (\texttt{cw1} : \texttt{Omega\_InterfaceNewStorage}) =>
  \texttt{cw1.\_label\_interfacenewstorage}.\_i\_label\_Setting\_NewStorage n \texttt{cw1}
\end{verbatim}
in let w3 := CallStatement1_FieldSetting w2 in w3.

(* Define a record type representing all methods of the class Supplier *)
Record MSupplier : Set := mk_MSupplier {
  label_SupplierCONST_Supplier : Unit SupplierCONST_Supplier ;
  label_GetCell_Supplier : Unit GetCell_Supplier ;
  label_ClearExtra_Supplier : Unit ClearExtra_Supplier ;
  label_Supply_Supplier : Unit (Supply_Supplier
    (label_ClearExtra_Supplier : ID(Omega->Omega) )) ;
  label_GetTotalStock_Supplier : Unit GetTotalStock_Supplier ;
  label_FieldSetting_Supplier : Unit FieldSetting_Supplier .
}.

(* Model the class Supplier as a record type *)
Record Supplier : Set := mk_Supplier {
  label_ssupplier : SSupplier ;
  label_msupplier :> MSupplier }. 

(* Model the interface-type InterfaceSupplier as a record type *)
Record InterfaceSupplier : Set := mk_InterfaceSupplier {
  i_label_SupplierCONST_Supplier : nat->nat->Omega->Omega ;
  i_label_GetCell_Supplier : Omega->SNewStorage ;
  i_label_ClearExtra_Supplier : Omega->Omega ;
  i_label_Supply_Supplier : InterfaceStorage->Omega->Omega ;
  i_label_GetTotalStock_Supplier : InterfaceStorage->Omega->Omega ;
  i_label_FieldSetting_Supplier : nat->Omega->Omega }. 

(* Define a coercion between MSupplier and InterfaceSupplier *)
Coercion MSupplier_InterfaceSupplier (x : MSupplier) : InterfaceSupplier :=
  mk_InterfaceSupplier
  (label_SupplierCONST_Supplier x : ID (nat->nat->Omega->Omega) )
  (label_GetCell_Supplier x : ID (Omega->SNewStorage) )
  (label_ClearExtra_Supplier x : ID (Omega->Omega) )
  (label_Supply_Supplier x : ID (InterfaceStorage->Omega->Omega) )
  (label_GetTotalStock_Supplier x : ID (InterfaceStorage->Omega->Omega) )
  (label_FieldSetting_Supplier x : ID (nat->Omega->Omega) ).

We can see that the methods GetTotalStock(InterfaceStorage x) and Supply(InterfaceStorage x) are modeled as the functions GetTotalStock_Supplier and Supply_Supplier which take an object of the record type InterfaceStorage as an argument respectively. All explicit-method-call expressions and explicit-method-call statements in these methods are modeled as record field-selection. The following codes are to demonstrate whether dynamic dispatch works actually or not.

Let myOmega : Omega := mk_Omega
  (mk_SStorage 2)
  (mk_SNewStorage (mk_SStorage 4) 6)
  (mk_SSupplier 8 10 (mk_SNewStorage (mk_SStorage 12) 14 ) )
  (mk_SManager 16 (mk_SNewStorage (mk_SStorage 12) 14 ) ).
Let myStorage : InterfaceStorage := mk_InterfaceStorage
  StorageCONST_Storage
  GetStock_Storage
  (Setting_Storage GetStock_Storage).

Let myNewStorage : InterfaceNewStorage :=
  mk_InterfaceNewStorage
  StorageCONST_NewStorage
  GetStock_NewStorage
  NewStorageCONST_NewStorage
  Decreasing_NewStorage
  (Setting_NewStorage GetStock_NewStorage)
  (Restore_NewStorage NewStorageCONST_NewStorage).

Let mySupplier : InterfaceSupplier :=
  mk_InterfaceSupplier
  SupplierCONST_Supplier
  GetCell_Supplier
  ClearExtra_Supplier
  (Supply_Supplier ClearExtra_Supplier)
  GetTotalStock_Supplier
  FieldSetting_Supplier.

Eval compute in mySupplier.(i_label_Supply_Supplier) myStorage myOmega.
= mk_Omega
  (mk_SStorage 10)
  (mk_SNewStorage (mk_SStorage 4) 6)
  (mk_SUPplier 8 0 (mk_SNewStorage (mk_SStorage 12) 14))
  (mk_SManager 16 (mk_SNewStorage (mk_SStorage 12) 14))
: Omega.

Eval compute in mySupplier.(i_label_Supply_Supplier) myNewStorage myOmega.
= mk_Omega
  (mk_SStorage 2)
  (mk_SNewStorage (mk_SStorage 10) 4)
  (mk_SUPplier 8 0 (mk_SNewStorage (mk_SStorage 12) 14))
  (mk_SManager 16 (mk_SNewStorage (mk_SStorage 12) 14))
: Omega.

The statement \texttt{x.\textit{Setting}(extra)} in the method \texttt{Supply} of the class \texttt{Supplier} is modeled as record field-selection \texttt{x.(i_label_Setting_Storage)} for \texttt{x:InterfaceStorage}. Evidently, if \texttt{myStorage} and \texttt{myOmega} are passed to the function \texttt{Supply_Supplier}, the label \texttt{(i\_label\_Setting\_Storage)} makes the function \texttt{Setting_Storage} execute to change the state of \texttt{SStorage}: the stock value(2) is replaced by the extra value(10), and the extra value is set to 0. Likewise, if \texttt{myNewStorage} and \texttt{myOmega} are passed to the function \texttt{Supply_Supplier}, the label \texttt{(i\_label\_Setting\_Storage)} makes the function \texttt{Setting_NewStorage} execute to change the state of \texttt{SNewStorage}: the stock value(4) is stored to the
field saleProduct, the stock value(4) is replaced by the extra value(10), and the extra value is set to 0. That is to say, depending on the object passed to the function Supply_Supplier, the label (i_label_Setting_Storage) refers to the different function according to coercive subtyping relation defined in Figure 2.2.

Field-method-call Statement and Expression

We are further exploring subtype polymorphism and dynamic dispatch which may also occur in fields being declared of an interface-type, introducing the notion of field-method-call statement and field-method-call expression. Let’s have a look at a few methods in the class Supplier. For example, cell.StorageCONST(10) in the method SupplierCONST(int x, int y) is a field-method-call statement, and cell.GetStock() in the method GetTotalStock(InterfaceStorage x) is a field-method-call expression.

As we mentioned earlier when constructing the Omega-state, the state of the class Supplier is decided upon the following assignment in the method SupplierCONST(int x, int y).

- if cell = new Storage() appears, an instance of the class Storage will be assigned to the field cell. Accordingly, the field cell is modeled as an object of the record type of state SStorage,

```
Record SSupplier : Set := mk_SSupplier { sum : nat ;
  extra : nat ;
  cell :> SStorage }.
```

- if cell = new NewStorage() appears, an instance of the class NewStorage will be assigned to the field cell. Accordingly, the field cell is modeled as an object of the record type of state SNewStorage.

```
Record SSupplier : Set := mk_SSupplier { sum : nat ;
  extra : nat ;
  cell :> SNewStorage }.
```

After the field cell is modeled as an object of the record type of state SStorage or SNewStorage, the object’s fields are initialized with 0 or false depending on the fields’ types nat or bool.

As a consequence, we can see another exciting dynamic dispatch from the above subtype polymorphism. In the method FieldSetting(int n),
the field-method-call statement \texttt{cell.Setting(n)} executes the method \texttt{Setting(int n)} of the class \texttt{Storage} if an instance of the class \texttt{Storage} is assigned to the field \texttt{cell}, or

the field-method-call statement \texttt{cell.Setting(n)} executes the method \texttt{Setting(int n)} of the class \texttt{NewStorage} if an instance of the class \texttt{NewStorage} is assigned to the field \texttt{cell}.

Now, we have to model decidedly these field-method-call statements and expressions with record types and field-selection. However, we have a problem when interpreting them due to the fact that although an object assigned to the field \texttt{cell} includes method functions as well as fields, we have just modeled \texttt{cell} as an object of the record type of state \texttt{SStorage} or \texttt{SNewStorage} which do not have function fields. Hence, in order to overcome this problem, in the following we are defining the new record types \texttt{Omega_InterfaceStorage} and \texttt{Omega_InterfaceNewStorage} which increase the Omega-state with the interface-types \texttt{InterfaceStorage} or \texttt{InterfaceNewStorage}, and the coercion function \texttt{Omega_InterfaceNewStorage_TO_Omega_InterfaceStorage} between them.

\begin{verbatim}
(* Define Omega extended with the interface-type InterfaceStorage *)
Record Omega_InterfaceStorage : Set := mk_Omega_InterfaceStorage {
  label_omega_interfacestorage :> Omega ;
  label_interfacestorage : InterfaceStorage }.

(* Define Omega extended with the interface-type InterfaceNewStorage *)
Record Omega_InterfaceNewStorage : Set := mk_Omega_InterfaceNewStorage {
  label_omega_interfacenewstorage :> Omega ;
  label_interfacenewstorage : InterfaceNewStorage }.

(* Define a coercion between Omega_InterfaceStorage and Omega_InterfaceNewStorage *)
Coercion Omega_InterfaceNewStorage_TO_Omega_InterfaceStorage
  (x : Omega_InterfaceNewStorage) : Omega_InterfaceStorage :=
    mk_Omega_InterfaceStorage
    x.(label_omega_interfacenewstorage)
    x.(label_interfacenewstorage).
\end{verbatim}

Therefore, we have eventually got record types which include function-fields as well as state-fields like an instance of a class.

A field-method-call statement is then interpreted as a function taking an object of the record type \texttt{Omega_InterfaceStorage} and returning an object of the Omega-state if an instance of the class \texttt{Storage} is assigned to the field \texttt{cell}, or
A field-method-call statement is then interpreted as a function taking an object of the record type \texttt{Omega\_InterfaceNewStorage} and returning an object of the \texttt{Omega}\textendash{}state if an instance of the class \texttt{NewStorage} is assigned to the field \texttt{cell},

Similarly, a statement including a field-method-call expression is also modeled as a function taking an object of the record type \texttt{Omega\_InterfaceStorage} or \texttt{Omega\_InterfaceNewStorage} and returning an object of the \texttt{Omega}\textendash{}state. Let’s have a look at the interpretation of the method \texttt{FieldSetting(int n)} of the class \texttt{Supplier} carefully.

\begin{verbatim}
(* Modeling the method FieldSetting of the class Supplier *)
Let FieldSetting_Supplier : nat-> Omega->Omega := fun(n : nat)(w : Omega) =>
let w1 := w
in let CallStatement1_FieldSetting := fun (cw1 : Omega\_InterfaceNewStorage) =>
cw1.(label_interfacenewstorage).(i_label_Setting_NewStorage ) n cw1
in let w2 := mk_Omega\_InterfaceNewStorage w1
    (mk\_InterfaceNewStorage StorageCONST\_Supplier
    GetStock\_Supplier
    NewStorageCONST\_Supplier
    Decreasing\_Supplier
    (Setting\_Supplier GetStock\_Supplier)
    (Restore\_Supplier NewStorageCONST\_Supplier))
in let w3 := CallStatement1\_FieldSetting w2
in w3.
\end{verbatim}

In the above Coq codes, the new state \texttt{w2} is defined with the current state \texttt{w1} and the new functions

\begin{verbatim}
StorageCONST\_Supplier
GetStock\_Supplier
NewStorageCONST\_Supplier
Decreasing\_Supplier
Setting\_Supplier
Restore\_Supplier
\end{verbatim}

These functions should be defined before interpreting the class \texttt{Supplier} as follows. Readers might have a doubt why these functions have been defined repeatedly even though in the context there already exist the functions corresponding to the methods in the class \texttt{NewStorage}:
Let’s remind of the features of the *Omega*-state: the above existing functions in the context are expected to modify only the state of the class *NewStorage*. For that reason, we need to define new functions corresponding to the methods of the class *NewStorage* which are required to modify the state of the field *cell* in the class *Supplier*.

```coq
Let StorageCONST_Supplier : nat->Omega->Omega := fun(x : nat)(w : Omega) =>
let w1 := w
in let Assign1_StorageCONST := fun (aw1 : Omega) =>
  ( mk_Omega aw1.(label_storage)
  aw1.(label_newstorage)
  ( mk_SSupplier aw1.(label_supplier).(sum)
  aw1.(label_supplier).(extra)
  ( mk_SNewStorage ( mk_SStorage x)
    aw1.(label_supplier).(saleProduct) )
  aw1.(label_manager) )
in let w2 := Assign1_StorageCONST w1
in w2.

Let GetStock_Supplier : Omega-> nat := fun(w : Omega) =>
w.(label_supplier).(stock).

Let NewStorageCONST_Supplier : Omega-> Omega := fun(w : Omega) =>
let w1 := w
in let Assign1_NewStorageCONST := fun (aw1 : Omega) =>
  ( mk_Omega aw1.(label_storage)
  aw1.(label_newstorage)
  ( mk_SSupplier aw1.(label_supplier).(sum)
  aw1.(label_supplier).(extra)
  ( mk_SNewStorage (mk_SStorage aw1.(label_supplier).(stock))
    0 )
  aw1.(label_manager) )
in let w2 := Assign1_NewStorageCONST w1
in w2.

Fixpoint While_Decreasing_Supplier1 (STOCK : nat)(SALEPRODUCT : nat) { struct STOCK }
: nat :=
```
if (LessThan SALEPRODUCT STOCK )
then let diff := (minus STOCK SALEPRODUCT)
in (minus STOCK diff)
else STOCK.

Let Decreasing_Supplier :
Omega->Omega := fun(w : Omega) =>

let w1 := w
in let While1_Decreasing := fun (STOCK : nat)(SALEPRODUCT : nat)
=> While_Decreasing_Supplier1 STOCK SALEPRODUCT
in let before_stock := w1.(label_supplier).(stock)
in let before_saleProduct := w1.(label_supplier).(saleProduct)
in let after_stock := While1_Decreasing before_stock before_saleProduct
in let w2 := ( mk_Omega w1.(label_storage)
w1.(label_newstorage)
( mk_SSupplier w1.(label_supplier).(sum)
w1.(label_supplier).(extra)
( mk_SNewStorage ( mk_SStorage after_stock )
w1.(label_supplier).(saleProduct) )
)
w1.(label_manager) )
in w2.

Let Setting_Supplier
( GetStock : Omega->nat ) : nat ->Omega-> Omega:= fun (n : nat) (w : Omega) =>

let w1 := w
in let If1_Setting := fun (iw1 : Omega) =>
if ( LessThan ( GetStock iw1) n)
then let Assign1_Setting := fun (aw1 : Omega) =>
( mk_Omega aw1.(label_storage)
aw1.(label_newstorage)
( mk_SSupplier aw1.(label_supplier).(sum)
aw1.(label_supplier).(extra)
( mk_SNewStorage ( mk_SStorage aw1.(label_supplier).stock))
( GetStock aw1) )
)

in let Assign2_Setting := fun (aw2 : Omega) =>
( mk_Omega aw2.(label_storage)
aw2.(label_newstorage)
( mk_SSupplier aw2.(label_supplier).(sum)
aw2.(label_supplier).(extra)
( mk_SNewStorage ( mk_SStorage n )
aw2.(label_supplier).(saleProduct) )
)

in Assign2_Setting (Assign1_Setting iw1 )
else iw1
in let w2 := If1_Setting w1
in w2.

Let Restore_Supplier
( NewStorageCONST : Omega->Omega ) : Omega -> Omega := fun (w : Omega) =>

let w1 := w
in let Assign1_Restore := fun (aw1 : Omega) =>
( mk_Omega aw1.(label_storage)
aw1.(label_newstorage)
As a consequence, if going back to the full translation of the class `Supplier`, we can see the field-method-call statement `cell.Setting(n)` in the method `Field-Setting(int n)` be modeled with dependent record types and coercive subtyping as we intended: the function `Setting_Supplier` is modeled as taking an object of `Omega`; however, as seen in defining the function `FieldSetting_Supplier`, the field `i_label_Setting_NewStorage` takes an object of `Omega_InterfaceNewStorage`. The reason of it being well-typed is that the structural coercion of record projection was defined between `Omega_InterfaceNewStorage` and `Omega`.

Modeling type-casting

The field-method-call statement `((NewStorage)cell).NewStorageCONST()` in the method `SupplierCONST(int x, int y)` is using type-casting because although the field `cell` is declared of the type `InterfaceStorage`, an instance of the class `NewStorage` is assigned to the field `cell`. To put the point another way, because the field `cell` is declared of the type `InterfaceStorage`, it does not have information about the methods which will be declared in an interface-type inheriting `InterfaceStorage`.

Conspicuously, this type casting is easily modeled in our interpretation because the record type `Omega_InterfaceNewStorage` constitutes already the function field `NewStorageCONST_Supplier`.

Modeling a return-value of interface-type

In the class `Supplier`, the method `GetCell()` is expected to return an object of the interface-type `InterfaceStorage`. Put otherwise, by subtype-polymorphism,

- the method actually returns an instance of the class `Storage` implementing `InterfaceStorage` if the field `cell` is initialized with an instance of the class `Storage` in the method `SupplierCONST()`.
• the method actually returns an instance of the class *NewStorage* implementing *InterfaceNewStorage* if the field *cell* is initialized with an instance of the class *NewStorage* in the method *SupplierCONST()*.

We are modeling the return-type of this method as returning an object of the record type of state *SStorage* or *SNewStorage* depending on the following assignment in the method *SupplierCONST()*:

- *SStorage* if *cell = new Storage()*
- *SNewStorage* if *cell = new NewStorage()*

**Modeling field-method-call statements with synchronization**

The class *Manager* has also a field of interface-type: the field *copyCell* is declared of *InterfaceStorage*. Hence, *copyCell* is modeled as an object of the record-type of state *SStorage* or *SNewStorage* depending on the following assignment with the explicit-method-call expression in the method *ManagerCONST()*,

\[
\text{copyCell} = \text{y.GetCell()}
\]

Here, we should recognize that the field *copyCell* is just a reference to the field *cell* in the class *Supplier*, because an object the explicit-method-call expression *y.GetCell()* returns is assigned to the field *copyCell*. Simply put, a memory space on the hardware is not made for the field *copyCell*. In spite of this fact, we model the field *copyCell*:

- as an object of the record-type of state *SStorage* if the expression *y.GetCell()* returns an object of the class *Storage*, or
- as an object of the record-type of state *SNewStorage* if the expression *y.GetCell()* returns an object of the class *NewStorage*.

Unquestionably, we must therefore synchronize the field *cell* in the record type of state *SSupplier* and the field *copyCell* in the record type of state *SManager*.

Next, we need to define the following functions again corresponding to the methods in the class *NewStorage*, in order to modify the state of the field *copyCell* in the class *Manager*. 
Let StorageCONST_Manager
: nat -> Omega -> Omega := fun(x : nat)(w : Omega) =>
let w1 := w
in let Assign1_StorageCONST := fun(aw1 : Omega) =>
  ( mk_Omega aw1.(label_storage)
  aw1.(label_newstorage)
  aw1.(label_supplier)
  ( mk_SManager aw1.(label_manager).(balance)
    ( mk_SNewStorage ( mk_SStorage x )
      aw1.(label_manager).(saleProduct))))
in let w2 := Assign1_StorageCONST w1 in w2.

Let GetStock_Manager
: Omega -> nat := fun(w : Omega) =>
w.(label_manager).(stock).

Let NewStorageCONST_Manager
: Omega -> Omega := fun(w : Omega) =>
let w1 := w
in let Assign1_NewStorageCONST := fun(aw1 : Omega) =>
  ( mk_Omega aw1.(label_storage)
  aw1.(label_newstorage)
  aw1.(label_supplier)
  ( mk_SManager aw1.(label_manager).(balance)
    ( mk_SNewStorage ( mk_SStorage aw1.(label_manager).(stock) )
      0 ) )
in let w2 := Assign1_NewStorageCONST w1 in w2.

Fixpoint While_Decreasing_Manager1
(STOCK : nat)(SALEPRODUCT : nat) { struct STOCK } : nat :=
if (LessThan SALEPRODUCT STOCK )
then let diff := (minus STOCK SALEPRODUCT)
in (minus STOCK diff)
else STOCK.

Let Decreasing_Manager :
Omega -> Omega := fun(w : Omega) =>
let w1 := w
in let While1_Decreasing := fun(STOCK : nat)(SALEPRODUCT : nat)
  => While_Decreasing_Manager1 STOCK SALEPRODUCT
in let before_stock := w1.(label_manager).(stock)
in let before_saleProduct := w1.(label_manager).(saleProduct)
in let after_stock := While1_Decreasing before_stock before_saleProduct
in let w2 :=
  ( mk_Omega w1.(label_storage)
  w1.(label_newstorage)
  w1.(label_supplier)
  ( mk_SManager w1.(label_manager).(balance)
    ( mk_SNewStorage ( mk_SStorage after_stock )
      w1.(label_manager).(saleProduct))))
in w2.
Let \( \text{Setting\_Manager} \)
\[
\begin{align*}
\text{GetStock} : \Omega \to \mathbb{N} &; \forall n : \mathbb{N} \rightarrow \text{GetStock} : \Omega \rightarrow \mathbb{N} \rightarrow \Omega \rightarrow \Omega := \text{fun } (n : \mathbb{N} ; w : \Omega) \rightarrow \\
\text{let } w := w &; \\
in \text{let } \text{If1\_Setting} := \text{fun } (iw1 : \Omega) \rightarrow \\
\text{if } (\text{LessThan} \text{(GetStock } iw1) n) &; \\
\text{then let } \text{Assign1\_Setting} := \text{fun } (aw1 : \Omega) \rightarrow \\
(\text{mk\_Omega } aw1.(\text{label\_storage}) &; \\
aw1.(\text{label\_newstorage}) &; \\
aw1.(\text{label\_supplier}) &; \\
(\text{mk\_SManager } aw1.(\text{label\_manager}).(\text{balance}) &; \\
(\text{mk\_SNewStorage } \text{(mk\_SStorage } aw1.(\text{label\_manager}).(\text{stock})) &; \\
(\text{GetStock } aw1)))) &; \\
in \text{let } w2 := \text{If1\_Setting } w1 &; \\
in \text{let } w := w2 &; \\
\text{else } iw1 &; \\
in \text{let } w := \text{If1\_Setting } w1 &; \\
in w2.
\end{align*}
\]

Let \( \text{Restore\_Manager} \)
\[
\begin{align*}
\text{NewStorageCONST : } \Omega \rightarrow \Omega &; \forall x : \Omega \rightarrow \Omega := \text{fun } (x : \Omega) \rightarrow \\
\text{let } w := w &; \\
in \text{let } \text{Assign1\_Restore} := \text{fun } (aw1 : \Omega) \rightarrow \\
(\text{mk\_Omega } aw1.(\text{label\_storage}) &; \\
aw1.(\text{label\_newstorage}) &; \\
aw1.(\text{label\_supplier}) &; \\
(\text{mk\_SManager } aw1.(\text{label\_manager}).(\text{balance}) &; \\
(\text{mk\_SNewStorage } \text{(mk\_SStorage } n) &; \\
aw2.(\text{label\_manager}).(\text{saleProduct})))) &; \\
in \text{let } w := \text{Assign1\_Restore } w1 &; \\
in \text{let } \text{CallStatement1\_Restore} := \text{fun } (cw1 : \Omega) \rightarrow \text{NewStorageCONST } cw1 &; \\
in \text{let } w := \text{CallStatement1\_Restore } w2 &; \\
in w3.
\end{align*}
\]

The following is the interpretation of the class \textit{Manager}. In particular,
during modeling the method \textit{SettingCopyCell(int \( n \))}, the new state \( w4 \) is defined for synchronizing the field \textit{cell} in the record type of state \textit{SSupplier}
and the field \textit{copyCell} in the record type of state \textit{SManager}.

\[
\begin{align*}
\text{Manager\_CONST} \text{Manager} &; \\
\text{ManagerCONST\_Manager} := \text{nat} \rightarrow \text{InterfaceSupplier} \rightarrow \Omega \rightarrow \Omega := \text{fun } (x : \mathbb{N} ; y : \text{InterfaceSupplier} ; w : \Omega) \rightarrow \\
\text{let } w := w &; \\
in \text{let } \text{Assign1\_ManagerCONST} := \text{fun } (aw1 : \Omega) \rightarrow \\
(\text{mk\_Omega } aw1.(\text{label\_storage}) &; \\
aw1.(\text{label\_newstorage}) &; \\
aw1.(\text{label\_supplier}) &; \\
(\text{mk\_SManager } aw1.(\text{label\_manager}).(\text{balance}) &; \\
(\text{mk\_SNewStorage } \text{(mk\_SStorage } aw1.(\text{label\_newstorage}).(\text{saleProduct})) &; \\
aw1.(\text{label\_manager}).(\text{saleProduct})))) &; \\
in \text{let } w := \text{Assign1\_ManagerCONST } w1 &; \\
in \text{let } \text{CallStatement1\_Manager} := \text{fun } (cw1 : \Omega) \rightarrow \text{NewStorageCONST } cw1 &; \\
in \text{let } w := \text{CallStatement1\_Manager } w2 &; \\
in w3.
\end{align*}
\]
aw1.(label_newstorage)
aw1.(label_supplier)
( mk_SManager x
  aw1.(label_manager).(copyCell) )

in let w2 := Assign1_ManagerCONST w1
in let w3 := Assign2_ManagerCONST := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_storage)
    aw2.(label_newstorage)
    aw2.(label_supplier)
    ( mk_SManager aw2.(label_manager).(balance)
      y.(i_label_GetCell_Supplier ) aw2 ) )

in let w3 := Assign2_ManagerCONST w2
in w3.

(* Modeling the method FindBalance of the class Manager *)
Let FindBalance_Manager
: InterfaceStorage -> Omega->Omega := fun(x : InterfaceStorage)(w : Omega) =>
  let w1 := w in
  let Assign1_FindBalance := fun (aw1 : Omega_InterfaceNewStorage) =>
  (mk_Omega aw1.(label_storage)
    aw1.(label_newstorage)
    aw1.(label_supplier)
    (mk_SManager
      (aw1.(label_manager).(balance) *
        ((( x.(i_label_GetStock_Storage ) aw1) *
          (aw1.(label_interfaceNewstorage).(i_label_GetStock_NewStorage) aw1)) * 10))
    aw1.(label_manager).(copyCell) )
  )
  in let w2 := mk_Omega_InterfaceNewStorage w1
  (mk_InterfaceNewStorage
    StorageCONST_Manager
    GetStock_Manager
    NewStorageCONST_Manager
    Decreasing_Manager
    (Setting_Manager GetStock_Manager)
    (Restore_Manager NewStorageCONST_Manager))
  in let w3 := Assign1_FindBalance w2
  in w3.

