ACHIM D. BRUCKER

AN INTERACTIVE PROOF ENVIRONMENT FOR OBJECT-ORIENTED SPECIFICATIONS

DISS. ETH. NO. 17097

An Interactive Proof Environment for Object-oriented Specifications

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

Achim D. Brucker

Diplom Informatiker, University of Freiburg, Germany
born 06.08.1975
citizen of Germany

accepted on the recommendation of
Prof. Ph.D. David Basin, examiner
Prof. Dr. Reiner Hähnle, co-examiner
Prof. Dr. Peter Müller, co-examiner

2007

The method of "postulating" what we want has many advantages; they are the same as the advantages of theft over honest toil. Let us leave them to others and proceed with our honest toil.

(Russell [103], p. 71)

ABSTRACT

We present a semantic framework for object-oriented specification languages. We develop this framework as a conservative shallow embedding in Isabelle/HOL. Using only conservative extensions guarantees by construction the consistency of our formalization. Moreover, we show how our framework can be used to build an interactive proof environment, called HOL-OCL, for object-oriented specifications in general and for UML/OCL in particular.

Our main contributions are an extensible encoding of object-oriented data structures in HOL, a datatype package for object-oriented specifications, and the development of several equational and tableaux calculi for object-oriented specifications. Further, we show that our formal framework can be the basis of a formal machine-checked semantics for OCL that is compliant to the OCL 2.0 standard.

ZUSAMMENFASSUNG

In dieser Arbeit wird ein semantisches Rahmenwerk für objektorientierte Spezifikationen vorgestellt. Das Rahmenwerk ist als konservative, flache Einbettung in Isabelle/HOL realisiert. Durch die Beschränkung auf konservative Erweiterungen kann die logische Konsistenz der Einbettung garantiert werden. Das semantische Rahmenwerk wird verwendet, um das interaktives Beweissystem HOL-OCL für objektorientierte Spezifikationen im Allgemeinen und insbesondere für UML/OCL zu entwickeln.

Die Hauptbeiträge dieser Arbeit sind die Entwicklung einer erweiterbaren Kodierung objektorientierter Datenstrukturen in HOL, ein Datentyp-Paket für objektorientierte Spezifikationen und die Entwicklung verschiedener Kalküle für objektorientierte Spezifikationen. Zudem zeigen wir, wie das formale Rahmenwerk verwendet werden kann, um eine formale, maschinell geprüfte Semantik für OCL anzugeben, die konform zum Standard für OCL 2.0 ist.

CONTENTS

1 INTRODUCTION 11
- 1.1 Motivation 11
- 1.2 Contributions 13
- 1.3 Related Work 15
- 1.4 Overview 18
- 1.5 Typographic Conventions 19

2 FOUNDATIONS AND BACKGROUND 21
- 2.1 Object-oriented Specifications 21
- 2.2 Formal Background 30

3 A FRAMEWORK FOR OBJECT-ORIENTED SPECIFICATION 41
- 3.1 Challenges 41
- 3.2 Theory Morphism and Structuring 45
- 3.3 Defining an Object-oriented Constraint Language 54
- 3.4 Formalizing Object-oriented Data Structures 61
- 3.5 Towards a Constrained Object Store 71
- 3.6 Equalities and Object-orientation 81
- 3.7 Operations for Accessing the System State 86
- 3.8 On Operation Specifications 87
- 3.9 Operation Calls 89
- 3.10 Operation Invocations 91
- 3.11 Limits to Recursive Invocations and Calls 94
- 3.12 Specifying Frame Properties 97
- 3.13 Discussion 99

4 A FORMAL SEMANTICS FOR UML/OCL 103
- 4.1 Challenges 103
- 4.2 A Note On OCL Standards 104
- 4.3 A Machine-checked OCL Semantics 107
- 4.4 A Note On Standard Compliance 116

4.5 Extending OCL 125
4.6 Discussion 129

5 CALCULI FOR OBJECT-ORIENTED SPECIFICATIONS 131
5.1 Challenges 131
5.2 Validity and Judgments 133
5.3 Equivalences and Congruences 135
5.4 Subcalculi 139
5.5 The Logic 144
5.6 Calculi 150
5.7 Discussion 158

6 APPLICATIONS 163
6.1 The HOL-OCL System 163
6.2 Case Studies 167
6.3 Discussion 172

7 RELATED WORK 175
7.1 Embeddings of Object-oriented Languages 175
7.2 Datatype Packages 177
7.3 Proof Support for Three-valued Logics 177
7.4 Formal OCL Semantics 178
7.5 Tool Support 179

8 CONCLUSION AND FUTURE WORK 183
8.1 Conclusions 183
8.2 Summary of Contributions 184
8.3 Future Work 186

A THE SYNTAX OF OCL 189

B THE INVOICE SYSTEM 195
B.1 Informal Description 195
B.2 Formal Specification 196

1 INTRODUCTION

In this chapter, we motivate our work and summarize our contributions. Moreover, we give a first, brief overview of related work and introduce the overall structure of this thesis.