(* Modeling the method SettingCopyCell of the class Manager *)
Let SettingCopyCell_Manager
: nat-> Omega->Omega := fun(n : nat)(w : Omega) =>
  let w1 := w in
  let w2 := mk_Omega_InterfaceNewStorage w1
  (mk_InterfaceNewStorage
    StorageCONST_Manager
    GetStock_Manager
    NewStorageCONST_Manager
    Decreasing_Manager
    (Setting_Manager GetStock_Manager)
    (Restore_Manager NewStorageCONST_Manager))
  in let w3 := CallStatement1_SettingCopyCell w2
  in let w4 := ( mk_Omega w3.(label_storage)
2.2 Verifying specifications of myJava programs in Coq

Now that we have defined lots of datatypes and functions corresponding to the myJava programs `Storage`, `NewStorage`, `Supplier`, and `Manager`. Let's turn to the questions of how to state and prove properties of these programs’ behavior. The verification of myJava programs which can embody specifications inside `/* ... */` can be 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 less than or equal to the `saleProduct` value. To put it more concretely, a class invariant is a statement about an object that is true after every constructor and that is preserved by every mutator provided that the caller respects all preconditions. Formally, we prove this invariant proposition...
for every method defined in the class `NewStorage` except a constructor and an accessor method.

```
Theorem Invariant_Decreasing_NewStorage : forall (r : NewStorage) (s : Omega), True ->
  (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  let s' := (r.(label_Decreasing_NewStorage) : ID (Omega->Omega) ) s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).

Theorem Invariant_Setting_NewStorage : forall (r : NewStorage) (s : Omega) (n : nat) , True ->
  (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  let s' := (r.(label_Setting_NewStorage) : ID (nat->Omega->Omega) ) n s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).

Theorem Invariant_Restore_NewStorage : forall (r : NewStorage) (s : Omega), True ->
  (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  let s' := (r.(label_Restore_NewStorage) : ID (Omega->Omega) ) s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).
```

The above generated theorems by JaCo are proved as follows,

```
Axiom Default_stock : forall (s:Omega), 0 <= s.(label_newstorage ).(stock).
Axiom Default_saleProduct : forall (s : Omega), 0 <= s.(label_newstorage ).(saleProduct).
Lemma While_case1 : forall (s : Omega),
  (s.(label_newstorage ).(saleProduct)) < (s.(label_newstorage ).(stock)) ->
  (s.(label_newstorage ).(saleProduct)) <=
  While_Decreasing_NewStorage1 (s.(label_newstorage ).(stock))
  (s.(label_newstorage ).(saleProduct)).
Admitted.
Lemma While_case2 : forall (s : Omega),
  (s.(label_newstorage ).(saleProduct)) = (s.(label_newstorage ).(stock)) ->
  (s.(label_newstorage ).(saleProduct)) <=
  While_Decreasing_NewStorage1 (s.(label_newstorage ).(stock))
  (s.(label_newstorage ).(saleProduct)).
Admitted.
Definition gt_eq_dec n m : {m > n} + {n = m}.
Admitted.
```

Theorem Invariant_Decreasing_NewStorage : forall (r : NewStorage) (s : Omega), True ->
  (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  let s' := (r.(label_Decreasing_NewStorage) : ID (Omega->Omega) ) s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).
Proof.
intros r s.
intros H1.
intros H2.
simpl.
elim (gt_eq_dec (s.(label_newstorage ).(saleProduct)) (s.(label_newstorage ).(stock)) ).
CHAPTER 2. VERIFICATION OF MYJAVA PROGRAMS IN COQ

Case " saleProduct < stock ".
  intros H3.
  apply While_case1 in H3.
  assumption.
Case " saleProduct = stock ".
  intros H3.
  apply While_case2 in H3.
  assumption.
Qed.

****************************************************************************************

Lemma LessThan_true : forall (s:Omega) (n:nat),
  s.(label_newstorage).(stock) < n ->
  (LessThan s.(label_newstorage).(stock) n) = true.
Admitted.

Lemma LessThan_false : forall (s:Omega) (n:nat),
  n <= s.(label_newstorage).(stock) ->
  (LessThan s.(label_newstorage).(stock) n) = false.
Admitted.

Lemma If_true : forall (r:NewStorage) (s:Omega) (n:nat),
  (LessThan s.(label_newstorage).(stock) n) = true ->
  let s' := (r.(label_Setting_NewStorage) : ID (nat->Omega->Omega) ) n s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).
Admitted.

Lemma If_false : forall (r:NewStorage) (s:Omega) (n:nat),
  (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  (LessThan s.(label_newstorage).(stock) n) = false ->
  let s' := (r.(label_Setting_NewStorage) : ID (nat->Omega->Omega) ) n s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).
Admitted.

****************************************************************************************

Theorem Invariant_Setting_NewStorage : forall (r : NewStorage) (s : Omega) (n : nat),
  True -> (s.(label_newstorage ).(saleProduct)) <= (s.(label_newstorage ).(stock)) ->
  let s' := (r.(label_Setting_NewStorage) : ID (nat->Omega->Omega) ) n s
  in (s'.(label_newstorage ).(saleProduct)) <= (s'.(label_newstorage ).(stock)).
Proof.
  intros r s n.
  intros H0 H1.
  simpl.
  elim (le_lt_dec n s.(label_newstorage).(stock)).
  Case " n <= stock ".
    intro H2.
    apply If_false.
    exact H1.
    apply LessThan_false.
    exact H2.
  Case " stock < n ".
    intro H2.
    apply If_true.
    apply LessThan_true.
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 \texttt{Supply(InterfaceStorage \textit{x})} of the class \texttt{Supplier} with the state \textit{extra} that is greater than zero, the method should definitely return a new state where \textit{extra} is equal to zero. The following is the generated theorem and its proof.

\begin{center}
\begin{verbatim}
Theorem PrePost_Supply_Supplier : forall (r : Supplier)(s : Omega) (x : InterfaceStorage),
    (0) < (s.(label_supplier).(extra)) ->
    let s' := (r.(label_Supply_Supplier) : ID (InterfaceStorage -> Omega -> Omega )) x s
    in (s'.(label_supplier).(extra)) = (0).
Proof.
intros r s x.
intros H1.
simpl.
reflexivity.
Qed.
\end{verbatim}
\end{center}
Chapter 3

UTT

In this chapter, we give a formal and brief presentation of UTT \(^1\) [Luo94] extended with coercive subtyping [Luo99] and dependent record types [Luo09]. UTT is an intensional and dependent type theory specified by a typed version of Martin-Löf’s logical framework [NPS90], which is a kind of meta-languages specifying object-languages by using type theories. The framework of coercive subtyping and dependent record types will be explored based on the logical framework LF. UTT includes an internal logic, a large class of inductive data types generated by inductive schemata, and universes.

3.1 The logical framework LF

Our formulation of dependent record types and the framework of coercive subtyping is based on the logical framework LF [Luo94]. LF is the typed version of Martin-Löf’s logical framework [NPS90]. It is a dependent type system that can be used to specify type theories such as Martin-Löf’s intensional type theory [NPS90] and UTT [Luo94]. The types in LF are called *kinds*, in order to be distinguished from the types in the specified type theories such as ECC [Luo90][Luo94], Calculus of Construction [CH88], and Martin-Löf’s type theory [ML84] [NPS90].

Terms

- *Type* - the kind representing the conceptual universe of types, i.e. \(A\) is a type if \(A : Type\)

\(^1\)UTT is short for Unifying Theory of dependent Types.
• \( El(A) \) - the kind of objects of type \( A \). We often omit the kind forming operator \( El \).

• \((x:K)K'\) or \((x:K)K'[x]\) - the dependent product kind for any kind \( K \) and any family of kinds \( K'[x] \) indexed by objects of kind \( K \). We often write \((K)K'\) instead of \((x:K)K'\) when \( x \) does not occur free in \( K' \).

• \([x:K]k'\) - the canonical object of the dependent product kind \((x:K)K'\), in other words, the functional operation which for any object \( k \) of kind \( K \), yields objects of kind \( K'[k] \), where the free occurrences of variable \( x \) in \( K' \) and \( k' \) are bound by the binding operators \((x:K)\) and \([x:K]\), respectively.

• \( f(k) \) - the application of a functional operation \( f \) of kind \((x:K)K'\) to an object \( k \) of kind \( K \).

Judgements and Inference rules

• \( \Gamma valid \), which asserts that \( \Gamma \) is a valid context,

• \( \Gamma \vdash K kind \), which asserts that \( K \) is a kind,

• \( \Gamma \vdash k : K \), which asserts that \( k \) is an object of kind \( K \),

• \( \Gamma \vdash k = k' : K \), which asserts that \( k \) and \( k' \) are equal objects of kind \( K \) (Judgemental Equality), and

• \( \Gamma \vdash K = K' \), which asserts that \( K \) and \( K' \) are equal kinds.

The rules for inferring judgements in LF are given in Appendix D. Additionally, we need two more inference rules in order to specify type theories in LF. In general, a specification of a type theory in LF will consist of a collection of declarations of new constants and a collection of computation rules. First, we declare a new constant \( k \) of kind \( K \) as follows,

\[
k : K
\]

which means that we extend the type theory specified in LF by adding the following new inference rule.

\[
\begin{array}{c}
\Gamma valid \\
\hline
\Gamma \vdash k : K
\end{array}
\]
Secondly, we assert a computation rule usually about the new constants,

\[ k = k' : K \]

which represents that we add the following new inference rule to the type theory.

\[
\Gamma \vdash k_i : K_i(i = 1, \ldots, n) \quad \Gamma \vdash k : K \quad \Gamma \vdash k' : K \\
\Gamma \vdash k = k' : K
\]

**Notation**

- **Equality signs**: We shall often use \( M \equiv N \) for the syntactical identity, meaning that the terms \( M \) and \( N \) are the same up to \( \alpha \)-conversion. \( M = N \) is for the definitional and computational equality between \( M \) and \( N \).

- **Substitution**: \([N/x]M\) stands for the term obtained from substitution \( N \) for the free occurrences of \( x \) in \( M \), defined as usual with the possible change of bound variables. We sometimes use \( M[x] \) to indicate that variable \( x \) may occur free in \( M \) and subsequently write \( M[N] \) for \([N/x]M\), when no confusion may occur.

- **Composition**: For \( f : (K_1)K_2 \) and \( g : (K_2)K_3 \), we define

\[ g \circ f \stackrel{=}{=}_{df} [x:K_1]g(f(x)) : (K_1)K_3 \]

where \( x \) does not occur free in \( f \) and \( g \).

**Definitional vs. Computational Equality**

We shall say that two objects of the same kind are definitionally equal if they are identical subject to the meta-level \( \beta \eta \)-equality. That is, for any type theory \( T \) specified in the logical framework, two objects \( k \) and \( k' \) of kind \( K \) are definitionally equal in \( T \), by the following notation,

\[ \Gamma \vdash^T k =_{\beta \eta} k' : K \]
if and only if $\Gamma \vdash k : K$ and $\Gamma \vdash k' : K$ are derivable in $T$, and $k$ and $k'$ are
convertible with regard to the following contraction rules:

$$\begin{align*}
[x : K] f(x) & \triangleright_\eta f \quad (x \notin FV(f)) \\
([x : K] k')(k) & \triangleright_\beta [k/x] k'
\end{align*}$$

Therefore, if two objects in $T$ are definitionally equal, they are certainly computationally equal. We note that two objects $k$ and $k'$ of the same kind $K$ in
the type theory UTT are computationally equal if for some valid context $\Gamma$, the
judgement $\Gamma \vdash k = k' : K$ is derivable in UTT. In the intensional type theory UTT, every well-typed term has a unique normal form. If $\Gamma \vdash k = k' : K$ is
derivable, the $k$ and $k'$ can be computed to the same normal form. Therefore, we
can say that $k$ and $k'$ are computationally equal if they are well-typed and have
the same normal form.

### 3.2 Specifying UTT in LF

When the type theory UTT is specified in LF, its types are declared, together
with their introduction/elimination operators and the associated computation
rule. Examples include inductive types such as $Nat$ of natural numbers, in-
ductive families of types such as $Vect(n)$ of vectors of length $n$, and families of
inductive types such as $\Pi$-types and $\Sigma$-types. We note that inductive types can be
parameterized. For instance, $\Pi$-types and $\Sigma$-types are parameterized by $A : Type$
and $B : (A)Type$

- $\Pi(A, B)$ is the $\Pi$-type of the functions $f \equiv \lambda(x : A)b$ that, when applied to
  any object $a$ of type $A$, returns $f(a) = [a/x]b$ of type $B(a)$.

  **Formation rule**

  $$\Gamma \text{ valid} \quad \frac{}{\Gamma \vdash \Pi(A : Type)((A)Type)Type}$$

  \footnote{We overload the notation for functional application: $f(a)$ denotes both applications for
dependent product kinds and $\Pi$-types. There should be no confusions in the contexts.}
Introduction rule

\[ \Gamma \text{ valid} \]
\[ \Gamma \vdash \lambda (A:Type)(B:(A)Type)((x:A)B(x))\Pi(A,B) \]

Elimination rule

\[ \Gamma \text{ valid} \]
\[ \Gamma \vdash \text{app}:(A:Type)(B:(A)Type)(\Pi(A,B))(a:A)(B(a)) \]

Computation rule

\[ \Gamma \vdash A:Type, B:(A)Type, \lambda (A,B,f):\Pi(A,B), a:A \]
\[ \Gamma \vdash \text{app}(A,B,\lambda (A,B,f),a) = f(a):B(a) \]

- \( \Sigma(A,B) \) is the \( \Sigma \)-type of pairs with \( \pi_1(A,B) \) and \( \pi_2(A,B) \) being the associated projection operators.

Next, we introduce the most important type for this thesis, which is the inductively defined parameterized unit type. In our type-theoretic models of object-oriented programs, all methods of classes will be interpreted as a unit type.

\[ \text{Unit} : (A:Type)(x:A)Type \]
\[ \text{unit} : (A:Type)(x:A)\text{Unit}(A,x) \]

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

Remark LF specifies intensional type theories, which have nice meta-theoretical properties such as the decidability of type-checking. For instance, the inductive types specified in LF do not have the \( \eta \)-like equality rules. As an example, the above unit type is different from the singleton type [Asp95] in that, for a variable \( x:\text{Unit}(A,a) \), \( x \) is not computationally equal to \( \text{unit}(A,a) \).
CHAPTER 3. UTT

3.3 Coercive subtyping

Coercive subtyping for dependent type theories has been developed as a general approach to abbreviation, subtyping, and inheritance in type theories with inductive types [Luo97] [Luo99] [Luo05] [Bai99]. Now, we extend UTT with coercive subtyping which is suitable not only for structural subtyping, but also for non-structural subtyping such as the coercion for the parameterized unit type. It has been implemented and used in the proof assistants Coq [Coq] [Sai97], Lego [LP92] [Bai99] and Plastic [CL01].

Judgements and Rules

We have two new forms of judgements in UTT with coercive subtyping. In coercive subtyping, a kind $K$ is a subkind of kind $K'$ if there is a coercion (functional operation) $c : (K)K'$ from $K$ to $K'$, which is unique up to computational equality, in other words, if $K \leq_c K'$ and $K \leq_{c'} K'$, then $c = c' : (K)K'$.

- **Subkinding judgement**: $\Gamma \vdash K \leq_c K'$ asserts that kind $K$ is a proper subkind of kind $K'$ with a unique coercion $c$, and

- **Subtyping judgement**: $\Gamma \vdash A \leq_c B : Type$ asserts that type $A$ is a proper subtype of type $B$ with a unique coercion $c$, which means that $El(A) \leq_c El(B)$.

Formally, $UTT[\mathcal{R}]$ is the extension of UTT with coercive subtyping specified by the rules in $\mathcal{R}$. It is usually presented in two stages:

- The system $UTT[\mathcal{R}]_0$ is the extension of UTT with the subtyping judgements and the rules in $\mathcal{R}$ together with the congruence, substitution, and transitivity rules for subtyping judgements. Also, in this paper, we assume that the identity is always a coercion, where $id_A \equiv [x:A]x$

  $\Gamma \vdash A : Type$

  $\Gamma \vdash A \leq_{id_A} A : Type$

- The system $UTT[\mathcal{R}]$ is the extension of $UTT[\mathcal{R}]_0$ with the subkinding judgements. In $UTT[\mathcal{R}]$, subtyping is lifted to subkinding

  $\Gamma \vdash A \leq_c B : Type$

  $\Gamma \vdash El(A) \leq_c El(B)$
In the theoretical framework of coercive subtyping, the role of \( c \) is represented by the coercive definition rule:

\[
\frac{f : (x : K) K' \quad k_0 : K_0 \quad K_0 \leq c \ K}{f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}
\]

which says that if \( f \) is a functional operation from \( K \) to \( K' \), \( k_0 \) is an object of \( K_0 \), and \( c \) is a coercion from \( K_0 \) to \( K \), then \( f(k_0) \) is well typed and definitionally equal to \( f(c(k_0)) \). In other words, if \( K_0 \) is a subtype of \( K \) via coercion \( c \), the any object \( k_0 \) of type \( K_0 \) can be regarded as the object \( c(k_0) \) of type \( K \). For a formal presentation of coercive subtyping with the complete rules, see [Luo99].

**Coherence**

Coercions may be declared by the users. Coherence is a crucial property of coercions for them to be used correctly. It essentially says that the coercions between any two types are unique up to the computational equality.

**Definition 3.3.1 (Coherence)** A set \( \mathcal{R} \) of coercion rules is coherent if the following two rules are admissible in UTT\[\mathcal{R}\]_0

\[
\begin{align*}
\Gamma \vdash A \leq c \ A : Type & \quad \Gamma \vdash A \leq c \ B : Type \\
\Gamma \vdash c = id_A : (A)A & \quad \Gamma \vdash c = c' : (A)B
\end{align*}
\]

### 3.4 Dependent record types

In this section, we shall formulate dependent record types and structural subtyping for them in the framework of coercive subtyping [Luo09, Luo99] as an extension of UTT which is specified in the logical framework LF [Luo94]. In the literature, there are several formulations of dependent record types [HL94, BT98, Pol02, CPT05] on which ours is based. In our formulation, record types are independent with structural subtyping whereas all of the previous formulations have made an essential use of structural subtyping except for [Pol02]. We shall follow [BT98] to introduce a special kind of record types and do not allow repetition of field labels in a record type different from [Pol02].
### 3.4.1 Terms

The syntax is extended with record types and records of the form:

\[ R = : \langle \rangle | \langle R, l : A \rangle \]
\[ r = : \langle \rangle | \langle r, l = a : A \rangle \]

where we overload the syntax \( \langle \rangle \) to stand for both the empty record type and its only object (the empty record).

For every finite set of labels \( L \), we introduce a kind \( RTypel \), the kind of the record types whose top level labels are all in \( L \). We shall also introduce the kind \( RTypel \) of all record types. These kinds obey obvious subkinding relationships:

\[
\begin{align*}
\Gamma &\vdash R : RTypel \quad L \subseteq L' \\
\Gamma &\vdash R : RTypel \quad L \subseteq L' \\
\Gamma &\vdash R : RTypel \quad L \subseteq L' \\
\Gamma &\vdash R : Type
\end{align*}
\]

### 3.4.2 Inference rules

Records are associated with two operations: restriction \([r]\) that removes the last component of record \( r \) and field selection \( r.l \) that selects the field labeled \( l \).

**Formation rules**

\[
\begin{align*}
\Gamma &\vdash \text{valid} & & \Gamma &\vdash R : RTypel \quad \Gamma &\vdash A : \text{Type} \quad l \notin L \\
\Gamma &\vdash \langle \rangle : RTypel[l] & & \Gamma &\vdash \langle R, l : A \rangle : RTypel[l \cup \{l\}]
\end{align*}
\]

**Introduction rules**

\[
\begin{align*}
\Gamma &\vdash \text{valid} & & \Gamma &\vdash \langle R, l : A \rangle : RTypel \quad \Gamma &\vdash r : R \\
\Gamma &\vdash \langle \rangle : \langle \rangle & & \Gamma &\vdash \langle r, l = a : A \rangle : \langle R, l : A \rangle
\end{align*}
\]

**Elimination rules**

\[
\begin{align*}
\Gamma &\vdash r : \langle R, l : A \rangle & & \Gamma &\vdash r : \langle R, l : A \rangle \\
\Gamma &\vdash \langle R, l : A \rangle & & \Gamma &\vdash \langle r, l : A \rangle & & \Gamma &\vdash \langle \rangle \quad \Gamma &\vdash \langle r, [r], l' : B \rangle \\
\Gamma &\vdash \langle r \rangle & & \Gamma &\vdash \langle r, l : A \rangle & & \Gamma &\vdash \langle r, l : A \rangle \quad \Gamma &\vdash \langle \rangle \quad \Gamma &\vdash \langle r, l = a : A \rangle \\
\end{align*}
\]

**Computation rules**

\[
\begin{align*}
\Gamma &\vdash \langle r, l = a : A \rangle : \langle R, l : A \rangle & & \Gamma &\vdash \langle r, l = a : A \rangle : \langle R, l : A \rangle \\
\Gamma &\vdash \langle r \rangle & & \Gamma &\vdash \langle \rangle \quad \Gamma &\vdash \langle r, l = a : A \rangle & & \Gamma &\vdash \langle r, l = a : A \rangle \quad \Gamma &\vdash \langle r, l = a : A \rangle \quad \Gamma &\vdash \langle r, l = a : A \rangle
\end{align*}
\]
3.4.3 Coercive subtyping for dependent record types

We introduce coercions that express structural subtyping relationships between
record types where the domain and codomain of a coercion should be both record
types. We see a new judgement form

\[ \Gamma \vdash R \leq_c R' : RType \]

together with the rule lifting the subtyping relation from \( RType \) to \( Type \):

\[ \Gamma \vdash R \leq_c R' : RType \]

\[ \Gamma \vdash R \leq_c R' : Type \]

We consider two kinds of structural subtyping rules for record types: in par-
ticular, we shall take record projections as non-dependent coercions.

Record projections: The field selections can be obtained by compositions of
the following two projection operations. For a non-empty record type
\( \langle R, l : A \rangle \),

1. the first projection is simply the restriction operation

\[ \lfloor \cdot \rfloor : \langle R, l : A \rangle R \]

mapping \( \langle r, l = a \rangle \) to \( r \).

\[ \Gamma \vdash \langle R, l : A \rangle : RType \]

\[ \Gamma \vdash \lfloor \langle R, l : A \rangle \rfloor \leq_c \lfloor R : RType \rfloor \]

2. the second projection is the functional operation.

\( \text{Snd} : (r : \langle R, l : A \rangle) \rightarrow l:A([r]) \rangle \)

mapping \( r \) to the record \( \langle l = r.l \rangle \) where the codomain type of \( \text{Snd} \)
should be the record type \( \langle l : A([r]) \rangle \) rather than \( A([r]) \) in order to
keep the coercion coherence.

\[
\Gamma \vdash A : \text{Type} \quad \Gamma \vdash \langle R, \, l : A \rangle : \text{RT}ype
\]

\[
\Gamma \vdash \langle R, \, l : A \rangle \leq_{\text{snd}} \langle l : A \rangle : \text{RT}ype
\]

where the label \( l \) in the codomain type of \( \text{Snd} \) is the same as the label \( l \) in its domain type.

**Component-wise coercion:** Component-wise coercions express the idea that coercive subtyping relations propagate through the dependent record types. If \( R \) is a subtype of \( R' \) and \( A \) is a subtype of \( A' \), then \( \langle R, \, l : A \rangle \) is a subtype of \( \langle R', \, l : A' \rangle \).

\[
\begin{align*}
\Gamma & \vdash A : (R)\text{Type} \quad \Gamma & \vdash A' : (R')\text{Type} \quad l \notin R \quad l \notin R' \\
\Gamma & \vdash R \leq_e R' : \text{RT}ype \quad \Gamma, \, x : R \vdash A(x) \leq_{c[x]} A'(c(x)) : \text{Type}
\end{align*}
\]

\[
\Gamma \vdash \langle R, \, l : A \rangle \leq_d \langle R', \, l : A' \rangle : \text{RT}ype
\]

where \( d \) maps a record \( \langle r, \, l = a \rangle \) to \( \langle c(r), \, l = c'[r](a) \rangle \) and is formally defined as, for any \( r' : \langle R, \, l : A \rangle \) and \( r_0 \equiv [r'] \),

\[
d(r') =_{df} \langle c(r_0), \, l = c'[r_0](r'.l) \rangle
\]

Note that a component-wise coercion can exist only between the record types that have the same corresponding labels.

**Remark** Coherence is a crucial property of coercion that guarantees the logical consistency and the nice meta-theoretic properties of the subtyping extensions. We see that for record types the projection coercions and component-wise coercion together are coherent [Luo09].
Chapter 4

Semantics of functional interpretation

In this chapter, it will be our purpose to define the type-theoretical semantics of myJava programs in UTT, taking a closer look at the BNF syntax shown in Figure 1.2 and Figure 1.3.

4.1 The state of objects

As discussed earlier in Chapter 2, the reason that the Ω-state is most importantly needed for our interpretation is to handle the contra-variant problem. As well, the Ω-state should be able to capture simultaneously the state of all objects in the context, because we are doing a functional interpretation which does not have an idea of changing stored values on the hardware memory. More remarkable is for the Ω-state even to capture tree graphs of inheritance relationships Java supports.

We shall now intend to focus more theoretically on the process of how the Ω-state is constructed from the state of each class in the context. The UML class diagram shown in Figure 4.1 constitutes the five classes $A_1$, $A_2$, $A_3$, $A_4$ and $B$.

As we interpret the state of these classes in sequence with their field declaration, we shall see how the state of an object is modeled generally.

The state of classes is interpreted by $\mathcal{F}$, so that the state $St(A_1)$ of the class

---

1In our model, we must assume that these classes are declared in the sequence of $A_1$, $A_2$, $A_3$, $A_4$ and $B$. This sequence will be modeled in the subscript of the Ω-definition. However, in Java programming the order of declaring classes is not important.
\(A_1\) which has \(n\) fields is defined as follows,

\[
\mathcal{F}[\text{FieldVarDecl}^{A_1}] =_{df} ( id^{A_1}_{field_1} : \zeta[type_{field_1}] , id^{A_1}_{field_2} : \zeta[type_{field_2}] , \ldots , \\
id^{A_1}_{field_n} : \zeta[type_{field_n}] )
\]

\[
St(A_1) \equiv \mathcal{F}[\text{FieldVarDecl}^{A_1}]
\]

where

- \(\text{FieldVarDecl}^{A_1}\) denotes the set of the fields declared in the class \(A_1\) as follows,

\[
\text{FieldVarDecl}^{A_1} \equiv \{ \text{protected type}_{field_1} \ id^{A_1}_{field_1} ; \ldots ; \text{protected type}_{field_n} \ id^{A_1}_{field_n} \}
\]

- \(\zeta[type_{field}] =_{df} \begin{cases} 
\text{nat} & \text{if type}_{field} \text{ is int} \\
\text{bool} & \text{if type}_{field} \text{ is boolean}
\end{cases}\)

- each field name \(id^{A_1}_{field_i}\) for \(i \in \{1, 2, \ldots, n\}\) plays a role as a label in the record type.

Next, the state \(St(A_2)\) of the class \(A_2\) which inherits from the class \(A_1\) and has new \(m\) fields is modeled with the state of its upper-class by means of the structural coercion between \(St(A_2)\) and \(St(A_1)\). Simply put, \(St(A_2) \leq_{c_1} St(A_1)\) where \(c_1\) is the composite function of a series of the first projection coercions \([\cdot]\)
introduced in the section 3.4. This coercion $c_1$ is modeling the feature that all fields of the class $A_1$ are implicitly inherited to the class $A_2$.

$$\mathcal{F}[\text{FieldVarDecl}^{A_2}] =_d \langle \text{St}(A_1), \ id^{A_2}_{\text{field}_1} : \zeta^{[\text{type field}_1]}, \cdots, \ id^{A_2}_{\text{field}_m} : \zeta^{[\text{type field}_m]} \rangle$$

$$\text{St}(A_2) \equiv \mathcal{F}[\text{FieldVarDecl}^{A_2}]$$

Of special important, the labels $id^{A_2}_{\text{field}_1}, id^{A_2}_{\text{field}_2}, \cdots, id^{A_2}_{\text{field}_m}$ must not be used in the record type $\text{St}(A_1)$.

Next, the state $\text{St}(A_3)$ of the class $A_3$ which inherits from the class $A_1$ and has new $p$ fields is modeled similarly with $\text{St}(A_3) \leq_{c_1} \text{St}(A_1)$,

$$\mathcal{F}[\text{FieldVarDecl}^{A_3}] =_d \langle \text{St}(A_1), \ id^{A_3}_{\text{field}_1} : \zeta^{[\text{type field}_1]}, \cdots, \ id^{A_3}_{\text{field}_p} : \zeta^{[\text{type field}_p]} \rangle$$

$$\text{St}(A_3) \equiv \mathcal{F}[\text{FieldVarDecl}^{A_3}]$$

As discussed above, the labels $id^{A_3}_{\text{field}_1}, id^{A_3}_{\text{field}_2}, \cdots, id^{A_3}_{\text{field}_p}$ must be different from the labels in the record type $\text{St}(A_1)$; however, they can be the same as the labels $id^{A_2}_{\text{field}_1}, id^{A_2}_{\text{field}_2}, \cdots, id^{A_2}_{\text{field}_m}$ introduced in the record type $\text{St}(A_2)$. It means that our model can exactly reflect the feature of naming fields in object-oriented programming, that is to say, the field names in the class $A_3$ can be the same as those in the class $A_2$ because the classes $A_2$ and $A_3$ inherit from $A_1$, having been a different branch. Regrettably, Coq restricts to choose label names. In Coq, labels are defined as global functions which are required to be different from others, so that field names of classes to be verified should be different each other even though they are members of different classes.

Next, the state $\text{St}(A_4)$ of the class $A_4$ which inherits from the classes $A_3$ and $A_1$ and has new $q$ fields is defined with $\text{St}(A_4) \leq_{c_1} \text{St}(A_3)$,

$$\mathcal{F}[\text{FieldVarDecl}^{A_4}] =_d \langle \text{St}(A_3), \ id^{A_4}_{\text{field}_1} : \zeta^{[\text{type field}_1]}, \cdots, \ id^{A_4}_{\text{field}_q} : \zeta^{[\text{type field}_q]} \rangle$$

$$\text{St}(A_4) \equiv \mathcal{F}[\text{FieldVarDecl}^{A_4}]$$
Next, the state $St(B)$ of the class $B$ which has $r$ fields is defined

$$F[FieldVarDecl^B] = df \langle id^B_{field_1} : \zeta[[type_{field_1}]], \ldots, id^B_{field_r} : \zeta[[type_{field_r}]] \rangle$$

$$St(B) \equiv F[FieldVarDecl^B]$$

Finally, we are defining the Ω-state consisting of all record types $St(A_1)$, $St(A_2)$, $St(A_3)$, $St(A_4)$ and $St(B)$ defined in the above.

$$\Omega_{\{B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1\}} \equiv \langle l_B : St(B), l_{A_4} : St(A_4), l_{A_3} : St(A_3), l_{A_2} : St(A_2), l_{A_1} : St(A_1) \rangle$$

Of particular value is here to mention the subscript of $\Omega_{\{B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1\}}$. The fact that $A_1$ appears four times in the subscript means that the fields declared in the class $A_1$ are inherited to the classes $A_2$, $A_3$ and $A_4$, so that they play a role as labels in each record type of state $St(A_1)$, $St(A_2)$, $St(A_3)$ and $St(A_4)$.

In the following, we use $\Omega$ instead of $\Omega_{\{B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1\}}$ as a shorthand. Most obviously, in the process of defining the Ω-state, there should not exist the structural coercive subtyping between Ω and the states $St(A_n)$ for $n \in \{1,2,3,4\}$, and between Ω and the state $St(B)$, in order to avoid path confusion; put otherwise, to keep coercion coherent.

Figure 4.2: Examples for UML class diagram

With the above modeling the state of objects, we could model any structure of classes in the context as the Ω-state. For example, two structures shown in...
Figure 4.2 are interpreted as $\Omega^1$ and $\Omega^2$ respectively as follows,

$\Omega^1_{\{(A_1;A_2),(A_1;A_4),(A_1;A_3),(A_1;A_2),A_1\}} \equiv \langle l_{A_5} : St(A_5) , l_{A_4} : St(A_4) , l_{A_3} : St(A_3) ,
\quad l_{A_2} : St(A_2) , l_{A_1} : St(A_1) \rangle$

$\Omega^2_{\{(B_1;B_2),(A_1;A_2),B_1,A_1\}} \equiv \langle l_{B_2} : St(B_2) , l_{A_2} : St(A_2) ,
\quad l_{B_1} : St(B_1) , l_{A_1} : St(A_1) \rangle$

Returning to Figure 4.1, all methods of the classes $A_1, A_2, A_3, A_4$ and $B$ are modeled as taking an object of $\Omega$ as an implicit parameter, but each method is interested only in the state of the class to which it belongs. As a consequence, for a field $id_{field}^A$ which is a member of the class $A_i$ where $i \in \{1, 2, 3, 4\}$ or a field $id_{field}^B$ of the class $B$, if it is used in the methods of the class $A_i$, it is interpreted as following, with the exception of field names appearing in the left-hand side of assignment statements: for $\omega$ being an object of $\Omega$ the method is taking

$E[ id_{field}^A ] =_df \omega.l_{A_i}.id_{field}^A$

$E[ id_{field}^B ] =_df \omega.l_{B_i}.id_{field}^B$

Furthermore, if a method $f$ has the following set of $s$ local variables

$\{ type_{local_1} \ id_{local_1}^f ; \cdots ; type_{local_s} \ id_{local_s}^f \}$

then the $\Omega$-state is temporarily increasing only for this method with the coercive subtyping $\Omega^f \leq \zeta$ where

$\Omega^f_{\{B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1\}} \equiv \langle \Omega(B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1) ,
\quad \zeta[type_{local_1}] , \cdots , \zeta[type_{local_s}] \rangle$

$\zeta[type_{local}] =_df \begin{cases} nat & \text{if } type_{local} \text{ is int} \\ bool & \text{if } type_{local} \text{ is boolean} \end{cases}$

The local variable $id_{local_j}^f$ where $j \in \{1, 2, \ldots, s\}$ in the method $f$ is therefore interpreted as follows: for $\omega$ being an object of $\Omega^f$ the method $f$ is taking

$E[ id_{local_j}^f ] =_df \omega.id_{local_j}^f$
4.2 Modeling a class as a dependent record type

4.2.1 Modeling statements

Let us now draw a distinction between a method including local variables and a method not including them. In our interpretation, each method consisting of a sequence of statements is modeled as taking an object of Ω-state implicitly regardless of having local variables, and returning a new object of Ω-state if the state is changed. However, depending on the existence of local variables, a statement is differently interpreted.

Without local variables

If a method does not include local variables, each statement is modeled as a function taking an object of the Ω-state and returning a new object of the Ω-state as follows except while-loop statements.

- **Assignment:** Assignment statements are modeled as the construction of a new state where the record field associated with the assigned field name is changed. In other words, we construct a function that takes a state as input and returns a new state as output. For example, suppose that the field $id_{field_i}$ of the class $A_2$ occurs in an assignment statement. The assignment statement is then modeled as a function which can access only the state $St(A_2)$ of the class $A_2$.

$$S[id_{field_i} = Expression] = df \lambda \omega: \Omega. \langle \omega.l_B, \omega.l_{A_4}, \omega.l_{A_3}, \langle \cdots.\omega.l_{A_2}.id_{field_{i-1}}^A, \epsilon[Expression], \omega.l_{A_2}.id_{field_{i+1}}^A \cdots \rangle, \omega.l_{A_1} \rangle$$

where

$$\Omega \equiv \langle l_B : St(B), l_{A_4} : St(A_4), l_{A_3} : St(A_3), \langle \cdots id_{field_{i-1}}^A : \zeta[type_{field_{i-1}}], id_{field_i}^A : \zeta[type_{field_i}], id_{field_{i+1}}^A : \zeta[type_{field_{i+1}}] \cdots \rangle, l_{A_1} : St(A_1) \rangle$$
• **Sequence statement:** When executing a sequence of statements, for instance $S_1; S_2$, the statement $S_2$ works in the state returned by $S_1$. If the two statements $S_1$ and $S_2$ are interpreted by $S[S_1]$ and $S[S_2]$ respectively, the function that represents the whole sequence has the following form.

$$S[S_1; S_2] =_{df} \lambda \omega: \Omega. (S[S_2](S[S_1]\omega))$$

• **IfThenElse statement:** If the statements $S_1$ and $S_2$ in the branch are interpreted as $S[S_1]$ and $S[S_2]$ respectively, the whole conditional statement can be represented as the following function.

$$S[if \; Expression \; S_1 \; else \; S_2] =_{df} \lambda \omega: \Omega. if \; B[Expression]$$

$$then \; (S[S_1] \omega) \; else \; (S[S_2] \omega)$$

$$S[if \; Expression \; S_1] =_{df} \lambda \omega: \Omega. if \; B[Expression]$$

$$then \; (S[S_1] \omega) \; else \; \omega$$

where $B:(Expression)bool$ and $B[Expression]$ has no side-effects.

• **Implicit-method-call statement, Explicit-method-call statement, and Field-method-call statement** will be explained later on.

• **While-Loop statement** will be explained later in Section 4.6.

**With local variables**

If a method $f$ has local variables, each statement is modeled as a function taking an object of $\Omega^f$ extended with local variables and returning a new object of $\Omega^f$. Regretfully, we assume that if a method $f$ has local variables, the $f$ can have only Assignment statements or IfThenElse statements.\(^2\)

• **Assignment:** We suppose that the method $f$ of the class $A_2$ has local variables.

$$S[id_{local_j}^{A_2} = Expression] =_{df} \lambda \omega: \Omega^f. \langle \omega.l_0, \cdots \omega.id_{local_{j-1}}^{A_2},$$

$$E[Expression], \omega.id_{local_{j+1}}^{A_2}, \cdots \rangle$$

\(^2\)JaCo can not currently interpret a method including local variables, method-call statements, and while-loop statements altogether.
where

\[ \Omega_f \equiv \langle l_\Omega: \Omega, \ldots, id_{local_{j-1}}: \zeta_{[\text{type}_{local_{j-1}}]},
   id_{local_j}: \zeta_{[\text{type}_{local_j}]}, id_{local_{j+1}}: \zeta_{[\text{type}_{local_{j+1}}]} \rangle \]

- Sequence statement:

\[ S[S_1; S_2] = df \lambda \omega: \Omega_f.(S[S_2](S[S_1]\omega)) \]

- IfThenElse statement:

\[ S[if \ \text{Expression} \ S_1 \ \text{else} \ S_2] = df \lambda \omega: \Omega_f. \ \text{if} \ B[\text{Expression}] \ \text{then} \ (S[S_1]\omega) \ \text{else} \ (S[S_2]\omega) \]

\[ S[if \ \text{Expression} \ S_1] = df \lambda \omega: \Omega_f. \ \text{if} \ B[\text{Expression}] \ \text{then} \ (S[S_1]\omega) \ \text{else} \ \omega \]

### 4.2.2 Modeling methods

Given that a method \(id_{method_i}^A\) of the class \(A\) is modeled as a composite function \(t_f\) of a sequence of statements, the method of type \(T_f\) is represented as an inductive unit type \(Unit(T_f, t_f)\) where \(Unit: (A:Type)(x:A)Type\). We assume that the class \(A\) has \(n\) methods, each method is interpreted by \(U\), for \(i \in \{1, 2, \ldots, n\}\)

\[ U[public \ \text{type}_{\text{return}} \ id_{method_i}^A(type_{\text{para}_1} id_{\text{para}_1}, \ldots, type_{\text{para}_p} id_{\text{para}_p})\{S_1, \ldots, S_m, \ (\text{return} \ id^A)\}] \]

where the method \(id_{method_i}^A\) has \(p\) parameters and \(m\) statements.

If the method \(id_{method_i}^A\) is a mutator, then

\[ t_{fi} \equiv \lambda(id_{\text{para}_1}: \zeta_{[\text{type}_{\text{para}_1}]}) \ldots (id_{\text{para}_p}: \zeta_{[\text{type}_{\text{para}_p}]})(w: \Omega). \]

\[ S[S_m] \cdot S[S_{m-1}] \cdots (S[S_1] \cdot w) \]

\[ T_{fi} \equiv \zeta_{[\text{type}_{\text{para}_1}]} \rightarrow \cdots \rightarrow \zeta_{[\text{type}_{\text{para}_p}]} \rightarrow \Omega \rightarrow \Omega \]
If the method $id_{method_i}^A$ is an accessor, then

$$t_{fi} \equiv \lambda(id_{para_1} : \zeta[\text{type}_{para_1}]) \ldots (id_{para_p} : \zeta[\text{type}_{para_p}]) (w : \Omega).$$

$$(S[S_m] \cdot S[S_{m-1}] \ldots (S[S_1] w)).I_A.id^A$$

$$T_{fi} \equiv \zeta[\text{type}_{para_1}] \rightarrow \cdots \rightarrow \zeta[\text{type}_{para_p}] \rightarrow \Omega \rightarrow \zeta[\text{type}_{return}]$$

If the method $id_{method_i}^A$ is both a mutator and an accessor, then

$$t_{fi} \equiv \lambda(id_{para_1} : \zeta[\text{type}_{para_1}]) \ldots (id_{para_p} : \zeta[\text{type}_{para_p}]) (w : \Omega).$$

\[
\text{let temp} := S[S_m] \cdot S[S_{m-1}] \ldots (S[S_1] w) \\
\text{in } (\text{temp , temp}.l_A.id^A)
\]

$$T_{fi} \equiv \zeta[\text{type}_{para_1}] \rightarrow \cdots \rightarrow \zeta[\text{type}_{para_p}] \rightarrow \Omega \rightarrow (\Omega * \zeta[\text{type}_{return}])$$

In the above, we have modeled parameters and a return value with the following interpretation,

$$\zeta[\text{type}_{para}] =_{df} \begin{cases} 
\text{nat} & \text{if } \text{type}_{para} \text{ is int} \\
\text{bool} & \text{if } \text{type}_{para} \text{ is boolean}
\end{cases}$$

$$\zeta[\text{type}_{return}] =_{df} \begin{cases} 
\text{nat} & \text{if } \text{type}_{return} \text{ is int} \\
\text{bool} & \text{if } \text{type}_{return} \text{ is boolean} \\
\text{empty} & \text{if } \text{type}_{return} \text{ is void}
\end{cases}$$

Then, the set of all $n$ methods of the class $A$ is interpreted by $\mathcal{M}$

$$\mathcal{M}[\text{MethodDecl}^A] =_{df} \langle f_1 : U[id_{method_1}^A] \ldots , f_n : U[id_{method_n}^A] \rangle$$

$$\equiv \langle f_1 : \text{Unit}(T_{f_1}, t_{f_1}) \ldots , f_n : \text{Unit}(T_{f_n}, t_{f_n}) \rangle \cdots (1)$$

$$\equiv \langle f_1 \sim t_{f_1} : T_{f_1} \ldots , f_n \sim t_{f_n} : T_{f_n} \rangle \cdots (2)$$

where $\text{MethodDecl}^A$ denotes the set of all method entries declared in the class $A$, and we are using $U[id_{method}^A]$ for denoting interpretation of the method $id_{method_i}^A$ as a shorthand. Regarding (1), it is possible to define the non-structural coercion ($\xi$) which maps any object of $\text{Unit}(T,t)$ to $t$ of type $T$ where $t$ stands for a method.
\[
\Gamma \vdash T : Type \quad \Gamma \vdash t : T
\]
\[
\Gamma \vdash Unit(T,t) \leq \xi(T,t) T : Type
\]

**Remark (Intensional manifest field)** [Luo09] With regard to (2), in the rest of the paper, for easy reading, we shall adopt the following notation

\[
x \sim t : T \text{ for } x : Unit(T,t)
\]

This does not mean that we have introduced manifest fields [Ler94] [Pol02] [CPT05]. It is only a notation for a unit type component in a record type.

**Modeling implicit-method-call statement and expression**

Given that in the class \(A\), the method \(id^A_{method_i}\) is interpreted as follows,

\[
U[id^A_{method_i}] = df Unit(T_f, t_f)
\]

the method \(id^A_{method_k}\) which has a implicit-method-call statement referring to \(id^A_{method_i}\) where \(i \neq k\) is interpreted

\[
U[id^A_{method_k}] = df Unit(T_{f_k}, t_{f_k})
\]

\[
t_{f_k} = df \lambda (g : T_f)(id^A_{para_1} : type_{para_1}) \cdots (id^A_{para_p} : type_{para_p})(w : \Omega).
\]

\[
S[S_m] \cdot S[S_{m-1}] \cdots (S[S_1] w)
\]

where the method \(id^A_{method_k}\) is assumed as a mutator. Then the implicit-method-call statement is interpreted as follows

\[
S[this.id^A_{method_i}((Expression)^*)] = df \lambda \omega:\Omega. (g \ E[(Expression)^*])
\]

In the above, we can see that the function \(t_{f_k}\) is parameterized by the function \(g\) of type \(T_f\). This parametrization is significantly modeled in the dependent record type

\[
\mathcal{M}[MethodDecl^A] = df \langle \cdots, f_i : U[id^A_{method_i}], f_k : U[id^A_{method_k}], \cdots \rangle
\]

\[
\equiv \langle \cdots, f_i : Unit(T_{f_i}, t_{f_i}), f_k : Unit(T_{f_k}, (t_{f_k} f_i)), \cdots \rangle
\]

\[
\equiv \langle \cdots, f_i \sim t_{f_i} : T_{f_i}, f_k \sim (t_{f_k} f_i) : T_{f_k}, \cdots \rangle
\]
We should note that $f_i$ is not a function since it is of type $\text{Unit}(T_{f_i}, t_{f_i})$. How could the application $(t_{f_k} f_i)$ be well-typed? The reason that $(t_{f_k} f_i)$ is well-typed is that $f_i$ being an object of type $\text{Unit}(T_{f_i}, t_{f_i}, f_i)$ is coerced into $\xi(T_{f_i}, t_{f_i}, f_i) = t_{f_i}$ of type $T_{f_i}$.

As well, an implicit-method-call expression appearing in the statement $S_i$ for $i \in \{1, 2, ..., m\}$ can be modeled similarly in a parameterized function and a dependent record type. However, since it is not a statement, it is interpreted with $\tau$. For $\omega$ being an object of $\Omega$ the statement $S_i$, to which the implicit-method-call expression belongs, is taking

$$\tau[\text{this.id}_{method_i}((Expression)^*)] =_d (g \ E[(Expression)^*] \omega)$$

### 4.2.3 Modeling classes

Finally, we are doing a functional interpretation of the class $A$ with $C$ where the first projection coercion $[\_]$ is defined:

$$C[ClassDecl^A] =_d \langle \ M[MethodDecl^A], \ l : F[VarDecl^A] \rangle$$

$$C[ClassDecl^A] \leq_d M[MethodDecl^A]$$

where $ClassDecl^A$ denotes the set of the method and field entries declared in the class $A$.

### 4.3 Inheritance in coercive subtyping

Given that the classes $A_3$ and $A_4$ are interpreted as we did in the previous section, we are now modeling their inheritance relationship by way of interpreting their interface-types $IA_3$ and $IA_3$ with $I$.

$$M[MethodDecl^{IA_3}] =_d \langle f_1 \sim t_{f_1}, f_2 \sim t_{f_2}, \ldots, f_n \sim t_{f_n} : T_{f_n} \rangle$$

$$I[InterfaceDecl^{IA_3}] =_d \langle f_1 : T_{f_1}, f_2 : T_{f_2}, \ldots, f_n : T_{f_n} \rangle$$

$$M[MethodDecl^{IA_3}] \leq_d I[InterfaceDecl^{IA_3}]$$
where InterfaceDecl$^{IA_3}$ denotes the set of the method signatures declared in the interface-type $IA_3$, and $d$ is the component-wise coercion function introduced in the section 3.4. As a result, we can see that the relation between a class and its interface-type is interpreted as subtyping relation:

$$\mathcal{C}[\text{ClassDecl}^{A_3}] \leq_{(\_)} \mathcal{M}[\text{MethodDecl}^{A_3}] \leq_d \mathcal{I}[\text{InterfaceDecl}^{IA_3}]$$

$$\mathcal{C}[\text{ClassDecl}^{A_3}] \leq_{d\circ(\_)} \mathcal{I}[\text{InterfaceDecl}^{IA_3}]$$

As is well known, Coq supports a limited form of coercions.\(^3\) Coq does not support user-defined coercion rules like the component-wise coercion $d$, so that we have to specify its effects on the record types manually as we did in Chapter 2.

Next we are modeling the class $A_4$ inheriting from the class $A_3$, and its interface-type $IA_4$ inheriting from the interface-type $IA_3$. All methods of the class $A_3$ are implicitly included in the class $A_4$. We assume that new methods defined in the class $A_4$ are interpreted as the function $t_{g_j}$ for $j \in \{1, \cdots, m\}$ respectively.

---

\(^3\)In Plastic [pla], we only have to declare parameterized coercions like $\xi$ and structural coercion rules such as the projections and the component-wise coercion. Then the Plastic system obtains automatically all of the derivable coercions, as intended. However, Plastic does not support record types.
\[ \mathcal{M}[\text{MethodDecl}^{IA_4}] =_{df} (f_1 \sim t_{f_1}, f_2 \sim t_{f_2}, \ldots, f_n \sim t_{f_n}, g_1 \sim t_{g_1}, g_2 \sim t_{g_2}, \ldots, g_m \sim t_{g_m}) \]

\[ \mathcal{I}[\text{InterfaceDeclExt}^{IA_4}] =_{df} (f_1 : T_{f_1}, f_2 : T_{f_2}, \ldots, f_n : T_{f_n}, g_{n+1} : T_{g_1}, g_{n+2} : T_{g_2}, \ldots, g_{n+m} : T_{g_m}) \]

\[ \mathcal{C}[\text{ClassDeclExt}^{IA_4}] =_{df} (\mathcal{M}[\text{MethodDecl}^{IA_4}], l : \mathcal{F}[\text{VarDecl}^{IA_4}]) \]

\[ \mathcal{C}[\text{ClassDeclExt}^{IA_4}] \leq_{[]} \mathcal{M}[\text{MethodDecl}^{IA_4}] \leq_{d} \mathcal{I}[\text{InterfaceDeclExt}^{IA_4}] \]

\[ \mathcal{C}[\text{ClassDeclExt}^{IA_4}] \leq_{d0[\cdot]} \mathcal{I}[\text{InterfaceDeclExt}^{IA_4}] \]

Subsequently we can interpret the inheritance relationship between \( IA_4 \) and \( IA_3 \) as subtyping relation with the coercion \( c_2 \) which is a composite function of a series of \( d \) and \([\cdot]\)

\[ \mathcal{I}[\text{InterfaceDeclExt}^{IA_4}] \leq_{c_2} \mathcal{I}[\text{InterfaceDecl}^{IA_3}] \]

Proceeding from what we have seen above, indirect inheritance relationship shown in Figure 4.3 is eventually modeled as the coercion graph shown in Figure 4.4.

\[ \mathcal{C}[\text{ClassDecl}^{IA_3}] \leq_{[]} \mathcal{M}[\text{MethodDecl}^{IA_3}] \leq_{d} \mathcal{I}[\text{InterfaceDecl}^{IA_3}] \]

\[ \mathcal{C}[\text{ClassDeclExt}^{IA_4}] \leq_{[]} \mathcal{M}[\text{MethodDecl}^{IA_4}] \leq_{d} \mathcal{I}[\text{InterfaceDeclExt}^{IA_4}] \]

Figure 4.4: Coercion graph
4.4 Subtype polymorphism and dynamic dispatch with interface-types

Modeling explicit-method-call statement and expression

We make two assumptions here: one is that the classes $A_3$ and $A_4$, and their interface-types $IA_3$ and $IA_4$ are interpreted as shown in Figure 4.4, and the other is that the method $id_{method_3}^{A_3}$ of the class $A_3$ is overridden as $id_{method_4}^{A_4}$ with the same signature in the class $A_4$, so that the method $id_{method_3}^{A_3}$ and $id_{method_4}^{A_4}$ are interpreted as $t_f$ and $t'_f$ respectively, but with the same label $f_i$:

\[
M[#MethodDecl^{A_3}] = def \langle \cdots f_i : U[id_{method_3}^{A_3}] \cdots \rangle \\
\equiv \langle \cdots f_i \sim t_f : T_f \cdots \rangle \\
I[#InterfaceDecl^{IA_3}] = def \langle \cdots f_i : T_f \cdots \rangle \\
M[#MethodDecl^{A_4}] = def \langle \cdots f_i : U[id_{method_4}^{A_4}] \cdots \rangle \\
\equiv \langle \cdots f_i \sim t'_f : T'_f \cdots \rangle \\
I[#InterfaceDeclExt^{IA_4}] = def \langle \cdots f_i : T'_f \cdots \rangle \\
\]

We shall thereafter model a method $id^{B}_{method_k}$ of the class $B$ which is taking an explicit parameter of type $IA_3$ to exchange messages with the classes $A_3$ or $A_4$. Accordingly, the method $id^{B}_{method_k}$ may have an explicit-method-call statement referring to $id^{A_3}_{method_k}$:

\[
U[#id^{B}_{method_k}] = def \ Unit(T_{f_k}, t_{f_k}) \\
t_{f_k} = def \ \lambda(id^{B}_{para_1} : \zeta[IA_3]) (id^{B}_{para_2} : \zeta[type_{para_2}]) \cdots (id^{B}_{para_p} : \zeta[type_{para_p}]) (\omega:\Omega). S[S_m] : S[S_{m-1}] \cdots (S[S_1] w) \\
\]

where we assume that $id^{B}_{method_k}$ is a mutator, and define the extended interpretation for parameters

\[
\zeta[type_{para}] = def \begin{cases} 
  nat & \text{if type}_{para} \text{ is int} \\
  bool & \text{if type}_{para} \text{ is boolean} \\
  I[#InterfaceDecl^{IA_i}] & \text{if type}_{para} \text{ is an interface-type } IA_i \text{ for } i \in \{3, 4\}
\end{cases}
\]
Then, the explicit-method-call statement is interpreted as a field selection \( (\text{id}^B_{\text{para}1}.f_i) \),

\[
\mathcal{S}[\text{id}^B_{\text{para}1}.\text{id}^A_3_{\text{method}_i}((\text{Expression})^*)] =_{df} \lambda \omega: \Omega. (\text{id}^B_{\text{para}1}.l_i) \mathcal{E}[(\text{Expression})^*] \omega
\]

Simultaneously, we should have another interpretation for this explicit-method-call statement with respect to invocation of the method \( \text{id}^B_{\text{method}_i} \). As observed earlier in the explanation of dynamic dispatch in Chapter 2, if it is applied to an object of the class \( A_3 \) implementing \( IA_3 \), then the method \( \text{id}^A_3_{\text{method}_i} \) of the class \( A_3 \) would be executed; likewise, if it is applied to an object of the class \( A_4 \) implementing \( IA_4 \), then the method \( \text{id}^A_4_{\text{method}_i} \) of the class \( A_4 \) would be executed.

Indisputably, we can see the notion of dynamic dispatch work correctly in the frame of coercive subtyping because the methods \( \text{id}^A_3_{\text{method}_i} \) and \( \text{id}^A_4_{\text{method}_i} \) are interpreted with the same label \( f_i \).

As well, an explicit-method-call expression appearing in the statement \( S_i \) for \( i \in \{1, 2, ..., m\} \) can be modeled similarly by using a dependent record type and field-selection. For \( \omega \) being an object of \( \Omega \) the statement \( S_i \), to which the explicit-method-call expression belongs, is taking

\[
\tau[\text{id}^B_{\text{para}1}.\text{id}^A_3_{\text{method}_i}((\text{Expression})^*)] =_{df} (\text{id}^B_{\text{para}1}.l_i) \mathcal{E}[(\text{Expression})^*] \omega
\]

\[
\Psi[\text{id}^B_{\text{para}1}.\text{id}^A_3_{\text{method}_i}((\text{Expression})^*)] =_{df}
\]

### 4.5 Modeling fields of interface-type

We extend the class diagram shown in Figure 4.1 with the new class \( C \) which has fields of the interface-types \( IA_3 \) or \( IA_4 \), where we assume that in the context there exist only two interface-types \( IA_3 \) or \( IA_4 \).

\[
\mathcal{F}[\text{FieldVarDecl}^C] =_{df} \langle \text{id}^C_{\text{field}_1 : \text{type field}_1}, \ldots, \text{id}^C_{\text{field}_n : \text{type field}_n} \rangle
\]

\[
\text{St}(C) \equiv \mathcal{F}[\text{FieldVarDecl}^C]
\]
where the interpretation $\zeta$ is extended for fields of interface-type,

$$
\zeta[type_{field}] =_{df} \begin{cases} 
\text{nat} & \text{if } type_{field} \text{ is int} \\
\text{bool} & \text{if } type_{field} \text{ is boolean} \\
St(A_i) & \text{if } type_{field} \text{ is an interface-type } IA_i \text{ for } i \in \{3, 4\}
\end{cases}
$$

One point to note is that to declare fields of the interface-types $IA_3$ or $IA_4$ in the class $C$, these interface-types must be defined earlier in the context before declaring the class $C$, because our functional interpretation is made in sequence from top to bottom.\(^4\)

We are then getting to the new $\Omega$-state including the new state $St(C)$ of the class $C$.

$$
\Omega_{\{C,B,(A_1;A_3;A_4),(A_1;A_3),(A_1;A_2),A_1\}} \equiv \langle \ l_C : St(C) , \ l_B : St(B) , \ l_{A_4} : St(A_4) , \\
\ l_{A_3} : St(A_3) , \ l_{A_2} : St(A_2) , \ l_{A_1} : St(A_1) \rangle
$$

This $\Omega$-state should be defined before starting to interpret methods of the classes $A_1$, $A_2$, $A_3$, $A_4$, $B$ and $C$.

\(^4\)This is a known deficit in myJava programs. In Java programming, we can however declare fields of interface-type anywhere if this type is just declared in the context.
CHAPTER 4. SEMANTICS OF FUNCTIONAL INTERPRETATION

Modeling an assignment statement with generating a new object

We assume that the field \( \text{id}^C_{\text{field}_i} \) in the class \( C \) is declared of the interface-type \( IA_3 \), so that it is interpreted as an object of the state \( St(A_3) \). Accordingly, an instance of the class \( A_3 \) implementing \( IA_3 \) can be assigned to the field \( \text{id}^C_{\text{field}_i} \), or an instance of the class \( A_4 \) implementing \( IA_4 \) which inherits from \( IA_3 \) can be assigned to the field \( \text{id}^C_{\text{field}_i} \). The following assignment statement with generating a new object is interpreted with initializing fields.

\[
S[\text{id}^C_{\text{field}_i} = \text{new } A_3()] =_df \ \lambda \omega : \Omega. \begin{align*}
\langle & \cdots \omega.l_C.\text{id}^C_{\text{field}_{i-1}}, \langle \cdots 0, \text{false} \cdots \rangle, \omega.l_C.\text{id}^C_{\text{field}_{i+1}} \rangle, \\
& \omega.l_B, \omega.l_{A_4}, \omega.l_{A_3}, \omega.l_{A_2}, \omega.l_{A_1} \rangle
\end{align*}
\]

where

\[
\Omega \equiv \langle \langle \cdots \text{id}^C_{\text{field}_{i-1}} : \zeta[\text{type}\text{field}_{i-1}], \text{id}^C_{\text{field}_i} : St(A_3), \text{id}^C_{\text{field}_{i+1}} : \zeta[\text{type}\text{field}_{i+1}] \cdots \rangle, \\
l_B : St(B), l_{A_4} : St(A_4), l_{A_3} : St(A_3), \\
l_{A_2} : St(A_2), l_{A_1} : St(A_1) \rangle
\]

\[
St(A_3) \equiv \langle \langle \cdots \text{id}^{A_3}_{\text{field}_j} : \text{nat}, \text{id}^{A_4}_{\text{field}_{j+1}} : \text{bool} \cdots \rangle \rangle
\]

If an instance of the class \( A_4 \) is assigned to the field \( \text{id}^C_{\text{field}_i} \), then the field is interpreted as an object of the state \( St(A_4) \).

\[
S[\text{id}^C_{\text{field}_i} = \text{new } A_4()] =_df \ \lambda \omega : \Omega. \begin{align*}
\langle & \cdots \omega.l_C.\text{id}^C_{\text{field}_{i-1}}, \langle \cdots 0, \text{false} \cdots, 0 \cdots \rangle, \\
& \omega.l_C.\text{id}^C_{\text{field}_{i+1}} \cdots \rangle, \omega.l_B, \omega.l_{A_4}, \omega.l_{A_3}, \omega.l_{A_2}, \omega.l_{A_1} \rangle
\end{align*}
\]

where

\[
\Omega \equiv \langle \langle \cdots \text{id}^C_{\text{field}_{i-1}} : \zeta[\text{type}\text{field}_{i-1}], \text{id}^C_{\text{field}_i} : St(A_4), \text{id}^C_{\text{field}_{i+1}} : \zeta[\text{type}\text{field}_{i+1}] \cdots \rangle, \\
l_B : St(B), l_{A_4} : St(A_4), l_{A_3} : St(A_3), \\
l_{A_2} : St(A_2), l_{A_1} : St(A_1) \rangle
\]

\[
St(A_4) \equiv \langle \langle St(A_3), \text{id}^{A_3}_{\text{field}_k} : \text{nat} \cdots \rangle \rangle
\]

\[
\equiv \langle \langle \cdots \text{id}^{A_3}_{\text{field}_j} : \text{nat}, \text{id}^{A_4}_{\text{field}_{j+1}} : \text{bool} \cdots \rangle, \text{id}^{A_4}_{\text{field}_k} : \text{nat} \cdots \rangle
\]
CHAPTER 4. SEMANTICS OF FUNCTIONAL INTERPRETATION

Modeling field-method-call statement and expression

As observed earlier in chapter 2, we need to first define the new \( \Omega \)-state with the interface-types \( IA_3 \) or \( IA_4 \), in order to interpret field-method-call statements and expressions as type-theoretic models with dependent record types and coercive subtyping.

The \( \Omega \)-state is temporarily being increased only for interpreting field-method-call statements and expressions

\[
\Omega^{IA_3}_{\{C,B,(A_1:A_3),(A_1:A_3),(A_1:A_2),A_1\}} \equiv \text{df} \langle \Omega\{C,B,(A_1:A_3),(A_1:A_3),(A_1:A_2),A_1\} ; l_{IA_3} : \mathcal{I}[\text{InterfaceDecl}^{IA_3}] \rangle
\]

\[
\Omega^{IA_4}_{\{C,B,(A_1:A_3),(A_1:A_3),(A_1:A_2),A_1\}} \equiv \text{df} \langle \Omega\{C,B,(A_1:A_3),(A_1:A_3),(A_1:A_2),A_1\} ; l_{IA_4} : \mathcal{I}[\text{InterfaceDecl}^{IA_4}] \rangle
\]

where the following coercive subtyping relations are defined

\[
\Omega^{IA_3} \leq_{\lceil \cdot \rceil} \Omega \quad \Omega^{IA_4} \leq_{\lceil \cdot \rceil} \Omega \quad \Omega^{IA_4} \leq_d \Omega^{IA_3}
\]

Let’s model the method \( id^C_{\text{method}} \) in the class \( C \) which includes the field-method-call statement \( id^C_{\text{field}} \cdot id^{A_3}_{\text{method}}((\text{Expression})^*) \), given that an instance of the class \( A_3 \) is assigned to the field \( id^C_{\text{field}} \) of the interface-type \( IA_3 \). We assume that the method \( id^{A_3}_{\text{method}} \) in the class \( A_3 \) is overridden as \( id^{A_4}_{\text{method}} \) in the class \( A_4 \).

\[
\mathcal{U}[\text{id}^C_{\text{method}}] = \text{df} \quad \text{Unit}(T_f, t_f)
\]

\[
t_f = \text{df} \lambda(id^C_{\text{para}_1} : \zeta[\text{type}_{\text{para}_1}]) (id^C_{\text{para}_2} : \zeta[\text{type}_{\text{para}_2}]) \cdots (id^C_{\text{para}_p} : \zeta[\text{type}_{\text{para}_p}])\\(\omega: \Omega). \mathcal{S}[S_m] \cdot \mathcal{S}[S_{m-1}] \cdots (\mathcal{S}[S_1] \omega)
\]

where we assume that the field-method-call statement is \( S_j \) for \( j \in \{1, 2, \ldots, m\} \), and that the methods of the class \( A_3 \) are interpreted again to access only the state \( St(A_3) \) declared inside \( St(C) \).
CHAPTER 4. SEMANTICS OF FUNCTIONAL INTERPRETATION

\[ U[id_{method}^{A_3}] =_{df} Unit(T_{f_1}, t_{f_1}^C) \]
\[ U[id_{method}^{A_3}] =_{df} Unit(T_{f_2}, t_{f_2}^C) \]
\[ \vdots \]
\[ U[id_{method}^{A_3}] =_{df} Unit(T_{f_n}, t_{f_n}^C) \]

Then, the field-method-call statement is interpreted as a function taking an object of \( \Omega_{IA_3} \) and returning an object of \( \Omega \),

\[ S[id_{field:id_{method}^{A_3}}((Expression)^*)] =_{df} \]

\[ \begin{align*}
\text{let } \omega_{before} & : = S[S_{j-1}] \cdots (S[S_1] \omega) \\
\text{in let } \omega^{IA_3} & : = \langle \omega_{before}, t_{f_1}^C, t_{f_2}^C, \ldots, t_{f_n}^C \rangle \\
\text{in let } \text{CallStatement} & : = \lambda(\omega; \Omega^{IA_3}), (\omega.l_{IA_3}, f_i) E[(Expression)^*]) \omega \\
\text{in let } \omega_{after} & : = \text{CallStatement} \omega^{IA_3} \\
\text{in } \omega_{after} & \end{align*} \]

Likewise, if an instance of the class \( A_4 \) is assigned to the field \( id_{field} \), the methods of the class \( A_4 \) are interpreted again to access the state \( St(A_4) \) declared inside \( St(C) \),

\[ U[id_{method}^{A_4}] =_{df} Unit(T_{f_1}, t_{f_1}^C) \]
\[ U[id_{method}^{A_4}] =_{df} Unit(T_{f_2}, t_{f_2}^C) \]
\[ \vdots \]
\[ U[id_{method}^{A_4}] =_{df} Unit(T_{f_m}, t_{f_m}^C) \]

The field-method-call statement is then interpreted as a function taking an object of \( \Omega^{IA_4} \) and returning an object of \( \Omega \),

\[ S[id_{field:id_{method}^{A_4}}((Expression)^*)] =_{df} \]
let $\omega_{\text{before}} := S[S_{j-1}] \cdots (S[S_i] \omega)$
in let $\omega^{IA_3} := \langle \omega_{\text{before}}, t_{C_1}^{IA_3}, t_{C_2}^{IA_3}, \cdots, t_{C_m}^{IA_3} \rangle$
in let $\text{CallStatement} := \lambda(\omega^{IA_3}). (\omega.l^{IA_3}.f_i) \mathcal{E}[((\text{Expression})^*)] \omega$
in $\omega^{after} := \text{CallStatement} \omega^{IA_3}$

From the above interpretation, we should note that the record field selection $(\omega.l^{IA_3}.f_i)$ and $(\omega.l^{IA_4}.f_i)$ have the same label $f_i$, which refers to the methods $id^{IA_3}_{\text{method}}$ or $id^{IA_4}_{\text{method}}$, according to the subtyping relation shown in Figure 4.4.

Moreover, the field-method-call expression $id^{IA_3}_{\text{field}}.id^{IA_3}_{\text{method}}$ appearing in the statement $S_j$ for $j \in \{1, 2, ..., m\}$ can be modeled similarly by using $\Omega^{IA_3}$ or $\Omega^{IA_4}$, and field-selection. Here, the statement $S_j$ is modeled as a function taking an object of $\Omega^{IA_3}$ or $\Omega^{IA_4}$ and returning a new object of $\Omega$.

If an instance of the class $A_3$ is assigned to the field $id^{IA_3}_{\text{field}}$ of the interface-type $IA_3$, then for $\omega$ being an object of $\Omega^{IA_3}$ the statement $S_j$ is taking,

$$\tau[id^{IA_3}_{\text{field}}.id^{IA_3}_{\text{method}}((\text{Expression})^*)] =_{df} (\omega.l^{IA_3}.f_i) \mathcal{E}[((\text{Expression})^*)] \omega$$

If an instance of the class $A_4$ is assigned to the field $id^{IA_3}_{\text{field}}$ of the interface-type $IA_3$, then for $\omega$ being an object of $\Omega^{IA_4}$ the statement $S_j$ is taking,

$$\tau[id^{IA_4}_{\text{field}}.id^{IA_4}_{\text{method}}((\text{Expression})^*)] =_{df} (\omega.l^{IA_4}.f_i) \mathcal{E}[((\text{Expression})^*)] \omega$$

**Modeling a return-value of interface-type**

If the class $C$ has a method returning an object of the interface-type $IA_3$ or $IA_4$, we are modeling the return-type of the method with the following interpretation,

$$\zeta[type_{\text{return}}] =_{df} \begin{cases} 
\text{nat} & \text{if } type_{\text{return}} \text{ is int} \\
\text{bool} & \text{if } type_{\text{return}} \text{ is boolean} \\
\text{empty} & \text{if } type_{\text{return}} \text{ is void} \\
St(A_i) & \text{if } type_{\text{return}} \text{ is an interface-type } IA_i \text{ for } i \in \{3, 4\}
\end{cases}$$

**Modeling field-method-call statements with synchronization**

We extend again the class diagram shown in Figure 4.5 with the new class $D$ which has a field of the interface-types $IA_3$ or $IA_4$. 
The new Ω-state is then defined with the record type of state $St(D)$ corresponding to the class $D$.

$$\Omega_{\{D,C,B,(A_1:A_3:A_4),(A_1:A_2),(A_1),A_1\}} \equiv \langle l_D : St(D) , l_C : St(C) , l_B : St(B) , l_{A_4} : St(A_4) , l_{A_3} : St(A_3) , l_{A_2} : St(A_2) , l_{A_1} : St(A_1) \rangle$$

We assume that the field $id^D_{field}$ is declared of the interface-type $IA_3$, and a value on the field $id^C_{field}$ declared of $IA_3$ in the class $C$ is assigned to $id^D_{field}$. Now, let’s model the method $id^D_{method}$ in the class $D$ which includes the field-method-call statement $id^D_{field},id^A_{method},((Expression)^*)$.

$$U[id^D_{method}] = df \ Unit(T_f,t_f)$$

$$t_f = df \lambda(id^D_{para_1} : \zeta[[type_{para_1}]])(id^D_{para_2} : \zeta[[type_{para_2}]]) \cdots (id^D_{para_p} : \zeta[[type_{para_p}]])$$

$$(\omega: \Omega). S[S_m] \cdot S[S_{m-1}] \cdots (S[S_1] \omega)$$

We assume more that the field-method-call statement is $S_j$ for $j \in \{1,2,\ldots,m\}$, and that the methods of the class $A_3$ are interpreted again to access only the
state $St(A_3)$ declared inside the state $St(D)$.

$$U[id_{method}^{A_3}] = df \ Unit(T_{f_1}, t_{f_1})$$
$$U[id_{method}^{A_3}] = df \ Unit(T_{f_2}, t_{f_2})$$
$$\vdots$$
$$U[id_{method}^{A_3}] = df \ Unit(T_{f_n}, t_{f_n})$$

Then, the field-method-call statement is interpreted as a function taking an object of $\Omega^{I_{A_3}}$ and returning an object of $\Omega$,

$$S[[id_{field}^{A_3}, id_{method}^{A_3}, ((Expression)^*)]] = df$$

let $\omega_{before} := S[S_{j-1}] \cdots (S[S_1] \omega)$
in let $\omega^{I_{A_3}} := \langle \omega_{before}, t_{f_1}, t_{f_2}, \ldots, t_{f_n} \rangle$
in let $CallStatement := \lambda(\omega; \Omega^{I_{A_3}}). (\omega.l_{A_3}; f_1) E[((Expression)^*)] \omega$
in let $\omega_{after} := CallStatement \omega^{I_{A_3}}$
in let $\omega^{synch} := \langle \omega_{after}.l_D, \ldots, \omega_{after}.l_{I_{A_3}}, \omega_{after}.l_{i+1}, \omega_{after}.l_{A_1} \rangle$
in $\omega^{synch}$

where the record field in the $St(C)$ corresponding to $id_{field}^{C}$ declared in the class $C$ is replaced by $\omega_{after}.l_D.id_{field}^{D}$ for synchronization.

### 4.6 Modeling a while-loop statement

We assume that the method $id_{method}^{A_1}$ of the class $A_1$ has a terminating while-loop statement where there exists only one field $id_{field}^{A_1}$, whose state is changed, and the state of other fields such as $id_{field}^{A_1}$ and $id_{field}^{A_{i+2}}$ is not changed.

$$U[id_{method}^{A_1}] = df \ Unit(T_f, t_f)$$
$$t_f = df \ \lambda(id_{para_1}^{A_1} : \xi[[type_{para_1}]])(id_{para_2}^{A_1} : \xi[[type_{para_2}]]) \cdots (id_{para_p}^{A_1} : \xi[[type_{para_p}]])$$
$$(\omega; \Omega). S[[S_m]] \cdots (S[S_1] \omega)$$

We assume more that the while-loop statement is $S_j$ for $j \in \{1, 2, \ldots, m\}$,

$$S[[while \ Expression \ (Statement)^*]] = df$$
CHAPTER 4. SEMANTICS OF FUNCTIONAL INTERPRETATION

let $\omega_{\text{before}} := S[S_{j-1}] \cdots (S[S_1] \omega)$
in let $\text{var}_{i_{\text{before}}} := (\omega_{\text{before}}).l_{A_1}.id_{field_i}$
in let $\text{var}_{i+1} := (\omega_{\text{before}}).l_{A_1}.id_{field_{i+1}}$
in let $\text{var}_{i+2} := (\omega_{\text{before}}).l_{A_1}.id_{field_{i+2}}$
in let $\text{WhileStatement} := \lambda(\text{arg}_i : \xi[[\text{type}_{field_i}]])(\text{arg}_{i+1} : \xi[[\text{type}_{field_{i+1}}]])$
$\quad (\text{arg}_{i+2} : \xi[[\text{type}_{field_{i+2}}]]). \text{fun}^*_{\text{rec}} \text{arg}_i \text{arg}_{i+1} \text{arg}_{i+2}$
in let $\text{var}_{i_{\text{after}}} := \text{WhileStatement} \text{var}_{i_{\text{before}}} \text{var}_{i+1} \text{var}_{i+2}$
in let $\omega_{\text{after}} := (\omega_{\text{before}}.l_D, \omega_{\text{before}}.l_C, \omega_{\text{before}}.l_B)$
$\quad (\omega_{\text{before}}.l_{A_4}, \omega_{\text{before}}.l_{A_3}, \omega_{\text{before}}.l_{A_2}, \langle \ldots \text{var}_{i_{\text{after}}} \ldots \rangle)$
in $\omega_{\text{after}}$

where $\text{fun}^*_{\text{rec}}$ is a user-defined primitive recursive function which has the same semantical result as the while-loop statement, and the record field in the state $\text{St}(A_1)$ corresponding to $id_{field_i}$ is replaced by the new state $\text{var}_{i_{\text{after}}}$.

Remark One point is worth noting that we could not define $\text{fun}^*_{\text{rec}}$ for all terminating while-loop statements. While this thesis does provide one possible way of modeling a while-loop statement with primitive recursion, it does not attempt here to outline general models for terminating or non-terminating while-loop statements with regard to general recursion. It is because such consideration goes beyond the bounds of the present thesis.
Chapter 5

Case study

In this chapter it is our intention, on the basis of the semantics defined in Chapter 4, to demonstrate how the myJava programs shown in Figure 5.1 can be succinctly constructed as type-theoretic models. Coq codes generated by JaCo corresponding to these classes are given in Appendix A. In the following there will be a brief explanation for each class and its interface-type.

```java
class Rect_A  //Rectangle
{
    // top-left coordinate (x1,y1)
    protected int x1;
    protected int y1;

    // bottom-right coordinate (x2,y2)
    protected int x2;
}```

Figure 5.1: UML class diagram
The class Rect_A models a rectangle by using only two coordinates, the top-left and the bottom-right. The two coordinates are enough to model a rectangle which has four vertices. The method Rect_A.CONST initializes the fields of coordinates like a constructor.
The class \textit{Rect\_B} inherits from the class \textit{Rect\_A}, so that it includes implicitly all fields and methods declared in the class \textit{Rect\_A}. Moreover, there are new methods in the class \textit{Rect\_B}: \textit{FindWidth}, \textit{FindHeight} and \textit{FindArea} which are finding the width, height and area of a rectangle, respectively. Here, we are mainly concerned with how local variables declared in these methods are modeled in the \(\Omega\)-state, and then how each statement appearing in these methods are interpreted with the \(\Omega\)-state extended with local variables.

```java
interface Interface\_Rect\_C
{
    public void Rect\_C\_CONST();
    public int GetArea();
    public void FindWidth();
    public void FindHeight();
    public void FindArea();
    public void Enlarge(int n);
}

class Rect\_C extends Rect\_A implements Interface\_Rect\_C
{
    protected int width;
    protected int height;
    protected int area;

    public void Rect\_C\_CONST()
    {
        width = 0;
        height = 0;
        area = 0;
    }

    public int GetArea()
    {
        return area;
    }

    public void FindWidth()
    /* PRECONDITION [[ !(x1 == x2) ]] */
    /* POSTCONDITION [[ 0 < width ]] */
    {
        if (x1 < x2)
            width = x2 - x1;
        else
            width = x1 - x2;
    }
```
public void FindHeight()
/* PRECONDITION [[ !(y1 == y2) ]] */
/* POSTCONDITION [[ 0 < height ]] */
{
    if (y1 < y2)
        height = y2 - y1;
    else
        height = y1 - y2;
}

public void FindArea()
/* PRECONDITION [[ (0 < width) && (0 < height) ]] */
/* POSTCONDITION [[ 0 < area]] */
{
    this.FindWidth(); // implicit-method-call statement
    this.FindHeight(); // implicit-method-call statement
    area = width * height;
}

public void Enlarge(int n)
// enlarging a rectangle along x-axis and y-axis
// by a scale factor(n) about the origin (0,0)
{
    x1 = x1 * n;
    y1 = y1 * n;
    x2 = x2 * n;
    y2 = y2 * n;
}

The class Rect_C implements the interface-type Interface_Rect_C and inherits from the class Rect_A. Therefore, the class Rect_C includes implicitly all fields and methods declared in the the class Rect_A. There are additionally three new fields in the Rect_C: width, height and area which are used to store the state of a rectangle object. The class Rect_C has also the new methods FindWidth, FindHeight and FindArea like the class Rect_B, but these methods do not have local variables. In addition, the class Rect_C has the method Enlarge which transforms a rectangle by a scale factor. Attention is here directed to modeling methods which has a pre-and-post condition, in other words, how this condition is interpreted after modeling the methods.

interface Interface_Quad extends Interface_Rect_C
{
    public void Quad_CONST(int c1, int d1, int c2, int d2);
    public int TriArea2(int p1, int q1, int p2, int q2, int p3, int q3);
    public void FindArea();
    public void Enlarge(int n);
}
class Quad extends Rect_C implements Interface_Quad

/* INVARIANT [[ (tri_1 <= area) && (tri_2 <= area) ]] */
{
    // the third coordinate
    protected int x3;
    protected int y3;

    // the fourth coordinate
    protected int x4;
    protected int y4;

    // finding the area of a quadrilateral by dividing it into two triangles
    protected int tri_1;
    protected int tri_2;

    public void Quad_CONST(int c1, int d1, int c2, int d2)
    {
        x3 = c1;
        y3 = d1;
        x4 = c2;
        y4 = d2;
        tri_1 = 0;
        tri_2 = 0;
    }

    public int TriArea2(int p1, int q1, int p2, int q2, int p3, int q3)
    //Calculate the area of a triangle, but return the double area
    //JaCo does not recognize real numbers, so that it could not have divide-operation.
    {
        int doubleArea_local;
        doubleArea_local = ((p2 - p1) * (q3 - q1)) - ((p3 - p1) * (q2 - q1));
        return doubleArea_local;
    }

    public void FindArea() // overrrding the method FindArea() defined in the upper-class
    {
        //implicit-method-call expressions
        tri_1 = this.TriArea2(x1, y1, x2, y2, x3, y3);
        tri_2 = this.TriArea2(x3, y3, x4, y4, x1, y1);
        area = tri_1 + tri_2;
    }

    public void Enlarge(int n) // overriding the method
    {
        x1 = x1 * n; y1 = y1 * n;
        x2 = x2 * n; y2 = y2 * n;
        x3 = x3 * n; y3 = y3 * n;
        x4 = x4 * n; y4 = y4 * n;
    }
}
The class Quad models a quadrilateral which has four vertexes. The class Quad inherits from the class Rect.C which has the fields representing only two vertices, so that there should be two more fields indicating other two vertices in the class Quad. Although the class Quad includes the fields height and width, it does not use these fields to calculate the area of a quadrilateral. Instead, the class Quad declares two new fields tri.1 and tri.2 to store each area of two triangles to which a quadrilateral can be divided by a diagonal. In the following, we will sketch briefly a geometry theory [O’R98] for how the method TriArea2 is constructed.

If we have a triangle ABC where the position vectors of A, B and C relative to an origin O are a, b and c, respectively, then the area of the triangle can be calculated with $\theta = \angle BAC$ as follows,

$$
\triangle ABC = \frac{1}{2} |\overrightarrow{AB}| |\overrightarrow{AC}| \sin \theta \\
= \frac{1}{2} |b - a||c - a| \sin \theta \\
= \frac{1}{2} |(b - a) \times (c - a)| \\
= \frac{1}{2} |a \times b + b \times c + c \times a|
$$

Then, the method FindArea calculates each triangle of a quadrilateral by calling the method TriArea2 twice. Regrettably, the method TriArea2 represents the double area of each triangle because myJava does not have the division operation. So, the field area stores actually the double area of a quadrilateral. Furthermore, much attention should be given to the fact that the methods FindArea and Enlarge override the methods defined in the class Rect.C.

```java
class LinearTrans {
    protected int areaBefore;
    protected int areaAfter;
    protected int scaleFactor;

    public void LinearTrans_CONST(int n) {
        areaBefore = 0;
    }
}
```
The class \textit{LinearTrans} is accessing an object of the classes \textit{Rect.C} or \textit{Quad} by using a method which takes an object as a parameter. The method \textit{CollectingArea} is first to take and store the area of an object, then to enlarge the object by a scale factor which should be greater than zero. Next, the method \textit{CollectingArea} is again to take and store the area of the translated object. With this procedure, we make an assertion as a pre-and-post condition. Given that \textit{areaAfter} is greater than \textit{areaBefore} and a scale factor is greater than zero, \textit{areaAfter} is also greater than \textit{areaBefore} after executing the method \textit{CollectingArea}. 
Chapter 6

Conclusion

6.1 Discussion and Future work

Method-call statement and expression

We are consequently made to see the following table, having provided a summary of interpreting method-call statements and expressions as type-theoretic models.

<table>
<thead>
<tr>
<th>Implicit-method-call</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>$S[this.id_{method}((Expressions)^*)]]$</td>
</tr>
<tr>
<td>Expression</td>
<td>$\tau[this.id_{method}((Expression)^*)]]$</td>
</tr>
<tr>
<td></td>
<td>Mutator</td>
</tr>
<tr>
<td></td>
<td>Being modeled with a parameterized function and a dependent record type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Explicit-method-call</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>$S[id_{para}.id_{method}((Expressions)^*)]]$</td>
</tr>
<tr>
<td>Expression</td>
<td>$\tau[id_{para}.id_{method}((Expression)^*)]]$</td>
</tr>
<tr>
<td></td>
<td>Mutator</td>
</tr>
<tr>
<td></td>
<td>Being modeled with a dependent record type and record field-selection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field-method-call</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>$S[id_{field}.id_{method}((Expressions)^*)]]$</td>
</tr>
<tr>
<td>Expression</td>
<td>$\tau[id_{field}.id_{method}((Expression)^*)]]$</td>
</tr>
<tr>
<td></td>
<td>Mutator</td>
</tr>
<tr>
<td></td>
<td>Being modeled with a dependent record type, record field-selection, and synchronization</td>
</tr>
</tbody>
</table>
Despite these findings, there remain basic limitations inherent in this approach. For example, implicit method-call statements and expressions can call only methods that were defined previously in the same class. Moreover, explicit or field method-call statements and expressions can call only methods defined in the other class which must be declared before. Put otherwise, it is impossible to model method-call statements and expressions which are calling methods that will be defined afterwards. For that reason, another limitation is that we can not model mutual invocation of methods or mutual recursive calls.

The universal state $\Omega$

A few further points need to be made with regard to managing the universal state $\Omega$, which reflects all states of objects in the context.

- As the number of classes declared in the context increases, it becomes very complicated to manage the $\Omega$-state as we have seen the case in Chapter 5.

- As mentioned earlier in Chapter 2 that we assume there is only one instance for each class, it is impossible to model individual objects of different state at the same time altogether, which are instantiated from the same class. To solve this problem, the following tentative $\Omega$-state can be defined with $List$-date type\(^1\). It is believed that further experimentation with this definition is worthwhile.

$$
\Omega_{\{B:(A_1,A_3,A_4);(A_1,A_3);(A_1,A_2);A_1}\} \equiv \langle l_B : List(St(B)) , l_{A_4} : List(St(A_4)) , l_{A_3} : List(St(A_3)) , l_{A_2} : List(St(A_2)) , l_{A_1} : List(St(A_1)) \rangle
$$

In addition, let’s have a look at the following method $Add(InterfaceStorage x, InterfaceStorage y)$ with the classes and their interface-types given in Chapter 2. Here, we assume that this method is declared as a member of the class Supplier.

\(^1\)Thanks to personal communication with Dr Robin Adams
class Supplier {

... 

public void Add(InterfaceStorage x, InterfaceStorage y) 
{
    sum = sum + (x.GetStock() + y.GetStock());
}

... 
}

This method will be interpreted by JaCo as follows,

Let Add_Supplier :
  InterfaceStorage$\rightarrow$InterfaceStorage$\rightarrow$Omega$\rightarrow$Omega $:=$
  fun (x : InterfaceStorage)(y : InterfaceStorage)(w : Omega) $\Rightarrow$
  let w1 := w in
  let Assign1_Add := fun (aw1 : Omega) $\Rightarrow$
    ( mk_Omega aw1.(label_storage)
      aw1.(label_newstorage)
      ( mk_SSupplier (aw1.(label_supplier).(sum) +
        (x.(i_label_GetStock_Storage) aw1) + (y.(i_label_GetStock_Storage) aw1))
       aw1.(label_supplier).(extra)
       aw1.(label_supplier).(cell) )
      aw1.(label_manager) )
  in let w2 := Assign1_Add w1
  in w2.

If this method may be required to take two different instances of the classes 
Storage or NewStorage, i.e. each instance has its own state, then the above interpretation is definitely not correct because the two explicit-method-call expressions $x$.GetStock() and $y$.GetStock() are modeled as taking the same state $aw1$. This problem occurs due to the assumption mentioned above that there exists only one instance for each class in the context.

Therefore, there should be another modification for interpreting a method with the $\Omega$-state. For example, the above method can be interpreted with the following type:

$$\text{InterfaceStorage} \rightarrow \text{InterfaceStorage} \rightarrow \Omega_1 \rightarrow \Omega_2 \rightarrow (\Omega_1 \times \Omega_2)$$

where $\Omega_1$ is for object states corresponding to all first instances of classes in the context, and $\Omega_2$ for object states corresponding to all second instances of classes. However, this suggestion leaves more to be investigated and answered.
CHAPTER 6. CONCLUSION

Modeling subtype-polymorphism and dynamic-dispatch without interface-types

This thesis is predicated on the notion of interface-types on which subtype-polymorphism and dynamic-dispatch are realized. However, Java programmers have generally slighted using interface-types on implementing subtype polymorphism and dynamic dispatch. For example, the method `Supply(InterfaceStorage x)` in the class `Supply` can be rewritten as follows without interface-types

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

where the method parameter’s type is the class `Storage`, not the interface-type `InterfaceStorage`. Here, we assume that we do not define interface-types anymore, so that there do not exist interface-types in the context. Nonetheless, we should note that we can still define the record types `InterfaceStorage` or `InterfaceNewStorage` from the record types `MStorage` or `MNewStorage` by means of extracting type information from each method.

![UML Class Diagram](image)

**Figure 6.1: Modeling direct inheritance relationship**

We could interpret direct inheritance relationship which does not have interface-types as type-theoretic models as above. Regrettably, this approach causes a model correctness problem when it is used for interpreting subtype-polymorphism
and dynamic-dispatch. The above method \texttt{Supply(Storage x)} is interpreted as shown in Chapter 2,

\begin{verbatim}
Let Supply_Supplier
( ClearExtra : Omega->Omega ) : InterfaceStorage -> Omega -> Omega :=
fun(x : InterfaceStorage) (w : Omega) => ......
\end{verbatim}

This interpretation does not not satisfy the model correctness because although the method \texttt{Supply(Storage x)} takes only an instance the class \texttt{Storage} or \texttt{NewStorage}, the function \texttt{Supply_Supplier} can take any object of the type \texttt{InterfaceStorage}. In other words, unrelated classes may produce objects of the same interface type. This observation leads us, necessarily, to declare the method as taking an object of the interface-type \texttt{InterfaceStorage}. However, further research should be directed at determining how methods taking a class instance as an argument without an interface-type are interpreted in our type-theoretic models.

### 6.2 Summary

This thesis has attempted to sketch out how to represent object-oriented programs in the intensional type theory UTT. 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 myJava 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 myJava programs. With such a tool support, a case study has been done with the myJava programs given in Chapter 2 and Chapter 5 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.

Although the present study offers an initial contribution to the literature concerning type-theoretic interpretation of object-oriented programs in intensional type theory with dependent-record types and coercive subtyping, there remain with strong possibility a range of problems to be tackled such as modeling inner-classes\footnote{The syntax of inner-class is defined in the BNF definition given in Appendix C.} as dependent record types and generic-classes [Hor02] as a parametric
polymorphism in Coq.

Having come to the end of our discussion, it should be concluded that I proposed an intensional type theory with dependent record types and coercive subtyping as a better and simple way to model and verify object-oriented programs in comparison with the complex semantics, e.g. object calculi [AC96].

Many aspects of object-oriented programming are worth preserving. In particular, self-referential method calls [Set07] and programs that make use of mutually recursive object classes should be reserved for a more extensive study of how these features can be modeled in an intensional type theory with dependent record types and coercive subtyping. Furthermore, if our model is extended to real numbers, JaCo could be pursued to the point where it links up with industrial applications.

Now that the thesis is complete, it is hoped that this project is able to provide an impetus for strengthening a functional interpretation of object-oriented programs.
Bibliography


[BT98] G. Betarte and A. Tasistro. *Extension of Martin-Löf’s type theory with record types and subtyping*, chapter 2 in Twenty-Five Years of


http://java.sun.com/javase/6/docs/api/.


Appendix A

Coq codes for case study

Defining the $\Omega$-state

Require Export MyLibrary.

(* The type of states for each object *)

Record SRect_A : Set := mk_SRect_A { x1 : nat ;
    y1 : nat ;
    x2 : nat ;
    y2 : nat }.

Record SRect_B : Set := mk_SRect_B { label_rect_a_1 := SRect_A }.

Record SRect_C : Set := mk_SRect_C { label_rect_a_2 := SRect_A ;
    width : nat ;
    height : nat ;
    area : nat }.

Record SQuad : Set := mk_SQuad { label_rect_c_1 := SRect_C ;
    x3 : nat ;
    y3 : nat ;
    x4 : nat ;
    y4 : nat ;
    tri_1 : nat ;
    tri_2 : nat }.

Record SLinearTrans : Set := mk_SLinearTrans { areaBefore : nat ;
    areaAfter : nat ;
    scaleFactor : nat }.

(* Define Omega : the type of states for all objects *)

Record Omega : Set := mk_Omega { label_rect_a := SRect_A ;
    label_rect_b := SRect_B ;
    label_rect_c := SRect_C ;
    label_quad := SQuad ;
    label_lineartrans := SLinearTrans }.

109
APPENDIX A. COQ CODES FOR CASE STUDY

(* Define Omegas extended with local variables *)

Record Omega_FindWidth_Rect_B : Set := mk_Omega_FindWidth_Rect_B {
  label_omega_findwidth_rect_b :> Omega ;
  width_local : nat }.

Record Omega_FindHeight_Rect_B : Set := mk_Omega_FindHeight_Rect_B {
  label_omega_findheight_rect_b :> Omega ;
  height_local : nat }.

Record Omega_FindArea_Rect_B : Set := mk_Omega_FindArea_Rect_B {
  label_omega_findarea_rect_b :> Omega ;
  area_local : nat }.

Record Omega_TriArea2_Quad : Set := mk_Omega_TriArea2_Quad {
  label_omega_triarea2_quad :> Omega ;
  doubleArea_local : nat }.

Modeling the class Rect_A

(* Modeling the class "Rect_A" *)

(* Modeling the method Rect_A_CONST of the class Rect_A *)

Let Rect_A_CONST_RECT_A : nat -> nat -> nat -> nat -> Omega -> Omega :=
  let w1 := w
  in let Assign1_RECT_A_CONST := fun (aw1 : Omega) =>
    ( mk_Omega ( mk_SRect_A a1
    aw1.(label_rect_a).(y1)
    aw1.(label_rect_a).(x2)
    aw1.(label_rect_a).(y2))
    aw1.(label_rect_b)
    aw1.(label_rect_c)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_RECT_A_CONST w1
  in let Assign2_RECT_A_CONST := fun (aw2 : Omega) =>
    ( mk_Omega ( mk_SRect_A aw2.(label_rect_a).(x1)
    b1
    aw2.(label_rect_a).(x2)
    aw2.(label_rect_a).(y2) )
    aw2.(label_rect_b)
    aw2.(label_rect_c)
    aw2.(label_quad)
    aw2.(label_lineartrans) )
  in let w3 := Assign2_RECT_A_CONST w2
  in let Assign3_RECT_A_CONST := fun (aw3 : Omega) =>
    ( mk_Omega ( mk_SRect_A aw3.(label_rect_a).(x1)
    a2
    aw3.(label_rect_a).(y1) )
APPENDIX A. COQ CODES FOR CASE STUDY

```coq
(* Define a record type representing all methods of the class Rect_A *)
Record MRect_A : Set := mk_MRect_A { label_Rect_A_CONST_Rect_A : Unit Rect_A_CONST_Rect_A }.

(* Model the class Rect_A as a record type *)
Record Rect_A : Set := mk_Rect_A { label_srect_a : SRect_A ;
    label_mrect_a :> MRect_A }.

Modeling the class Rect_B

(* Modeling the class "Rect_B" *)

(* Modeling the method Rect_A_CONST of the class Rect_B *)
let w1 := w in
let Assign1_Rect_A_CONST := fun (aw1 : Omega) =>
  ( mk_Omega
    ( mk_SRect_B ( mk_SRect_A a1 aw1.(label_rect_b).(y1) aw1.(label_rect_b).(x2) aw1.(label_rect_b).(y2) )
    aw1.(label_rect_c)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_Rect_A_CONST w1
  in let Assign2_Rect_A_CONST := fun (aw2 : Omega) =>
    ( mk_Omega
      ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_b).(x1) b1 aw2.(label_rect_b).(x2) aw2.(label_rect_b).(y2) )
      aw2.(label_rect_c) )

(* Modeling the method Rect_B_DIGEST of the class Rect_B *)
Let Rect_B_DIGEST_Rect_B : nat->nat->Omega := fun(a : nat)(b : nat) =>
let w1 := w in
let Assign1_Rect_B_DIGEST := fun (aw1 : Omega) =>
  ( mk_Omega
    ( mk_SRect_B ( mk_SRect_A aw1.(label_rect_a) .(x1) aw1.(label_rect_b) .(y1) aw1.(label_rect_b).(x2) aw1.(label_rect_b).(y2) )
    aw1.(label_rect_c)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_Rect_B_DIGEST w1
  in let Assign2_Rect_B_DIGEST := fun (aw2 : Omega) =>
    ( mk_Omega
      ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_a) .(x1) b1 aw2.(label_rect_b) .(x2) aw2.(label_rect_b).(y2) )
      aw2.(label_rect_c) )
```

Modeling the class Rect_B

(* Modeling the class "Rect_B" *)

(* Modeling the method Rect_A_CONST of the class Rect_B *)
let w1 := w in
let Assign1_Rect_A_CONST := fun (aw1 : Omega) =>
  ( mk_Omega
    ( mk_SRect_B ( mk_SRect_A a1 aw1.(label_rect_a).(x1) aw1.(label_rect_b).(y1) aw1.(label_rect_b).(x2) aw1.(label_rect_b).(y2) )
    aw1.(label_rect_c)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_Rect_A_CONST w1
  in let Assign2_Rect_A_CONST := fun (aw2 : Omega) =>
    ( mk_Omega
      ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_a) .(x1) b1 aw2.(label_rect_b) .(x2) aw2.(label_rect_b).(y2) )
      aw2.(label_rect_c) )
```

Modeling the class Rect_B

(* Modeling the class "Rect_B" *)

(* Modeling the method Rect_A_CONST of the class Rect_B *)
let w1 := w in
let Assign1_Rect_A_CONST := fun (aw1 : Omega) =>
  ( mk_Omega
    ( mk_SRect_B ( mk_SRect_A a1 aw1.(label_rect_a) .(x1) aw1.(label_rect_b) .(x2) aw1.(label_rect_b).(y2) )
    aw1.(label_rect_c)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_Rect_A_CONST w1
  in let Assign2_Rect_A_CONST := fun (aw2 : Omega) =>
    ( mk_Omega
      ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_a) .(x1) b1 aw2.(label_rect_b) .(x2) aw2.(label_rect_b).(y2) )
      aw2.(label_rect_c) )
```
APPENDIX A. COQ CODES FOR CASE STUDY

aw2.(label_quad)
aw2.(label_lineartrans)  )
in let w3 := Assign2_Rect_A_CONST w2
in let Assign3_Rect_A_CONST := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a)
    ( mk_SRect_B ( mk_SRect_A aw3.(label_rect_b).(x1)
        aw3.(label_rect_b).(y1)
        a2
        aw3.(label_rect_b).(y2) )
    aw3.(label_rect_c)
    aw3.(label_quad)
  aw3.(label_lineartrans)  )
in
let w4 := Assign3_Rect_A_CONST w3
in let Assign4_Rect_A_CONST := fun (aw4 : Omega) =>
  ( mk_Omega aw4.(label_rect_a)
    ( mk_SRect_B ( mk_SRect_A aw4.(label_rect_b).(x1)
        aw4.(label_rect_b).(y1)
        aw4.(label_rect_b).(x2)
        b2 )
    aw4.(label_rect_c)
    aw4.(label_quad)
  aw4.(label_lineartrans)  )
in
let w5 := Assign4_Rect_A_CONST w4
in w5.

(* Modeling the method FindWidth of the class Rect_B *)
Let FindWidth_Rect_B
: Omega-> nat:= fun(w : Omega) =>
let beforeEw := mk_Omega_FindWidth_Rect_B w 0
in let If1_FindWidth := fun (iw1:Omega_FindWidth_Rect_B) =>
  if ( LessThan iw1.(label_rect_b).(x1) iw1.(label_rect_b).(x2))
  then let Assign1_FindWidth := fun (aw1 : Omega_FindWidth_Rect_B) =>
      ( mk_Omega_FindWidth_Rect_B
        ( mk_Omega aw1.(label_rect_a)
          ( mk_SRect_B ( mk_SRect_A aw1.(label_rect_b).(x1)
              aw1.(label_rect_b).(y1)
              aw1.(label_rect_b).(x2)
              aw1.(label_rect_b).(y2))
          aw1.(label_rect_c)
          aw1.(label_quad)
          aw1.(label_lineartrans)  )
        (aw1.(label_rect_b).(x2) - aw1.(label_rect_b).(x1)) )
  else let Assign2_FindWidth := fun (aw2 : Omega_FindWidth_Rect_B) =>
      ( mk_Omega_FindWidth_Rect_B
        ( mk_Omega aw2.(label_rect_a)
          ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_b).(x1)
              aw2.(label_rect_b).(y1)
              aw2.(label_rect_b).(x2)
              aw2.(label_rect_b).(y2))
          aw2.(label_rect_c)
          aw2.(label_quad)
          aw2.(label_lineartrans)  )
        (aw2.(label_rect_b).(x1) - aw2.(label_rect_b).(x2)) )


in Assign2_FindWidth iw1
in let afterEw := ( If1_FindWidth beforeEw )
in afterEw.(width_local).

(* Modeling the method FindHeight of the class Rect_B *)
Let FindHeight_Rect_B
: Omega-> nat:= fun(w : Omega) =>
let beforeEw := mk_Omega_FindHeight_Rect_B w 0
in let If1_FindHeight := fun (iw1 : Omega_FindHeight_Rect_B) =>
  if ( LessThan iw1.(label_rect_b).(y1) iw1.(label_rect_b).(y2))
  then let Assign1_FindHeight := fun (aw1 : Omega_FindHeight_Rect_B) =>
    ( mk_Omega_FindHeight_Rect_B
      ( mk_Omega aw1.(label_rect_a)
        ( mk_SRect_B ( mk_SRect_A aw1.(label_rect_b).(x1) aw1.(label_rect_b).(y1) aw1.(label_rect_b).(x2) aw1.(label_rect_b).(y2) )
      aw1.(label_rect_c)
      aw1.(label_quad)
      aw1.(label_lineartrans) )
    (aw1.(label_rect_b).(y2) - aw1.(label_rect_b).(y1)) )
in Assign1_FindHeight iw1
else let Assign2_FindHeight := fun (aw2 : Omega_FindHeight_Rect_B) =>
    ( mk_Omega_FindHeight_Rect_B
      ( mk_Omega aw2.(label_rect_a)
        ( mk_SRect_B ( mk_SRect_A aw2.(label_rect_b).(x1) aw2.(label_rect_b).(y1) aw2.(label_rect_b).(x2) aw2.(label_rect_b).(y2) )
      aw2.(label_rect_c)
      aw2.(label_quad)
      aw2.(label_lineartrans) )
    (aw2.(label_rect_b).(y1) - aw2.(label_rect_b).(y2)) )
in Assign2_FindHeight iw1
in let afterEw := ( If1_FindHeight beforeEw )
in afterEw.(height_local).

(* Modeling the method FindArea of the class Rect_B *)
Let FindArea_Rect_B
( FindWidth : Omega->nat )( FindHeight : Omega->nat ) : Omega->nat := fun(w : Omega) =>
let beforeEw := mk_Omega_FindArea_Rect_B w 0
in let Assign1_FindArea := fun (awl : Omega_FindArea_Rect_B) =>
    ( mk_Omega_FindArea_Rect_B
      ( mk_Omega awl.(label_rect_a)
        ( mk_SRect_B ( mk_SRect_A awl.(label_rect_b).(x1) awl.(label_rect_b).(y1) awl.(label_rect_b).(x2) awl.(label_rect_b).(y2) )
      awl.(label_rect_c)
      awl.(label_quad)
      awl.(label_lineartrans) )
    ((FindWidth awl) * (FindHeight awl)) )
in let afterEw := ( Assign1_FindArea beforeEw )
in afterEw.(area_local).
APPENDIX A. COQ CODES FOR CASE STUDY

(* Define a record type representing all methods of the class Rect_B *)
Record MRect_B : Set := mk_MRect_B {
  label_Rect_A_CONST_Rect_B : Unit Rect_A_CONST_Rect_B ;
  label_FindWidth_Rect_B : Unit FindWidth_Rect_B ;
  label_FindHeight_Rect_B : Unit FindHeight_Rect_B ;
  label_FindArea_Rect_B :
    Unit (FindArea_Rect_B(label_FindWidth_Rect_B : ID(Omega->nat) )
      (label_FindHeight_Rect_B : ID(Omega->nat) )) }.

(* Model the class Rect_B as a record type *)
Record Rect_B : Set := mk_Rect_B { label_srect_b : SRect_B ;
  label_mrect_b :> MRect_B }.

Modeling the class Rect_C

(* Modeling the class "Rect_C" *)

(* Modeling the method Rect_A_CONST of the class Rect_C *)
Let Rect_A_CONST_Rect_C :
  nat->nat->nat->nat->Omega->Omega :=
    let w1 := w in let Assign1_Rect_A_CONST := fun (aw1 : Omega) =>
      ( mk_Omega aw1.(label_rect_a)
        aw1.(label_rect_b)
        ( mk_SRect_C ( mk_SRect_A a1
          aw1.(label_rect_c).(y1)
        aw1.(label_rect_c).(x2)
        aw1.(label_rect_c).(y2))
      aw1.(label_rect_c).(width)
      aw1.(label_rect_c).(height)
      aw1.(label_rect_c).(area)
    aw1.(label_quad)
    aw1.(label_lineartrans) )
  in let w2 := Assign1_Rect_A_CONST w1
  in let Assign2_Rect_A_CONST := fun (aw2 : Omega) =>
    ( mk_Omega aw2.(label_rect_a)
      aw2.(label_rect_b)
      ( mk_SRect_C ( mk_SRect_A aw2.(label_rect_c).(x1)
        b1
      aw2.(label_rect_c).(x2)
      aw2.(label_rect_c).(y2))
    aw2.(label_rect_c).(width)
    aw2.(label_rect_c).(height)
    aw2.(label_rect_c).(area) )
  in let w3 := Assign2_Rect_A_CONST w2
  in let Assign3_Rect_A_CONST := fun (aw3 :Omega) =>
    ( mk_Omega aw3.(label_rect_a)
APPENDIX A. COQ CODES FOR CASE STUDY

aw3.(label_rect_b)
( mk_SRect_C ( mk_SRect_A aw3.(label_rect_c).(x1)
    aw3.(label_rect_c).(y1)
    a2
    aw3.(label_rect_c).(y2) )
    aw3.(label_rect_c).(width)
    aw3.(label_rect_c).(height)
    aw3.(label_rect_c).(area) )
aw3.(label_quad)
aw3.(label_lineartrans) )
in let w4 := Assign3_Rect_A_CONST w3
in let Assign4_Rect_A_CONST := fun (aw4 : Omega) =>
( mk_Omega aw4.(label_rect_a)
aw4.(label_rect_b)
( mk_SRect_C ( mk_SRect_A aw4.(label_rect_c).(x1)
    aw4.(label_rect_c).(y1)
    aw4.(label_rect_c).(x2)
    b2 )
    aw4.(label_rect_c).(width)
    aw4.(label_rect_c).(height)
    aw4.(label_rect_c).(area) )
aw4.(label_quad)
aw4.(label_lineartrans) )
in let w5 := Assign4_Rect_A_CONST w4
in w5.

(* Modeling the method Rect_C_CONST of the class Rect_C *)
Let Rect_C_CONST_Rect_C : Omega-> Omega:= fun(w : Omega) =>
let w1 := w in
let Assign1_Rect_C_CONST := fun (aw1 : Omega) =>
( mk_Omega aw1.(label_rect_a)
aw1.(label_rect_b)
( mk_SRect_C ( mk_SRect_A aw1.(label_rect_c).(x1)
    aw1.(label_rect_c).(y1)
    aw1.(label_rect_c).(x2)
    aw1.(label_rect_c).(y2) )
    0
    aw1.(label_rect_c).(height)
    aw1.(label_rect_c).(area) )
aw1.(label_quad)
aw1.(label_lineartrans) )
in let w2 := Assign1_Rect_C_CONST w1
in let Assign2_Rect_C_CONST := fun (aw2 : Omega) =>
( mk_Omega aw2.(label_rect_a)
aw2.(label_rect_b)
( mk_SRect_C ( mk_SRect_A aw2.(label_rect_c).(x1)
    aw2.(label_rect_c).(y1)
    aw2.(label_rect_c).(x2)
    aw2.(label_rect_c).(y2) )
    aw2.(label_rect_c).(width)
    0
    aw2.(label_rect_c).(area) )
aw2.(label_quad)
APPENDIX A. COQ CODES FOR CASE STUDY

```
aw2.(label_lineartrans) )
in let w3 := Assign2_Rect_C_CONST w2
in let Assign3_Rect_C_CONST := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a)
    aw3.(label_rect_b)
    ( mk_SRect_C ( mk_SRect_A aw3.(label_rect_c).(x1)
      aw3.(label_rect_c).(y1)
      aw3.(label_rect_c).(x2)
      aw3.(label_rect_c).(y2) )
    aw3.(label_rect_c).(width)
    aw3.(label_rect_c).(height)
    0 )
  aw3.(label_quad)
  )
aw3.(label_lineartrans) )
in let w4 := Assign3_Rect_C_CONST w3
in w4.

(* Modeling the method GetArea of the class Rect_C *)
Let GetArea_RECT_C
  : Omega -> nat := fun(w : Omega) => w.(label_rect_c).(area).

(* Modeling the method FindWidth of the class Rect_C *)
Let FindWidth_RECT_C
  : Omega -> omega := fun(w : Omega) =>
in let w1 := w
in let If1_FindWidth := fun (iw1 : omega) =>
  if ( LessThan iw1.(label_rect_c).(x1) iw1.(label_rect_c).(x2))
  then let Assign1_FindWidth := fun (aw1 : Omega) =>
    ( mk_Omega aw1.(label_rect_a)
      aw1.(label_rect_b)
      ( mk_SRect_C ( mk_SRect_A aw1.(label_rect_c).(x1)
        aw1.(label_rect_c).(y1)
        aw1.(label_rect_c).(x2)
        aw1.(label_rect_c).(y2) )
      (aw1.(label_rect_c).(x2) - aw1.(label_rect_c).(x1))
      aw1.(label_rect_c).(height)
      aw1.(label_rect_c).(area) )
    aw1.(label_quad)
    )
in Assign1_FindWidth iw1
else let Assign2_FindWidth := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_rect_a)
    aw2.(label_rect_b)
    ( mk_SRect_C ( mk_SRect_A aw2.(label_rect_c).(x1)
      aw2.(label_rect_c).(y1)
      aw2.(label_rect_c).(x2)
      aw2.(label_rect_c).(y2) )
    (aw2.(label_rect_c).(x1) - aw2.(label_rect_c).(x2))
    aw2.(label_rect_c).(height)
    aw2.(label_rect_c).(area) )
  aw2.(label_quad)
  )
in Assign2_FindWidth iw1
in let w2 := If1_FindWidth w1
```
APPENDIX A. COQ CODES FOR CASE STUDY

(* Modeling the method FindHeight of the class Rect_C *)

Let FindHeight_Rect_C : Omega->Omega := fun(w : Omega) =>
  let w1 := w
  in let If1_FindHeight := fun (iw1:Omega) =>
    if ( LessThan iw1.(label_rect_c).(y1) iw1.(label_rect_c).(y2))
    then let Assign1_FindHeight := fun (aw1 : Omega) =>
      ( mk_Omega aw1.(label_rect_a)
        aw1.(label_rect_b)
        ( mk_SRect_C (mk_SRect_A aw1.(label_rect_c).(x1)
          aw1.(label_rect_c).(y1)
          aw1.(label_rect_c).(x2)
          aw1.(label_rect_c).(y2))
        aw1.(label_rect_c).(width)
        (aw1.(label_rect_c).(y2) = aw1.(label_rect_c).(y1))
        aw1.(label_rect_c).(area) )
      in Assign1_FindHeight iw1
    else let Assign2_FindHeight := fun (aw2 : Omega) =>
      ( mk_Omega aw2.(label_rect_a)
        aw2.(label_rect_b)
        ( mk_SRect_C (mk_SRect_A aw2.(label_rect_c).(x1)
          aw2.(label_rect_c).(y1)
          aw2.(label_rect_c).(x2)
          aw2.(label_rect_c).(y2))
        aw2.(label_rect_c).(width)
        (aw2.(label_rect_c).(y1) = aw2.(label_rect_c).(y2))
        aw2.(label_rect_c).(area) )
      in Assign2_FindHeight iw1
  in w2.

(* Modeling the method FindArea of the class Rect_C *)

Let FindArea_Rect_C : Omega->Omega := fun(w : Omega) =>
  let w1 := w in
  let CallStatement1_FindArea := fun (cw1 : Omega) => FindWidth cw1
  in let w2 := CallStatement1_FindArea w1
  in let CallStatement2_FindArea := fun (cw2 : Omega) => FindHeight cw2
  in let w3 := CallStatement2_FindArea w2
  in let Assign1_FindArea := fun (aw1 :Omega) =>
    ( mk_Omega aw1.(label_rect_a)
      aw1.(label_rect_b)
      ( mk_SRect_C (mk_SRect_A aw1.(label_rect_c).(x1)
        aw1.(label_rect_c).(y1)
        aw1.(label_rect_c).(x2)
        aw1.(label_rect_c).(y2))
      aw1.(label_rect_c).(width)
      (aw1.(label_rect_c).(y2) = aw1.(label_rect_c).(y1))
      aw1.(label_rect_c).(area) )
  in Assign1_FindArea w1
aw1.(label_rect_c).(height) 
  (aw1.(label_rect_c).(width) * aw1.(label_rect_c).(height)) 

  aw1.(label_quad) 
  aw1.(label_lineartrans) 
)
in let w4 := Assign1_FindArea w3 
in w4.

(* Modeling the method Enlarge of the class Rect_C *)
Let Enlarge_Rect_C :
  nat->Omega->Omega := fun(n : nat)(w : Omega) =>
let w1 := w 
in let Assign1_Enlarge := fun (aw1 : Omega) =>
  ( mk_Omega aw1.(label_rect_a) 
    aw1.(label_rect_b) 
    (mk_SRect_C (mk_SRect_A (aw1.(label_rect_c).(x1) * n) 
         aw1.(label_rect_c).(y1) 
         aw1.(label_rect_c).(x2) 
         aw1.(label_rect_c).(y2) ) 
    aw1.(label_rect_c).(width) 
    aw1.(label_rect_c).(height) 
    aw1.(label_rect_c).(area) )

  aw1.(label_quad) 
  aw1.(label_lineartrans) 
)
in let w2 := Assign1_Enlarge w1 
in let Assign2_Enlarge := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_rect_a) 
    aw2.(label_rect_b) 
    (mk_SRect_C (mk_SRect_A aw2.(label_rect_c).(x1) 
         (aw2.(label_rect_c).(y1) * n) 
         aw2.(label_rect_c).(x2) 
         aw2.(label_rect_c).(y2) ) 
    aw2.(label_rect_c).(width) 
    aw2.(label_rect_c).(height) 
    aw2.(label_rect_c).(area) )

  aw2.(label_quad) 
  aw2.(label_lineartrans) 
)
in let w3 := Assign2_Enlarge w2 
in let Assign3_Enlarge := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a) 
    aw3.(label_rect_b) 
    (mk_SRect_C (mk_SRect_A aw3.(label_rect_c).(x1) 
         aw3.(label_rect_c).(y1) 
         (aw3.(label_rect_c).(x2) * n) 
         aw3.(label_rect_c).(y2) ) 
    aw3.(label_rect_c).(width) 
    aw3.(label_rect_c).(height) 
    aw3.(label_rect_c).(area) )

  aw3.(label_quad) 
  aw3.(label_lineartrans) 
)
in let w4 := Assign3_Enlarge w3 
in let Assign4_Enlarge := fun (aw4 : Omega) =>
  ( mk_Omega aw4.(label_rect_a) 
    aw4.(label_rect_b) 
    (mk_SRect_C (mk_SRect_A aw4.(label_rect_c).(x1) 

APPENDIX A. COQ CODES FOR CASE STUDY

```coq
aw4.(label_rect_c).(y1)
aw4.(label_rect_c).(x2)
(aw4.(label_rect_c).(y2) * n) 
aw4.(label_rect_c).(width)
aw4.(label_rect_c).(height)
aw4.(label_rect_c).(area) )

in let w5 := Assign4_Enlarge w4
  in w5.

(* Define a record type representing all methods of the class Rect_C *)
Record MRect_C : Set := mk_MRect_C {
  label_Rect_A_CONST_Rect_C : Unit Rect_A_CONST_Rect_C ;
  label_Rect_C_CONST_Rect_C : Unit Rect_C_CONST_Rect_C ;
  label_GetArea_Rect_C : Unit GetArea_Rect_C ;
  label_FindWidth_Rect_C : Unit FindWidth_Rect_C ;
  label_FindHeight_Rect_C : Unit FindHeight_Rect_C ;
  label_FindArea_Rect_C :
    Unit (FindArea_Rect_C(label_FindWidth_Rect_C : ID(Omega->Omega) )
            (label_FindHeight_Rect_C : ID(Omega->Omega) )) ;
  label_Enlarge_Rect_C : Unit Enlarge_Rect_C }.

(* Model the class Rect_C as a record type *)
Record Rect_C : Set := mk_Rect_C { label_srect_c : SRect_C ;
                                   label_mrect_c => MRect_C }.

(* Model the interface-type Interface_Rect_C as a record type *)
Record Interface_Rect_C : Set := mk_Interface_Rect_C {
  i_label_Rect_A_CONST_Rect_C : nat->nat->nat->nat->Omega->Omega ;
  i_label_Rect_C_CONST_Rect_C : Omega->Omega ;
  i_label_GetArea_Rect_C : Omega->nat ;
  i_label_FindWidth_Rect_C : Omega->Omega ;
  i_label_FindHeight_Rect_C : Omega->Omega ;
  i_label_FindArea_Rect_C : Omega->Omega ;
  i_label_Enlarge_Rect_C : nat->Omega->Omega }.

(* Define a coercion between MRect_C and Interface_Rect_C *)
Coercion MRect_C_Interface_Rect_C (x : MRect_C) : Interface_Rect_C :=
  mk_Interface_Rect_C {
    label_Rect_A_CONST_Rect_C x : ID (nat->nat->nat->nat->Omega->Omega) ;
    label_Rect_C_CONST_Rect_C x : ID (Omega->Omega) ;
    label_GetArea_Rect_C x : ID (Omega->nat) ;
    label_FindWidth_Rect_C x : ID (Omega->Omega) ;
    label_FindHeight_Rect_C x : ID (Omega->Omega) ;
    label_FindArea_Rect_C x : ID (Omega->Omega) ;
    label_Enlarge_Rect_C x : ID (nat->Omega->Omega) }.

(* Define Omega extended with the interface-type Interface_Rect_C *)
Record Omega_Interface_Rect_C : Set := mk_Omega_Interface_Rect_C {
  label_omega_interface_rect_c :> Omega ;
  label_interface_rect_c : Interface_Rect_C }.
```
APPENDIX A. COQ CODES FOR CASE STUDY

Theorems generated from the class \textit{Rect\_C}

Theorem \textit{PrePost\_FindWidth\_Rect\_C} : \forall (r : \textit{Rect\_C}) (s : \textit{Omega}),
\(~ ( (s.(\text{label\_rect\_c}).(x1)) = (s.(\text{label\_rect\_c}).(x2)) ) \) \rightarrow
let \( s' := (r.(\text{label\_FindWidth\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(width))).

Theorem \textit{PrePost\_FindHeight\_Rect\_C} : \forall (r : \textit{Rect\_C})(s : \textit{Omega}),
\(~ ( (s.(\text{label\_rect\_c}).(y1)) = (s.(\text{label\_rect\_c}).(y2)) ) \) \rightarrow
let \( s' := (r.(\text{label\_FindHeight\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(height))).

Theorem \textit{PrePost\_FindArea\_Rect\_C} : \forall (r : \textit{Rect\_C})(s : \textit{Omega}),
\((0) < (s.(\text{label\_rect\_c}).(width)) /
(0) < (s.(\text{label\_rect\_c}).(height))) \rightarrow
let \( s' := (r.(\text{label\_FindArea\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(area))).

Proof of the above theorems

Definition \textit{gt\_lt\_dec} \( n m : (n < m) + (m < n). \)
Admitted.

Lemma \textit{FindWidth\_case1} : \forall (r : \textit{Rect\_C}) (s : \textit{Omega}),
\((s.(\text{label\_rect\_c}).(x1)) < (s.(\text{label\_rect\_c}).(x2)) \) \rightarrow
let \( s' := (r.(\text{label\_FindWidth\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(width))).
Admitted.

Lemma \textit{FindWidth\_case2} : \forall (r : \textit{Rect\_C}) (s : \textit{Omega}),
\((s.(\text{label\_rect\_c}).(x2)) < (s.(\text{label\_rect\_c}).(x1)) \) \rightarrow
let \( s' := (r.(\text{label\_FindWidth\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(width))).
Admitted.

(*----------------------------------------------------------------------*)

Theorem \textit{PrePost\_FindWidth\_Rect\_C} : \forall (r : \textit{Rect\_C}) (s : \textit{Omega}),
\(~ ( (s.(\text{label\_rect\_c}).(x1)) = (s.(\text{label\_rect\_c}).(x2)) ) \) \rightarrow
let \( s' := (r.(\text{label\_FindWidth\_Rect\_C}) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \) \( s \)
in \((0) < (s'.(\text{label\_rect\_c}).(width))).

Proof.
intros \( r s \).
simpl.
intros \( H1 \).
elim \( \text{gt\_lt\_dec} \ (s.(\text{label\_rect\_c}).(x1)) \ (s.(\text{label\_rect\_c}).(x2)) \).
Case " \( x1 < x2 \)."
apply \textit{FindWidth\_case1}.
Case " \( x2 < x1 \)."
apply \textit{FindWidth\_case2}.
Qed.
(*----------------------------------------------------------------------*)

Lemma \textit{FindHeight\_case1} : \forall (r : \textit{Rect\_C}) (s : \textit{Omega}),
\((s.(\text{label\_rect\_c}).(y1)) < (s.(\text{label\_rect\_c}).(y2)) \rightarrow

let \( s' \) := (\( r.\text{label\_FindHeight\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{height})\).
Admitted.

Lemma \text{FindHeight\_case2} : \forall (r : \text{Rect\_C})(s : \text{Omega}),
\((s.\text{label\_rect\_c}.(y2)) < (s.\text{label\_rect\_c}.(y1)) \rightarrow\)
let \( s' \) := (\( r.\text{label\_FindHeight\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{height})\).
Admitted.

(*******************************************************************************************)

Theorem \text{PrePost\_FindHeight\_Rect\_C} : \forall (r : \text{Rect\_C})(s : \text{Omega}),
( \neg ( \( s.\text{label\_rect\_c}.(y1) \) = \( s.\text{label\_rect\_c}.(y2) \) ) \rightarrow \)
let \( s' \) := (\( r.\text{label\_FindHeight\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{height})\).
Proof.
intros r s.
simpl.
intros H1.
elim (\text{gt\_lt\_dec} (s.\text{label\_rect\_c}.(y1)) (s.\text{label\_rect\_c}.(y2))).
Case "y1 < y2 ".
apply \text{FindHeight\_case1}.
Case "y2 < y1 ".
apply \text{FindHeight\_case2}.
Qed.
(*******************************************************************************************)

Lemma \text{FindArea\_mult} : \forall (n m : \text{nat}), \( 0 < n \\land \ 0 < m \rightarrow \ 0 < n*m \).
Admitted.

Lemma \text{FindArea\_width} : \forall (r : \text{Rect\_C}) (s : \text{Omega}),
( \neg ( \( s.\text{label\_rect\_c}.(x1) \) = \( s.\text{label\_rect\_c}.(x2) \) ) \rightarrow \)
let \( s' \) := (\( r.\text{label\_FindArea\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{width})\).
Admitted.

Lemma \text{FindArea\_height} : \forall (r : \text{Rect\_C}) (s : \text{Omega}),
( \neg ( \( s.\text{label\_rect\_c}.(y1) \) = \( s.\text{label\_rect\_c}.(y2) \) ) \rightarrow \)
let \( s' \) := (\( r.\text{label\_FindArea\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{height})\).
Admitted.

(*******************************************************************************************)

Theorem \text{PrePost\_FindArea\_Rect\_C} : \forall (r : \text{Rect\_C})(s : \text{Omega}),
\(( \neg ( \( s.\text{label\_rect\_c}.(x1) \) = \( s.\text{label\_rect\_c}.(x2) \) ) \land \)
\(( \neg ( \( s.\text{label\_rect\_c}.(y1) \) = \( s.\text{label\_rect\_c}.(y2) \) ) \) \rightarrow \)
let \( s' \) := (\( r.\text{label\_FindArea\_Rect\_C} \) : \text{ID} (\text{Omega}\rightarrow\text{Omega}) ) \( s \) in \((0) < (s'.\text{label\_rect\_c}.\text{area})\).
Proof.
intros r s.
inversion H1 as [H2 H3].
simpl.
apply \text{FindArea\_mult}.
APPENDIX A. COQ CODES FOR CASE STUDY

apply conj.
apply (FindArea_width r s) in H2.
simpl in H2.
assumption.
apply (FindArea_height r s) in H3.
simpl in H3.
assumption.
Qed.

(*******************************************************************************************)

Modeling the class Quad

(* Modeling the class "Quad" *)

(* Modeling the method Rect_A_CONST of the class Quad *)
Let Rect_A_CONST_Quad
: nat->nat->nat->nat->Omega->Omega
let w1 := w in let Assign1_Rect_A_CONST := fun (av1 : Omega) <=>
( mk_Omega av1.(label_rect_a)
  av1.(label_rect_b)
  av1.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A a1
                  av1.(label_quad).(y1)
                  av1.(label_quad).(x2)
                  av1.(label_quad).(y2)
                  av1.(label_quad).(x3)
                  av1.(label_quad).(y3)
                  av1.(label_quad).(x4)
                  av1.(label_quad).(y4)
                  av1.(label_quad).(tri_1)
                  av1.(label_quad).(tri_2)
    )
    av1.(label_quad).(area) )
  )
in let w2 := Assign1_Rect_A_CONST w1
in let Assign2_Rect_A_CONST := fun (av2 : Omega) <=>
( mk_Omega av2.(label_rect_a)
  av2.(label_rect_b)
  av2.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A av2.(label_quad).(x1)
                  b1
                  av2.(label_quad).(x2)
                  av2.(label_quad).(y2) )
                av2.(label_quad).(width)
                av2.(label_quad).(height)
                av2.(label_quad).(area) )
  )
  av2.(label_quad).(x3)
  av2.(label_quad).(y3)
  av2.(label_quad).(x4)
  av2.(label_quad).(y4)
aw2.(label_quad).(tri_1)
aw2.(label_quad).(tri_2)

aw2.(label_lineartrans) 

in let w3 := Assign2_Rect_A_CONST w2

in let Assign3_Rect_A_CONST := fun (aw3 : Omega) =>

( mk_Omega aw3.(label_rect_a)
aw3.(label_rect_b)
aw3.(label_rect_c)

( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw3.(label_quad).(x1)
aw3.(label_quad).(y1)
a2
aw3.(label_quad).(y2) )

aw3.(label_quad).(width)
aw3.(label_quad).(height)
aw3.(label_quad).(area) )

aw3.(label_quad).(x3)
aw3.(label_quad).(y3)
aw3.(label_quad).(x4)
aw3.(label_quad).(y4)
aw3.(label_quad).(tri_1)
aw3.(label_quad).(tri_2) )

aw3.(label_lineartrans) 

in let w4 := Assign3_Rect_A_CONST w3

in let Assign4_Rect_A_CONST := fun (aw4 : Omega) =>

( mk_Omega aw4.(label_rect_a)
aw4.(label_rect_b)
aw4.(label_rect_c)

( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw4.(label_quad).(x1)
aw4.(label_quad).(y1)
aw4.(label_quad).(x2)
b2 )

aw4.(label_quad).(width)
aw4.(label_quad).(height)
aw4.(label_quad).(area) )

aw4.(label_quad).(x3)
aw4.(label_quad).(y3)
aw4.(label_quad).(x4)
aw4.(label_quad).(y4)
aw4.(label_quad).(tri_1)
aw4.(label_quad).(tri_2) )

aw4.(label_lineartrans) 

in let w5 := Assign4_Rect_A CONST w4

in w5.

(* Modeling the method Rect_C_CONST of the class Quad *)

Let Rect_C_CONST_Quad
: Omega->Omega:= fun(w : Omega) =>

let w1 := w

in let Assign1_Rect_C_CONST := fun (aw1 : Omega) =>

( mk_Omega aw1.(label_rect_a)
aw1.(label_rect_b)
aw1.(label_rect_c)

( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw1.(label_quad).(x1)
aw1.(label_quad).(y1) 

aw1.(label_quad).(x2)
aw1.(label_quad).(y2) )

aw1.(label_quad).(width)
aw1.(label_quad).(height)
aw1.(label_quad).(area) )

aw1.(label_quad).(x3)
aw1.(label_quad).(y3)
aw1.(label_quad).(x4)
aw1.(label_quad).(y4)
aw1.(label_quad).(tri_1)
aw1.(label_quad).(tri_2) )

aw1.(label_lineartrans) 

in
APPENDIX A. COQ CODES FOR CASE STUDY

(* Modeling the method GetArea of the class Quad *)

in let w2 := Assign1_Rect_CCONST w1
in let Assign2_Rect_CCONST := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_rect_a)
  aw2.(label_rect_b)
  aw2.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw2.(label_quad).(x1)
  aw2.(label_quad).(y1)
  aw2.(label_quad).(x2)
  aw2.(label_quad).(y2))
  aw2.(label_quad).(width)
  0
  aw2.(label_quad).(area) )
  aw2.(label_quad).(x3)
  aw2.(label_quad).(y3)
  aw2.(label_quad).(x4)
  aw2.(label_quad).(y4)
  aw2.(label_quad).(tri_1)
  aw2.(label_quad).(tri_2) )
  aw2.(label_lineartrans) )

in let w3 := Assign2_Rect_CCONST w2
in let Assign3_Rect_CCONST := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a)
  aw3.(label_rect_b)
  aw3.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw3.(label_quad).(x1)
  aw3.(label_quad).(y1)
  aw3.(label_quad).(x2)
  aw3.(label_quad).(y2))
  aw3.(label_quad).(width)
  aw3.(label_quad).(height)
  0 )
  aw3.(label_quad).(x3)
  aw3.(label_quad).(y3)
  aw3.(label_quad).(x4)
  aw3.(label_quad).(y4)
  aw3.(label_quad).(tri_1)
  aw3.(label_quad).(tri_2) )
  aw3.(label_lineartrans) )

in let w4 := Assign3_Rect_CCONST w3
in w4.
Let GetArea_Quad
  : Omega-> nat := fun (w : Omega) => w.(label_quad).(area).

(* Modeling the method FindWidth of the class Quad *)
Let FindWidth_Quad
  : Omega-> Omega := fun (w : Omega) =>
let w1 := w
in let If1_FindWidth
    := fun (iw1:Omega) =>
if ( LessThan iw1.(label_rect_c).(x1) iw1.(label_rect_c).(x2))
then let Assign1_FindWidth := fun (aw1 : Omega) =>
    ( mk_Omega aw1.(label_rect_a)
      aw1.(label_rect_b)
      aw1.(label_rect_c)
      (mk_SQuad (mk_SRect_C (mk_SRect_A aw1.(label_quad).(x1)
                    aw1.(label_quad).(y1)
                    aw1.(label_quad).(x2)
                    aw1.(label_quad).(y2)))
      (aw1.(label_rect_c).(x2) - aw1.(label_rect_c).(x1))
      aw1.(label_quad).(x3)
      aw1.(label_quad).(x4)
      aw1.(label_quad).(y3)
      aw1.(label_quad).(y4)
      aw1.(label_quad).(tri_1)
      aw1.(label_quad).(tri_2) )
    aw1.(label_lineartrans) )
  in Assign1_FindWidth iw1
else let Assign2_FindWidth := fun (aw2 : Omega) =>
    ( mk_Omega aw2.(label_rect_a)
      aw2.(label_rect_b)
      aw2.(label_rect_c)
      (mk_SQuad (mk_SRect_C (mk_SRect_A aw2.(label_quad).(x1)
                    aw2.(label_quad).(y1)
                    aw2.(label_quad).(x2)
                    aw2.(label_quad).(y2)))
      (aw2.(label_rect_c).(x1) - aw2.(label_rect_c).(x2))
      aw2.(label_quad).(x3)
      aw2.(label_quad).(x4)
      aw2.(label_quad).(y3)
      aw2.(label_quad).(y4)
      aw2.(label_quad).(tri_1)
      aw2.(label_quad).(tri_2) )
    aw2.(label_lineartrans) )
  in Assign2_FindWidth iw1
in let w2 := If1_FindWidth w1
in w2.

(* Modeling the method FindHeight of the class Quad *)
Let FindHeight_Quad
  : Omega-> Omega := fun (w : Omega) =>
let \( w_1 \) := \( w \)

in let If1_FindHeight := fun (iw1:\text{Omega}) =>

if ( LessThan iw1.(label_rect_c).(y1) iw1.(label_rect_c).(y2))

then let Assign1_FindHeight := fun (aw1 : \text{Omega}) =>

( \text{mk}_\text{Omega} aw1.(label_rect_a)
aw1.(label_rect_b)
aw1.(label_rect_c)
( \text{mk}_\text{SQuad} ( \text{mk}_\text{SRect_C} ( \text{mk}_\text{SRect_A} aw1.(label_quad).(x1)
aw1.(label_quad).(y1)
aw1.(label_quad).(x2)
aw1.(label_quad).(y2) )
aw1.(label_quad).(width)
(aw1.(label_rect_c).(y2) - aw1.(label_rect_c).(y1))
aw1.(label_quad).(area) )

aw1.(label_quad).(x3)
aw1.(label_quad).(y3)
aw1.(label_quad).(x4)
aw1.(label_quad).(y4)
aw1.(label_quad).(tri_1)
aw1.(label_quad).(tri_2) )

aw1.(label_lineartrans) )

in Assign1_FindHeight iw1 in

else let Assign2_FindHeight := fun (aw2 : \text{Omega}) =>

( \text{mk}_\text{Omega} aw2.(label_rect_a)
aw2.(label_rect_b)
aw2.(label_rect_c)
( \text{mk}_\text{SQuad} ( \text{mk}_\text{SRect_C} ( \text{mk}_\text{SRect_A} aw2.(label_quad).(x1)
aw2.(label_quad).(y1)
aw2.(label_quad).(x2)
aw2.(label_quad).(y2) )
aw2.(label_quad).(width)
(aw2.(label_rect_c).(y1) - aw2.(label_rect_c).(y2))
aw2.(label_quad).(area) )

aw2.(label_quad).(x3)
aw2.(label_quad).(y3)
aw2.(label_quad).(x4)
aw2.(label_quad).(y4)
aw2.(label_quad).(tri_1)
aw2.(label_quad).(tri_2) )

aw2.(label_lineartrans) )

in Assign2_FindHeight iw1 in

let \( w_2 \) := If1_FindHeight \( w_1 \)

in \( w_2 \).

(* Modeling the method Quad\_CONST of the class Quad *)

Let Quad\_CONST_: nat\rightarrow nat\rightarrow nat\rightarrow nat\rightarrow \text{Omega}\rightarrow\text{Omega}

let \( w_1 \) := \( w \)

in let Assign1_Quad\_CONST := fun (aw1 : \text{Omega}) =>

( \text{mk}_\text{Omega} aw1.(label_rect_a)
aw1.(label_rect_b)
aw1.(label_rect_c)
( \text{mk}_\text{SQuad} ( \text{mk}_\text{SRect_C} ( \text{mk}_\text{SRect_A} aw1.(label_quad).(x1)
aw1.(label_quad).(y1)
aw1.(label_quad).(x2)
aw1.(label_quad).(y2) )
aw1.(label_quad).(width)
(aw1.(label_rect_c).(y1) - aw1.(label_rect_c).(y2))
aw1.(label_quad).(area) )

aw1.(label_quad).(x3)
aw1.(label_quad).(y3)
aw1.(label_quad).(x4)
aw1.(label_quad).(y4)
aw1.(label_quad).(tri_1)
aw1.(label_quad).(tri_2) )

aw1.(label_lineartrans) )

in Assign1_Quad\_CONST \( w_1 \) in

let \( w_2 \) := If1_FindHeight \( w_1 \)

in \( w_2 \).
aw1.(label_quad).(y2)  
aw1.(label_quad).(width)  
aw1.(label_quad).(height)  
aw1.(label_quad).(area)  
c1  
aw1.(label_quad).(y3)  
aw1.(label_quad).(x4)  
aw1.(label_quad).(y4)  
aw1.(label_quad).(tri_1)  
aw1.(label_quad).(tri_2)  
aw1.(label_lineartrans)  

in let w2 := Assign1_Quad_CONST w1  
in let Assign2_Quad_CONST := fun (aw2 : Omega) =>  
  ( mk_Omega aw2.(label_rect_a)  
    aw2.(label_rect_b)  
    aw2.(label_rect_c)  
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw2.(label_quad).(x1)  
                      aw2.(label_quad).(y1)  
                      aw2.(label_quad).(x2)  
                      aw2.(label_quad).(y2)  
                      aw2.(label_quad).(x3)  
                      aw2.(label_quad).(x4)  
                      aw2.(label_quad).(y4)  
                      aw2.(label_quad).(tri_1)  
                      aw2.(label_quad).(tri_2)  
                  )  
            )  
  )  
aw2.(label_lineartrans)  

in let w3 := Assign2_Quad_CONST w2  
in let Assign3_Quad_CONST := fun (aw3 : Omega) =>  
  ( mk_Omega aw3.(label_rect_a)  
    aw3.(label_rect_b)  
    aw3.(label_rect_c)  
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw3.(label_quad).(x1)  
                      aw3.(label_quad).(y1)  
                      aw3.(label_quad).(x2)  
                      aw3.(label_quad).(y2)  
                      aw3.(label_quad).(x3)  
                      aw3.(label_quad).(y3)  
                      c2  
                      aw3.(label_quad).(y4)  
                      aw3.(label_quad).(tri_1)  
                      aw3.(label_quad).(tri_2)  
                  )  
            )  
aw3.(label_lineartrans)  

in let w4 := Assign3_Quad_CONST w3  
in let Assign4_Quad_CONST := fun (aw4 : Omega) =>  
  ( mk_Omega aw4.(label_rect_a)  
    aw4.(label_rect_b)  
    aw4.(label_rect_c)
APPENDIX A. COQ CODES FOR CASE STUDY

( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw4.(label_quad).(x1)
aw4.(label_quad).(y1)
aw4.(label_quad).(x2)
aw4.(label_quad).(y2) )
aw4.(label_quad).(width)
aw4.(label_quad).(height)
aw4.(label_quad).(area) )
aw4.(label_quad).(x3)
aw4.(label_quad).(y3)
aw4.(label_quad).(x4)
d2
aw4.(label_quad).(tri_1)
aw4.(label_quad).(tri_2) )
aw4.(label_lineartrans) )
in let w5 := Assign4_Quad_CONST w4
in let Assign5_Quad_CONST := fun (aw5 : Omega) =>
( mk_Omega aw5.(label_rect_a)
aw5.(label_rect_b)
aw5.(label_rect_c)
( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw5.(label_quad).(x1)
aw5.(label_quad).(y1)
aw5.(label_quad).(x2)
aw5.(label_quad).(y2) )
aw5.(label_quad).(width)
aw5.(label_quad).(height)
aw5.(label_quad).(area) )
aw5.(label_quad).(x3)
aw5.(label_quad).(y3)
aw5.(label_quad).(x4)
aw5.(label_quad).(y4)
0
aw5.(label_quad).(tri_2) )
aw5.(label_lineartrans) )
in let w6 := Assign5_Quad_CONST w5
in let Assign6_Quad_CONST := fun (aw6 : Omega) =>
( mk_Omega aw6.(label_rect_a)
aw6.(label_rect_b)
aw6.(label_rect_c)
( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw6.(label_quad).(x1)
aw6.(label_quad).(y1)
aw6.(label_quad).(x2)
aw6.(label_quad).(y2) )
aw6.(label_quad).(width)
aw6.(label_quad).(height)
aw6.(label_quad).(area) )
aw6.(label_quad).(x3)
aw6.(label_quad).(y3)
aw6.(label_quad).(x4)
aw6.(label_quad).(y4)
aw6.(label_quad).(tri_1)
0 )
aw6.(label_lineartrans) )
in let w7 := Assign6_Quad_CONST w6
in w7.
APPENDIX A. COQ CODES FOR CASE STUDY

(* Modeling the method TriArea2 of the class Quad *)
Let TriArea2_Quad : nat->nat->nat->nat->nat->nat->Omega->nat :=
let beforeEw := mk_Omega_TriArea2_Quad w 0
in let Assign1_TriArea2 := fun (aw1 : Omega_TriArea2_Quad) =>
(mk_Omega_TriArea2_Quad
(mk_Omega aw1.(label_rect_a)
aw1.(label_rect_b)
aw1.(label_rect_c)
(mk_SQuad (mk_SRect_C (mk_SRect_A aw1.(label_quad).(x1)
aw1.(label_quad).(y1)
aw1.(label_quad).(x2)
aw1.(label_quad).(y2)
aw1.(label_quad).(width)
aw1.(label_quad).(height)
aw1.(label_quad).(area)

aw1.(label_quad).(x3)
aw1.(label_quad).(y3)
aw1.(label_quad).(x4)
aw1.(label_quad).(y4)
aw1.(label_quad).(tri_1)
aw1.(label_quad).(tri_2)
aw1.(label_lineartrans)
)(((p2 - p1) * (q3 - q1)) - ((p3 - p1) * (q2 - q1))))
in let afterEw := (Assign1_TriArea2 beforeEw)
in afterEw.(doubleArea_local).

(* Modeling the method FindArea of the class Quad *)
Let FindArea_Quad (TriArea2 : nat->nat->nat->nat->nat->nat->Omega->nat) : Omega->Omega:=
fun (w : Omega) =>
let w1 := w
in let Assign1_FindArea := fun (aw1 :Omega) =>
(mk_Omega aw1.(label_rect_a)
aw1.(label_rect_b)
aw1.(label_rect_c)
(mk_SQuad (mk_SRect_C (mk_SRect_A aw1.(label_quad).(x1)
aw1.(label_quad).(y1)
aw1.(label_quad).(x2)
aw1.(label_quad).(y2)
aw1.(label_quad).(width)
aw1.(label_quad).(height)
aw1.(label_quad).(area)

aw1.(label_quad).(x3)
aw1.(label_quad).(y3)
aw1.(label_quad).(x4)
aw1.(label_quad).(y4)
aw1.(label_quad).(tri_1)
aw1.(label_quad).(tri_2)

aw1.(label_lineartrans)
)(((p2 - p1) * (q3 - q1)) - ((p3 - p1) * (q2 - q1))))
in let afterEw := (Assign1_TriArea2 beforeEw)
in afterEw.(doubleArea_local).
APPENDIX A. COQ CODES FOR CASE STUDY

```coq
let w2 := Assign1_FindArea w1
in let Assign2_FindArea := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_rect_a)
    aw2.(label_rect_b)
    aw2.(label_rect_c)
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw2.(label_quad).(x1)
         aw2.(label_quad).(y1)
         aw2.(label_quad).(x2)
         aw2.(label_quad).(y2)
         aw2.(label_quad).(width)
         aw2.(label_quad).(height)
         aw2.(label_quad).(area)
         )
      aw2.(label_quad).(x3)
      aw2.(label_quad).(y3)
      aw2.(label_quad).(x4)
      aw2.(label_quad).(y4)
      aw2.(label_quad).(tri_1)
      ( TriArea2 aw2.(label_quad).(x3)
         aw2.(label_quad).(y3)
         aw2.(label_quad).(x4)
         aw2.(label_quad).(y4)
         aw2.(label_quad).(x1)
         aw2.(label_quad).(y1)
         aw2)
    )
  )

let w3 := Assign2_FindArea w2
in let Assign3_FindArea := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a)
    aw3.(label_rect_b)
    aw3.(label_rect_c)
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw3.(label_quad).(x1)
         aw3.(label_quad).(y1)
         aw3.(label_quad).(x2)
         aw3.(label_quad).(y2)
         aw3.(label_quad).(width)
         aw3.(label_quad).(height)
         (aw3.(label_quad).(tri_1) + aw3.(label_quad).(tri_2))
      aw3.(label_quad).(x3)
      aw3.(label_quad).(y3)
      aw3.(label_quad).(x4)
      aw3.(label_quad).(y4)
      aw3.(label_quad).(tri_1)
      aw3.(label_quad).(tri_2)
      )
  )

let w4 := Assign3_FindArea w3
in w4.

(* Modeling the method Enlarge of the class Quad *)

Let Enlarge_Quad
: nat-> Omega->Omega := fun(n : nat)(w : Omega) =>
```
let w1 := w
in let Assign1_Enlarge := fun (aw1 : Omega) =>
  ( mk_Omega aw1.(label_rect_a)
  aw1.(label_rect_b)
  aw1.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A (aw1.(label_quad).(x1) * n)
    aw1.(label_quad).(y1)
    aw1.(label_quad).(x2)
    aw1.(label_quad).(y2) )
    aw1.(label_quad).(width)
    aw1.(label_quad).(height)
    aw1.(label_quad).(area) )
  aw1.(label_quad).(x3)
  aw1.(label_quad).(y3)
  aw1.(label_quad).(x4)
  aw1.(label_quad).(y4)
  aw1.(label_quad).(tri_1)
  aw1.(label_quad).(tri_2) )
  aw1.(label_lineartrans) )
in let w2 := Assign1_Enlarge w1
in let Assign2_Enlarge := fun (aw2 : Omega) =>
  ( mk_Omega aw2.(label_rect_a)
  aw2.(label_rect_b)
  aw2.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw2.(label_quad).(x1)
    (aw2.(label_quad).(y1) * n)
    aw2.(label_quad).(x2)
    aw2.(label_quad).(y2) )
    aw2.(label_quad).(width)
    aw2.(label_quad).(height)
    aw2.(label_quad).(area) )
  aw2.(label_quad).(x3)
  aw2.(label_quad).(y3)
  aw2.(label_quad).(x4)
  aw2.(label_quad).(y4)
  aw2.(label_quad).(tri_1)
  aw2.(label_quad).(tri_2) )
  aw2.(label_lineartrans) )
in let w3 := Assign2_Enlarge w2
in let Assign3_Enlarge := fun (aw3 : Omega) =>
  ( mk_Omega aw3.(label_rect_a)
  aw3.(label_rect_b)
  aw3.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw3.(label_quad).(x1)
    aw3.(label_quad).(y1)
    (aw3.(label_quad).(x2) * n)
    aw3.(label_quad).(y2) )
    aw3.(label_quad).(width)
    aw3.(label_quad).(height)
    aw3.(label_quad).(area) )
  aw3.(label_quad).(x3)
  aw3.(label_quad).(y3)
  aw3.(label_quad).(x4)
  aw3.(label_quad).(y4)
aw3.(label_quad).(tri_1)
aw3.(label_quad).(tri_2)

in let w4 := Assign3_Enlarge w3
in let Assign4_Enlarge := fun (aw4 : Omega) =>
  ( mk_Omega aw4.(label_rect_a)
  aw4.(label_rect_b)
  aw4.(label_rect_c)
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw4.(label_quad).(x1)
    aw4.(label_quad).(y1)
    aw4.(label_quad).(x2)
    (aw4.(label_quad).(y2) * n) )
    aw4.(label_quad).(width)
    aw4.(label_quad).(height)
    aw4.(label_quad).(area) )
  aw4.(label_quad).(x3)
  aw4.(label_quad).(y3)
  aw4.(label_quad).(x4)
  aw4.(label_quad).(y4)
  aw4.(label_quad).(tri_1)
  aw4.(label_quad).(tri_2)
  )
aw4.(label_lineartrans)

in let w5 := Assign4_Enlarge w4
in let Assign5_Enlarge := fun (aw5 : Omega) =>
  ( mk_Omega aw5.(label_rect_a)
  aw5.(label_rect_b)
  aw5.(label_rect_c)
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw5.(label_quad).(x1)
    aw5.(label_quad).(y1)
    aw5.(label_quad).(x2)
    aw5.(label_quad).(y2) )
    aw5.(label_quad).(width)
    aw5.(label_quad).(height)
    aw5.(label_quad).(area) )
  (aw5.(label_quad).(x3) * n)
  aw5.(label_quad).(y3)
  aw5.(label_quad).(x4)
  aw5.(label_quad).(y4)
  aw5.(label_quad).(tri_1)
  aw5.(label_quad).(tri_2)
  )
aw5.(label_lineartrans)

in let w6 := Assign5_Enlarge w5
in let Assign6_Enlarge := fun (aw6 : Omega) =>
  ( mk_Omega aw6.(label_rect_a)
  aw6.(label_rect_b)
  aw6.(label_rect_c)
    ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aw6.(label_quad).(x1)
    aw6.(label_quad).(y1)
    aw6.(label_quad).(x2)
    aw6.(label_quad).(y2) )
    aw6.(label_quad).(width)
    aw6.(label_quad).(height)
    aw6.(label_quad).(area) )
  aw6.(label_quad).(x3)
APPENDIX A. COQ CODES FOR CASE STUDY

(aws6.(label_quad).(y3) * n)
aws6.(label_quad).(x4)
aws6.(label_quad).(y4)
aws6.(label_quad).(tri_1)
aws6.(label_quad).(tri_2)
aws6.(label_quad).in

in let w7 := Assign6_Enlarge w6
in let Assign7_Enlarge := fun (aws7 : Omega) =>
  ( mk_Omega aws7.(label_rect_a)
aws7.(label_rect_b)
aws7.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aws7.(label_quad).(x1)
aws7.(label_quad).(y1)
aws7.(label_quad).(x2)
aws7.(label_quad).(y2) )
aws7.(label_quad).(width)
aws7.(label_quad).(height)
aws7.(label_quad).(area) )
aws7.(label_quad).(x3)
aws7.(label_quad).(y3)
(aws7.(label_quad).(x4) * n)
aws7.(label_quad).(y4)
aws7.(label_quad).(tri_1)
aws7.(label_quad).(tri_2)
aws7.(label_quad).in

in let w8 := Assign7_Enlarge w7
in let Assign8_Enlarge := fun (aws8 : Omega) =>
  ( mk_Omega aws8.(label_rect_a)
aws8.(label_rect_b)
aws8.(label_rect_c)
  ( mk_SQuad ( mk_SRect_C ( mk_SRect_A aws8.(label_quad).(x1)
aws8.(label_quad).(y1)
aws8.(label_quad).(x2)
aws8.(label_quad).(y2) )
aws8.(label_quad).(width)
aws8.(label_quad).(height)
aws8.(label_quad).(area) )
aws8.(label_quad).(x3)
aws8.(label_quad).(y3)
aws8.(label_quad).(x4)
(aws8.(label_quad).(y4) * n)
aws8.(label_quad).(tri_1)
aws8.(label_quad).(tri_2)
aws8.(label_quad).in

in let w9 := Assign8_Enlarge w8
in w9.

(* Define a record type representing all methods of the class Quad *)
Record MQuad : Set := mk_MQuad { 
  label_Rect_A_CONST_Quad : Unit Rect_A_CONST_Quad ;
  label_Rect_C_CONST_Quad : Unit Rect_C_CONST_Quad ;
  label_GetArea_Quad : Unit GetArea_Quad ;
  label_FindWidth_Quad : Unit FindWidth_Quad ;
  label_FindHeight_Quad : Unit FindHeight_Quad ;
}
APPENDIX A. COQ CODES FOR CASE STUDY

```coq
label_Quad_CONST_Quad : Unit Quad_CONST_Quad ;
label_TriArea2_Quad : Unit TriArea2_Quad ;
label_FindArea_Quad :
  Unit (FindArea_Quad(label_TriArea2_Quad :
    ID(nat->nat->nat->nat->nat->nat->Omega->nat) )) ;
label_Enlarge_Quad : Unit Enlarge_Quad }.

(* Model the class Quad as a record type *)
Record Quad : Set := mk_Quad {
  label_squad : SQuad ;
  label_mquad :> MQuad }.

(* Model the interface-type Interface_Quad as a record type *)
Record Interface_Quad : Set := mk_Interface_Quad {
  i_label_Rect_A_CONST_Quad : nat->nat->nat->nat->Omega->Omega ;
  i_label_Rect_C_CONST_Quad : Omega->Omega ;
  i_label_GetArea_Quad : Omega->nat ;
  i_label_FindWidth_Quad : Omega->Omega ;
  i_label_FindHeight_Quad : Omega->Omega ;
  i_label_Quad_CONST_Quad : nat->nat->nat->nat->Omega->Omega ;
  i_label_TriArea2_Quad : nat->nat->nat->nat->nat->nat->Omega->nat ;
  i_label_FindArea_Quad : Omega->Omega ;
  i_label_Enlarge_Quad : nat->Omega->Omega }.

(* Define a coercion between MQuad and Interface_Quad *)
Coercion MQuad_Interface_Quad (x : MQuad) : Interface_Quad := mk_Interface_Quad
  (label_Rect_A_CONST_Quad x : ID (nat->nat->nat->nat->Omega->Omega) )
  (label_Rect_C_CONST_Quad x : ID (Omega->Omega) )
  (label_GetArea_Quad x : ID (Omega->nat) )
  (label_FindWidth_Quad x : ID (Omega->Omega) )
  (label_FindHeight_Quad x : ID (Omega->Omega) )
  (label_Quad_CONST_Quad x : ID (nat->nat->nat->nat->Omega->Omega) )
  (label_TriArea2_Quad x : ID (nat->nat->nat->nat->nat->nat->Omega->nat) )
  (label_FindArea_Quad x : ID (Omega->Omega) )
  (label_Enlarge_Quad x : ID (nat->Omega->Omega) ) .

(* Define a coercion between Interface_Quad and Interface_Rect_C *)
Coercion Interface_Quad_Interface_Rect_C (x : Interface_Quad) : Interface_Rect_C :=
  mk_Interface_Rect_C
  x.(i_label_Rect_A_CONST_Quad)
  x.(i_label_Rect_C_CONST_Quad)
  x.(i_label_GetArea_Quad)
  x.(i_label_FindWidth_Quad)
  x.(i_label_FindHeight_Quad)
  x.(i_label_FindArea_Quad)
  x.(i_label_Enlarge_Quad) .

(* Define Omega extended with the interface-type Interface_Quad *)
Record Omega_Interface_Quad : Set := mk_Omega_Interface_Quad {
  label_omega_interface_quad :> Omega ;
  label_interface_quad : Interface_Quad }.

(* Define a coercion between Omega_Interface_Rect_C and Omega_Interface_Quad *)
Coercion Omega_Interface_Quad_TO_Omega_Interface_Rect_C (x : Omega_Interface_Quad) :
  Omega_Interface_Rect_C :=
  mk_Omega_Interface_Rect_C
```

Theorems generated from the class \textit{Quad}

**Theorem Invariant\_FindWidth\_Quad**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}), \, \text{True} \rightarrow ((s.(\text{label}\_\text{quad}).(\text{tri}\_1)) \leq (s.(\text{label}\_\text{quad}).(\text{area})) \land (s.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s.(\text{label}\_\text{quad}).(\text{area}))) \rightarrow \)
\begin{align*}
\text{let } s' & := (r.(\text{label}\_\text{FindWidth}\_\text{Quad}) : \text{ID (Omega}\to\text{Omega})) \, s \\
in ((s'.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})) \land (s'.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s'.(\text{label}\_\text{quad}).(\text{area}))).
\end{align*}

**Theorem Invariant\_FindHeight\_Quad**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}), \, \text{True} \rightarrow ((s.(\text{label}\_\text{quad}).(\text{tri}\_1)) \leq (s.(\text{label}\_\text{quad}).(\text{area})) \land (s.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s.(\text{label}\_\text{quad}).(\text{area}))) \rightarrow \)
\begin{align*}
\text{let } s' & := (r.(\text{label}\_\text{FindHeight}\_\text{Quad}) : \text{ID (Omega}\to\text{Omega})) \, s \\
in ((s'.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})) \land (s'.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s'.(\text{label}\_\text{quad}).(\text{area}))).
\end{align*}

**Theorem Invariant\_FindArea\_Quad**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}), \, \text{True} \rightarrow ((s.(\text{label}\_\text{quad}).(\text{tri}\_1)) \leq (s.(\text{label}\_\text{quad}).(\text{area})) \land (s.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s.(\text{label}\_\text{quad}).(\text{area}))) \rightarrow \)
\begin{align*}
\text{let } s' & := (r.(\text{label}\_\text{FindArea}\_\text{Quad}) : \text{ID (Omega}\to\text{Omega})) \, s \\
in ((s'.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})) \land (s'.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s'.(\text{label}\_\text{quad}).(\text{area}))).
\end{align*}

**Theorem Invariant\_Enlarge\_Quad**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}) \, (n : \text{nat}), \, \text{True} \rightarrow ((s.(\text{label}\_\text{quad}).(\text{tri}\_1)) \leq (s.(\text{label}\_\text{quad}).(\text{area})) \land (s.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s.(\text{label}\_\text{quad}).(\text{area}))) \rightarrow \)
\begin{align*}
\text{let } s' & := (r.(\text{label}\_\text{Enlarge}\_\text{Quad}) : \text{ID (nat}\to\text{Omega}\to\text{Omega})) \, n \, s \\
in ((s'.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})) \land (s'.(\text{label}\_\text{quad}).(\text{tri}\_2)) \leq (s'.(\text{label}\_\text{quad}).(\text{area}))).
\end{align*}

**Proof of the above theorems**

**Lemma Quad\_width1**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}), \,
\begin{align*}
(s.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s.(\text{label}\_\text{quad}).(\text{area})) \rightarrow \\
\text{let } s' & := (r.(\text{label}\_\text{FindWidth}\_\text{Quad}) : \text{ID (Omega}\to\text{Omega})) \, s \\
in (s'.(\text{label}\_\text{quad}).(\text{tri}\_1)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})).
\end{align*}
\)
Admitted.

**Lemma Quad\_width2**: \(\forall (r : \text{Quad}) \, (s : \text{Omega}), \,
\begin{align*}
(s.(\text{label}\_\text{quad}).(\text{tri}\_2)) & \leq (s.(\text{label}\_\text{quad}).(\text{area})) \rightarrow \\
\text{let } s' & := (r.(\text{label}\_\text{FindWidth}\_\text{Quad}) : \text{ID (Omega}\to\text{Omega})) \, s \\
in (s'.(\text{label}\_\text{quad}).(\text{tri}\_2)) & \leq (s'.(\text{label}\_\text{quad}).(\text{area})).
\end{align*}
\)
Admitted.

\begin{align*}
\text{Proof of the above theorems} & \rightarrow \\
\text{Lemma Quad\_width1 : forall (r : Quad) (s : Omega), } & \\
\text{( s.(label_quad).(tri_1) ) <= ( s.(label_quad).(area) ) } & \rightarrow \\
\text{let s' := (r.(label_FindWidth_Quad) : ID (Omega->Omega)) } & \, s \\
\text{in ( s'.(label_quad).(tri_1) ) <= ( s'.(label_quad).(area) ).} & \\
\text{Admitted.} & \\
\text{Lemma Quad\_width2 : forall (r : Quad) (s : Omega), } & \\
\text{( s.(label_quad).(tri_2) ) <= ( s.(label_quad).(area) ) } & \rightarrow \\
\text{let s' := (r.(label_FindWidth_Quad) : ID (Omega->Omega)) } & \, s \\
\text{in ( s'.(label_quad).(tri_2) ) <= ( s'.(label_quad).(area) ).} & \\
\text{Admitted.} & \\
\text{Proof of the above theorems} & \rightarrow
\end{align*}

\begin{align*}
\text{Theorem Invariant\_FindWidth\_Quad : forall (r : Quad) (s : Omega), } & \text{True } \rightarrow \\
\text{( s.(label_quad).(tri_1) ) <= ( s.(label_quad).(area) ) } & \land \\
\text{( s.(label_quad).(tri_2) ) <= ( s.(label_quad).(area) ) } & \rightarrow
\end{align*}
APPENDIX A. COQ CODES FOR CASE STUDY

136

let s' := (r.(label_FindWidth_Quad) : ID (Omega->Omega) ) s
in (s'.(label_quad ).(tri_1)) <= (s'.(label_quad ).(area)) /
(s'.(label_quad ).(tri_2)) <= (s'.(label_quad ).(area)).

Proof.
intros r s.
intros H1.
intros H2.
inversion H2 as [H3 H4].
simpl.
apply conj.
apply (Quad_width1 r s).
effect H3.
apply (Quad_width2 r s).
effect H4.
Qed.

*******************************************************************************************

Lemma Quad_height1 : forall (r : Quad) (s : Omega) ,
(s.(label_quad ).(tri_1)) <= (s.(label_quad ).(area)) ->
let s' := (r.(label_FindHeight_Quad) : ID (Omega->Omega) ) s
in (s'.(label_quad ).(tri_1)) <= (s'.(label_quad ).(area)).
Admitted.

Lemma Quad_height2 : forall (r : Quad) (s : Omega) ,
(s.(label_quad ).(tri_2)) <= (s.(label_quad ).(area)) ->
let s' := (r.(label_FindHeight_Quad) : ID (Omega->Omega) ) s
in (s'.(label_quad ).(tri_2)) <= (s'.(label_quad ).(area)).
Admitted.

*******************************************************************************************

Theorem Invariant_FindHeight_Quad : forall (r : Quad) (s : Omega), True ->
((s.(label_quad ).(tri_1)) <= (s.(label_quad ).(area)) /
(s.(label_quad ).(tri_2)) <= (s.(label_quad ).(area)) ->
let s' := (r.(label_FindHeight_Quad) : ID (Omega->Omega) ) s
in (s'.(label_quad ).(tri_1)) <= (s'.(label_quad ).(area)) /
(s'.(label_quad ).(tri_2)) <= (s'.(label_quad ).(area)).

Proof.
intros r s.
intros H1.
intros H2.
inversion H2 as [H3 H4].
simpl.
apply conj.
apply (Quad_height1 r s).
effect H3.
apply (Quad_height2 r s).
effect H4.
Qed.

*******************************************************************************************

Lemma Quad_area1 : forall (r : Quad) (s : Omega) ,
(s.(label_quad ).(tri_1)) <= (s.(label_quad ).(area)) ->
let s' := (r.(label_FindArea_Quad) : ID (Omega->Omega) ) s
in (s'.(label_quad ).(tri_1)) <= (s'.(label_quad ).(area)).
Lemma Quad_area2 : forall (r : Quad) (s : Omega),
(s.(label_quad).(tri_2)) <= (s.(label_quad).(area)) ->
let s' := (r.(label_FindArea_Quad) : ID (Omega->Omega)) s
in (s'.(label_quad).(tri_2)) <= (s'.(label_quad).(area)).
Admitted.

Theorem Invariant_FindArea_Quad : forall (r : Quad) (s : Omega), True ->
((s.(label_quad).(tri_1)) <= (s.(label_quad).(area)) /
(s.(label_quad).(tri_2)) <= (s.(label_quad).(area)) ->
let s' := (r.(label_FindArea_Quad) : ID (Omega->Omega)) s
in ((s'.(label_quad).(tri_1)) <= (s'.(label_quad).(area)) /
(s'.(label_quad).(tri_2)) <= (s'.(label_quad).(area))).
Proof.
intros r s.
intros H1.
intros H2.
inversion H2 as [H3 H4].
apply conj.
apply (Quad_area1 r s).
exact H3.
apply (Quad_area2 r s).
exact H4.
Qed.

Theorem Invariant_Enlarge_Quad : forall (r : Quad) (s : Omega) (n : nat), True ->
((s.(label_quad).(tri_1)) <= (s.(label_quad).(area)) /
(s.(label_quad).(tri_2)) <= (s.(label_quad).(area)) ->
let s' := (r.(label_Enlarge_Quad) : ID (nat->Omega->Omega)) n s
in ((s'.(label_quad).(tri_1)) <= (s'.(label_quad).(area)) /
(s'.(label_quad).(tri_2)) <= (s'.(label_quad).(area))).
Proof.
intros r s n.
intros H1.
intros H2.
inversion H2 as [H3 H4].
simpl.
apply conj.
exact H3.
exact H4.
Qed.

Modeling the class LinearTrans

(* Modeling the class "LinearTrans" *)

(* Modeling the method LinearTrans_CONST of the class LinearTrans *)
Let \( \text{LinearTrans\_CONST\_LinearTrans} : \text{naturals} \rightarrow \text{Omega} \rightarrow \text{Omega} \rightarrow \text{Omega} \rightarrow \text{Omega} \Rightarrow \text{fun}(n : \text{naturals})(w : \text{Omega}) \Rightarrow \)

\[
\begin{align*}
\text{let } w_1 & := w \\
\text{in let } \text{Assign1\_LinearTrans\_CONST} & := \text{fun}(aw_1 : \text{Omega}) \\
& \text{=> (mk\_Omega } aw_1.(\text{label\_rect\_a}) \\
& \text{aw_1.(label\_rect\_b}) \\
& \text{aw_1.(label\_rect\_c)} \\
& \text{aw_1.(label\_quad)} \\
& \text{(mk\_SLinearTrans } 0 \\
& \text{aw_1.(label\_lineartrans).(area\_after)} \\
& \text{aw_1.(label\_lineartrans).(scaleFactor) ) )} \\
\text{in let } w_2 & := \text{Assign1\_LinearTrans\_CONST} w_1 \\
\text{in let } \text{Assign2\_LinearTrans\_CONST} & := \text{fun}(aw_2 : \text{Omega}) \\
& \text{=> (mk\_Omega } aw_2.(\text{label\_rect\_a}) \\
& \text{aw_2.(label\_rect\_b}) \\
& \text{aw_2.(label\_rect\_c)} \\
& \text{aw_2.(label\_quad)} \\
& \text{(mk\_SLinearTrans } aw_2.(\text{label\_lineartrans}.(area\_before)} \\
& \text{0} \\
& \text{aw_2.(label\_lineartrans).(scaleFactor) ) )} \\
\text{in let } w_3 & := \text{Assign2\_LinearTrans\_CONST} w_2 \\
\text{in let } \text{Assign3\_LinearTrans\_CONST} & := \text{fun}(aw_3 : \text{Omega}) \\
& \text{=> (mk\_Omega } aw_3.(\text{label\_rect\_a}) \\
& \text{aw_3.(label\_rect\_b}) \\
& \text{aw_3.(label\_rect\_c)} \\
& \text{aw_3.(label\_quad)} \\
& \text{(mk\_SLinearTrans } aw_3.(\text{label\_lineartrans}.(area\_before)} \\
& \text{aw_3.(label\_lineartrans).(area\_after)} \\
& n ) ) \\
\text{in let } w_4 & := \text{Assign3\_LinearTrans\_CONST} w_3 \\
\text{in } w_4.
\end{align*}
\]

(* Modeling the method CollectingArea of the class LinearTrans *)

Let \( \text{CollectingArea\_LinearTrans} : \text{Interface\_Rect\_C} \rightarrow \text{Omega} \rightarrow \text{Omega} \rightarrow \text{Omega} \rightarrow \text{Omega} \rightarrow \text{fun}(x : \text{Interface\_Rect\_C})(w : \text{Omega}) \Rightarrow \)

\[
\begin{align*}
\text{let } w_1 & := w \\
\text{in let } \text{Assign1\_CollectingArea} & := \text{fun}(aw_1 : \text{Omega}) \\
& \text{=> (mk\_Omega } aw_1.(\text{label\_rect\_a}) \\
& \text{aw_1.(label\_rect\_b}) \\
& \text{aw_1.(label\_rect\_c)} \\
& \text{aw_1.(label\_quad)} \\
& \text{(mk\_SLinearTrans } x.(i\_Get\_Area\_Rect\_C ) aw_1) \\
& \text{aw_1.(label\_lineartrans).(area\_after)} \\
& \text{aw_1.(label\_lineartrans).(scaleFactor) ) )} \\
\text{in let } w_2 & := \text{Assign1\_CollectingArea} w_1 \\
\text{in let } \text{CallStatement1\_CollectingArea} & := \text{fun}(cw_1 : \text{Omega}) \\
& \text{x.(i\_Enlarge\_Rect\_C ) } cw_1.(label\_lineartrans).(scaleFactor) cw_1 \\
\text{in let } w_3 & := \text{CallStatement1\_CollectingArea} w_2 \\
\text{in let } \text{CallStatement2\_CollectingArea} & := \text{fun}(cw_2 : \text{Omega}) \\
& \text{x.(i\_Find\_Area\_Rect\_C ) } cw_2 \\
\text{in let } w_4 & := \text{CallStatement2\_CollectingArea} w_3 \\
\text{in let } \text{Assign2\_CollectingArea} & := \text{fun}(aw_2 : \text{Omega}) \\
& \text{(mk\_Omega } aw_2.(\text{label\_rect\_a}) \\
& \text{aw_2.(label\_rect\_b)}
\end{align*}
\]
APPENDIX A. COQ CODES FOR CASE STUDY

(* Define a record type representing all methods of the class LinearTrans *)
Record MLinearTrans : Set := mk_MLinearTrans {
  label_LinearTrans_CONST_LinearTrans : Unit LinearTrans_CONST_LinearTrans ;
  label_CollectingArea_LinearTrans : Unit CollectingArea_LinearTrans }.

(* Model the class LinearTrans as a record type *)
Record LinearTrans : Set := mk_LinearTrans { label_slineartrans : SLinearTrans ;
  label_mlineartrans :> MLinearTrans }.

Theorem generated from the class LinearTrans

Theorem PrePost_CollectingArea_LinearTrans : forall (r : LinearTrans)(s : Omega) (x : Interface_Rect_C) , ((0) < (s.(label_lineartrans).(areaBefore)) /\ (0) < (s.(label_lineartrans).(areaAfter))) ->
let s' := (r.(label_CollectingArea_LinearTrans) : ID (Interface_Rect_C->Omega->Omega)) x s
in (s'.(label_lineartrans).(areaBefore)) <= (s'.(label_lineartrans).(areaAfter)).

Proof of the above theorem

Lemma CollectingArea_statement1 : forall (s : Omega) (x : Interface_Rect_C),
  (s.(label_lineartrans).(areaBefore)) <= (s.(label_lineartrans).(areaAfter)) ->
let s' := ( mk_Omega s.(label_rect_a)
  s.(label_rect_b)
  s.(label_rect_c)
  s.(label_quad)
  ( mk_SLinearTrans ( x.(i_label_GetArea_Rect_C ) s)
  s.(label_lineartrans).(areaAfter)
  s.(label_lineartrans).(scaleFactor) )
  in (s'.(label_lineartrans).(areaBefore)) <= (s'.(label_lineartrans).(areaAfter)).
Admitted.

Lemma CollectingArea_statement2 : forall (s : Omega)(x : Interface_Rect_C),
  let s' := ( mk_Omega s.(label_rect_a)
  s.(label_rect_b)
  s.(label_rect_c)
  s.(label_quad)
  ( mk_SLinearTrans ( x.(i_label_GetArea_Rect_C ) s)
  s.(label_lineartrans).(areaAfter)
  s.(label_lineartrans).(scaleFactor) )
  in (s'.(label_lineartrans).(areaBefore)) <= (s'.(label_lineartrans).(areaAfter)) ->
(0 < s'.(label_lineartrans).(areaAfter)) ->
let s'' := x.(i_label_Enlarge_Rect_C ) s'.(label_lineartrans).(scaleFactor) s'
in (s''.(label_lineartrans).(areaBefore)) <= (s''.(label_lineartrans).(areaAfter)).
Lemma CollectingArea_statement3 : forall (s : Omega) (x : Interface_Rect_C),
let s' := ( mk_Omega s.(label_rect_a)
s.(label_rect_b)
s.(label_rect_c)
s.(label_quad)
  ( mk_SLinearTrans x.(i_label_GetArea_Rect_C) s)
s.(label_lineartrans).(areaAfter)
s.(label_lineartrans).(scaleFactor) )
in let s'' := x.(i_label_Enlarge_Rect_C) s'.(label_lineartrans).(scaleFactor) s'
in (s''.(label_lineartrans).(areaBefore)) <= (s''.(label_lineartrans).(areaAfter)) ->
let s''' := x.(i_label_FindArea_Rect_C) s''
in (s'''.(label_lineartrans).(areaBefore)) <= (s'''.(label_lineartrans).(areaAfter)).
Admitted.

Lemma CollectingArea_statement4 : forall (s : Omega) (x : Interface_Rect_C),
let s' := ( mk_Omega s.(label_rect_a)
s.(label_rect_b)
s.(label_rect_c)
s.(label_quad)
  ( mk_SLinearTrans x.(i_label_GetArea_Rect_C) s)
s.(label_lineartrans).(areaAfter)
s.(label_lineartrans).(scaleFactor) )
in let s'' := x.(i_label_Enlarge_Rect_C) s'.(label_lineartrans).(scaleFactor) s'
in (s''.(label_lineartrans).(areaBefore)) <= (s''.(label_lineartrans).(areaAfter)) ->
let s''' := ( mk_Omega s'''.(label_rect_a)
  s'''.(label_rect_b)
s'''.(label_rect_c)
s'''.(label_quad)
  ( mk_SLinearTrans s'''.(label_lineartrans).(areaBefore)
    x.(i_label_GetArea_Rect_C) s'''
    s'''.(label_lineartrans).(scaleFactor) ) )
in (s'''.(label_lineartrans).(areaBefore)) <= (s'''.(label_lineartrans).(areaAfter)).
Admitted.

Theorem PrePost_CollectingArea_LinearTrans :
forall (r : LinearTrans)(s : Omega) (x : Interface_Rect_C),
((s.(label_lineartrans).(areaBefore)) <= (s.(label_lineartrans).(areaAfter)) /
(0) < (s.(label_lineartrans).(scaleFactor))) ->
let s' := (r.(Label_CollectingArea_LinearTrans) : ID (Interface_Rect_C->Omega->Omega) ) x s
in (s'.(label_lineartrans).(areaBefore)) <= (s'.(label_lineartrans).(areaAfter)).
Proof.
intros r s x.
intros H1.
inversion H1 as [H2 H3].
apply (CollectingArea_statement1 s x) in H2.
apply (CollectingArea_statement2 s x) in H2.
apply (CollectingArea_statement3 s x) in H2.
apply (CollectingArea_statement4 s x) in H2.
simpl.
auto.
exact H3.
Qed.

(*---------------------------------------------------------------*)
Appendix B

myLibrary.v


Definition UserDefn {T : Type} : T.
Admitted.

Inductive Unit (A:Set) (a:A) :Set := unit : Unit A a.
Implicit Arguments Unit.
Implicit Arguments unit.

Definition ID (A:Set) : Set := A.
Coercion unit_coercion (A:Set)(a:A)(_:Unit a) := a : ID A.

Fixpoint LessThan (m:nat) : nat -> bool :=
  match m with
  | O =>
    fun n:nat => match n with
    | O => false
    | S n' => true
  end
  | S m' =>
    fun n:nat => match n with
    | O => false
    | S n' => LessThan m' n'
  end.
end.

Fixpoint Equal (n m :nat) {struct n} : bool :=
  match n, m with
  | O, O => true
  | O, S _ => false
  | S _, O => false
  | S n1, S m1 => Equal n1 m1
  end.

Definition AND (b1 b2:bool) : bool :=
  match b1, b2 with
| true, true => true  
| true, false => false  
| false, true => false  
| false, false => false | end.

Definition OR (b1 b2:bool) : bool :=  
match b1, b2 with  
| true, true => true  
| true, false => true  
| false, true => true  
| false, false => false | end.

Definition LessEqual (n m:nat) : bool := OR (LessThan n m) (Equal n m).

Definition NOT (b:bool):bool := if b then false else true.

(* The fact that there is no explicit command for moving from one branch of  
a case analysis to the next can make proof scripts rather hard to read.  
Disciplined use of indentation and comments can help, but  
a better way is to use the [Case] tactic. *)

Require String.
Open Scope string_scope.

Ltac move_to_top x :=  
match reverse goal with  
| H : _ |- _ => try move x after H  
end.

Tactic Notation "assert_eq" ident(x) constr(v) :=  
let H := fresh in  
assert (x = v) as H by reflexivity;  
clear H.

Tactic Notation "Case_aux" ident(x) constr(name) :=  
first [  
set (x := name); move_to_top x  
| assert_eq x name; move_to_top x  
| fail 1 "because we are working on a different case" ].

Ltac Case name := Case_aux Case name.
Ltac SCase name := Case_aux SCase name.
Ltac SSCase name := Case_aux SSCase name.
Ltac SSSCase name := Case_aux SSSCase name.
Ltac SSSSSCase name := Case_aux SSSSSCase name.
Ltac SSSSSSSCase name := Case_aux SSSSSSSCase name.
Appendix C

The BNF syntax of myJava in JavaCC

```
Goal ::= ( ClassDecl | InterfaceDecl )* <EOF>
InterfaceDecl ::= <INTERFACE> <IDENTIFIER> ( "extends" <IDENTIFIER> )?
   "(" ( IMethodDecl )* ")"
ClassDecl ::= "class" <IDENTIFIER> ( "extends" <IDENTIFIER> )?
   ( <IMPLEMENTS> <IDENTIFIER> )?
   ( "/*" <INVARIANT> "[[" Expression "]]" "*/" )?
   "(" ( VarDecl )* ( InnerClassDecl )* ( MethodDecl )* ")"
InnerClassDecl ::= "class" <IDENTIFIER> "(" ( VarDecl )* ( MethodDecl )* ")"
VarDecl ::= "protected" Type <IDENTIFIER> ";"
   | Type <IDENTIFIER> ";"
IMethodDecl ::= "public" Type <IDENTIFIER>
   "(" ( Type <IDENTIFIER> ( "," Type <IDENTIFIER> )* ? )? ")" ";"
   | "public" VoidType <IDENTIFIER>
   "(" ( Type <IDENTIFIER> ( "," Type <IDENTIFIER> )* ? )? ")" ";"
MethodDecl ::= "public" Type <IDENTIFIER>
   "(" ( Type <IDENTIFIER> )? ")" ";"
   | "public" VoidType <IDENTIFIER>
   "(" ( Type <IDENTIFIER> )? ")" ";"
   | "public" VoidType <IDENTIFIER>
   "(" ( Type <IDENTIFIER> )? ")" ";"
   | "public" VoidType <IDENTIFIER>
   "(" ( Type <IDENTIFIER> )? ")" ";"
Type ::= <BOOLEAN>
   | <INT>
   | <IDENTIFIER>
VoidType ::= "void"
Statement ::= "(" ( Statement )* ")"
   | "if" "(" Expression ")" Statement ( "else" Statement )?
   | <IDENTIFIER> "=" Expression ";"
   | <IDENTIFIER> "." <IDENTIFIER> "(" Expression ")"
   | ( COMMA Expression )? ";"
   | <THIS> "." <IDENTIFIER> "(" ( Expression )?
```

144
APPENDIX C. THE BNF SYNTAX OF MYJAVA IN JAVACC

```
( <COMMA> Expression )* ";"
| "(" <IDENTIFIER> "(" <IDENTIFIER> ")" ")" "," <IDENTIFIER>
| "(" ( Expression )? ( <COMMA> Expression )* ")" ";"
| <WHILE> "(" Expression ")" Statement

Expression ::= <INTEGER_LITERAL> ( ExpressionP )?
| <TRUE> ( ExpressionP )?
| <FALSE> ( ExpressionP )?
| <IDENTIFIER> ( ExpressionP )?
| <THIS> ( ExpressionP )?
| <BANG> Expression ( ExpressionP )?
| <LPAREN> Expression <RPAREN> ( ExpressionP )?
| <NEW> <IDENTIFIER> <LPAREN> <RPAREN>

ExpressionP ::= <AND> Expression ( ExpressionP )?
| <OR> Expression ( ExpressionP )?
| <LESS> Expression ( ExpressionP )?
| <EQUALS> Expression ( ExpressionP )?
| <LESSEQUALS> Expression ( ExpressionP )?
| <PLUS> Expression ( ExpressionP )?
| <MINUS> Expression ( ExpressionP )?
| <TIMES> Expression ( ExpressionP )?
| <DOT> <IDENTIFIER> <LPAREN> ( Expression ( <COMMA> Expression )* )? <RPAREN> ( ExpressionP )?

// Define keyword tokens
<DEFAULT> TOKEN : {
  <BOOLEAN: "boolean">
  | <CLASS: "class">
  | <ELSE: "else">
  | <FALSE: "false">
  | <FOR: "for">
  | <RETURN: "return">
  | <THEN: "then">
  | <NEW: "new">
  | <VOID: "void">
  | <TRUE: "true">
  | <MAIN: "main">
  | <THIS: "this">
  | <WHILE: "while">
  | <PUBLIC: "public">
  | <PRIVATE: "private">
  | <PROTECTED: "protected">
  | <STATIC: "static">
  | <STRING: "String">
  | <LENGTH: "length">
  | <EXTENDS: "extends">
  | <IF: "if">
  | <INT: "int">
  | <SYSOUT: "System.out.println">
  | <INVARIANT: "INVARIANT">
  | <PRECONDITION: "PRECONDITION">
  | <POSTCONDITION: "POSTCONDITION">
  | <INTERFACE: "interface">
  | <IMPLEMENTS: "implements">
```
APPENDIX C. THE BNF SYNTAX OF MYJAVA IN JAVACC

146

// Define identifiers and literals

<DEFAULT> TOKEN : {
  <INTEGER_LITERAL: <DECIMAL_LITERAL> (["1","L"])?
  | <#DECIMAL_LITERAL: ["0"-"9"] (["0"-"9"])>*
  | <IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)+
  | <LETTER: ["a"-"z","A"-"Z","_"]>
  | <DIGIT: ["0"-"9"]>
}

// Define enclosing tokens

<DEFAULT> TOKEN : {
  <LPAREN: "(">
  | <RPAREN: ")"> 
  | <LSQRBRAC: "["> 
  | <RSQRBRAC: "]"> 
  | <LBRACE: "{"> 
  | <RBRACE: "}"> 
  | <SEMICOLON: ";"> 
}

// Define operator tokens

<DEFAULT> TOKEN : {
  <BANG: "!"> 
  | <LESS: "<"> 
  | <PLUS: "+"> 
  | <TIMES: "*"> 
  | <MINUS: "-"> 
  | <DOT: "."> 
  | <COMMA: ","> 
  | <EQUALS: "=="> 
  | <AND: "&&"> 
  | <OR: "||"> 
  | <LESSEQUALS: "<="> 
}

Appendix D

Inference rules in LF
### Appendix D. Inference Rules in LF

#### Contexts and Assumptions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash K$ kind $x \notin \text{FV}(\Gamma)$</td>
<td>$\Gamma, x : K, \Gamma' \vdash \Gamma, x : K$ valid</td>
</tr>
</tbody>
</table>

#### General Equality Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash K$ kind</td>
<td>$\Gamma \vdash K = K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash K = K'$</td>
<td>$\Gamma \vdash K = K''$</td>
</tr>
<tr>
<td>$\Gamma \vdash k : K$</td>
<td>$\Gamma \vdash k = k' : K$</td>
</tr>
<tr>
<td>$\Gamma \vdash k = k' : K$</td>
<td>$\Gamma \vdash k = k'' : K$</td>
</tr>
</tbody>
</table>

#### Equality Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash k : K$</td>
<td>$\Gamma \vdash K = K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash k = k' : K$</td>
<td>$\Gamma \vdash K = K''$</td>
</tr>
</tbody>
</table>

#### Substitution Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : K, \Gamma' \vdash k : K$ kind $\Gamma \vdash k = k' : K$</td>
<td>$\Gamma, [k/x] \Gamma' \vdash k : [k/x]K' \text{ kind}$</td>
</tr>
<tr>
<td>$\Gamma, x : K, \Gamma' \vdash k = k' : K$</td>
<td>$\Gamma, [k/x] \Gamma' \vdash k = k' : [k/x]K' \text{ kind}$</td>
</tr>
</tbody>
</table>

#### The Kind Type

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash A : \text{Type}$</td>
<td>$\Gamma \vdash A = B : \text{Type}$</td>
</tr>
</tbody>
</table>

#### Dependent Product Kinds

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash K$ kind $\Gamma, x : K \vdash K'$ kind</td>
<td>$\Gamma \vdash (x : K)K'$ kind</td>
</tr>
<tr>
<td>$\Gamma \vdash (x : K)K'$ kind</td>
<td>$\Gamma \vdash K_1 = K_2$</td>
</tr>
<tr>
<td>$\Gamma, x : K \vdash k : K'$</td>
<td>$\Gamma \vdash K_1 = K_2$</td>
</tr>
<tr>
<td>$\Gamma, x : K \vdash k = k_1 = k_2 : K$</td>
<td>$\Gamma \vdash [x : K]k_1 = [x : K]k_2 : (x : K)K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash f : (x : K)K'$</td>
<td>$\Gamma \vdash f = f' : (x : K)K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash f = f' : (x : K)K'$</td>
<td>$\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash f : (x : K)K'$</td>
<td>$\Gamma \vdash [x : K]f(x) = f : (x : K)K'$</td>
</tr>
<tr>
<td>$\Gamma \vdash (x : K)K'$</td>
<td>$\Gamma \vdash f : (x : K)K'$</td>
</tr>
</tbody>
</table>