1.1 MOTIVATION

Computer systems, both in hardware and in software, are becoming more and more complex. Our daily life depends on the reliable and correct behavior of such systems, e. g., electronic drive control systems used in cars, automatic flight control systems, or medical systems. It is difficult to get computer systems right. Formal methods are one way to ensure the correctness of such vital systems; thus, formal techniques should be integrated into the development process of such systems. Just like using blueprints in common engineering practice, the development of complex software systems requires a detailed specification describing the data structures and the desired behavior of the system.

Specification documents can vary in their precision from informal textual descriptions or structured text to formal specification using a language based on mathematical logic such as Z [108] or VDM [62]. Depending on their precision, computer-supported techniques can be applied that assure the consistency of specification documents. For example, type-checking or well-formedness-checking can be used to find some inconsistencies in requirement and design documents of a system and thus provide an a priori analysis. More evolved techniques can assure the correct transition from a specification document to the implementation. These techniques and their computer support are summarized under the term formal methods. Obviously, the power of these techniques crucially depends on the degree of formality.

Producing formal specifications and maintaining their consistency during system development is a task that requires a lot of effort and training. This is to an even larger extent true for the validation phase, where techniques such as model-based testing, model-checking or inter-

active theorem proving are applied. Industry has been reluctant to accept formal methods within their daily practice so far. Although, it is meanwhile widely accepted that specification and testing activities outweigh by far the costs of the implementation phase of a large system and that formal methods have a positive effect here. Rather, the overwhelming need for specification led to the development of semi-formal specification documents that have their roots in light-weight graphical notations. A prominent example is the *Unified Modeling Language* (UML) [90], which is widely accepted by the industry for developing software following the object-oriented methodology. Instead of using mathematical notation such as Z or the Hoare Calculus, there is a trend to introduce formal specification techniques featuring a syntax that is close to the syntax of object-oriented programming languages. Usually, these formal specifications based on the specification of preconditions and postconditions. Moreover, the choice of using a syntax similar to programming languages helps software developers by using specification formalisms they are familiar with, both with respect to the syntax and semantics.

The definition of the UML [90] is one result of the industrial need for computer-support of such light-weight specification methods. UML is defined in an open standardization process led by the Object Management Group (OMG) and both OMG standards in general and UML in particular are widely accepted in the industry. Overall, the UML offers an integrated object-oriented development methodology ranging from informal requirement analysis over object-oriented modeling (design phase) to code-generation. This design-centered, or model-driven software development process is known as Model Driven Architecture (MDA) or Model Driven Engineering (MDE).

The UML provides a standard for graphical, or diagrammatic, specification notations, representation in abstract syntax and partly also their semantics. These notations comprise among others activity diagrams, sequence diagrams, class diagrams and state diagrams. The latter two notations are of particular interest from the perspective of formal methods, since they represent forms of data-oriented and behavioral modeling, which can be considered as well-known concepts in a new shape. Moreover, with Unified Modeling Language (UML) version 1.3 the *Object Constraint Language* (OCL), a textual annotation formalism (constraint language) was added to the UML standard. OCL is heavily used in the specification documents of the metamodels of the UML itself.

From the perspective of formal methods, the success of CASE tools supporting the UML opens a door for bringing formal methods a step closer to industry. However, to turn this vision into reality, several challenges need to be faced: first, the semantics of UML/OCL is conceptually much closer to an object-oriented programming language than to a traditional logic, although OCL comprises a version of predicate logic and

arithmetic. Second, much effort has to be invested to cope with the object-oriented features of OCL in a logically clean way, allowing for adequate symbolic computations. Third, an adequate proof methodology for object-oriented modeling, as it is meanwhile established in the user community of UML/OCL, has to be developed. Overall, this will allow for applying formal methods using a uniform specification language during the complete software development process. Probably, this will increase the use of formal methods during the development of computer systems and thus improve the quality of software we daily use.

1.2 CONTRIBUTIONS

This thesis shows that the conservative embedding technique can be successfully used for both defining the semantics of a specification language and for building formal tools for the embedded language. Moreover, this construction guarantees the correctness of the developed tools with respect to the defined semantics. In particular, we confirm the following claims:

- A shallow embedding can be used for defining the semantics of an object-oriented specification language; including the underlying object-oriented data structures, i. e., for modeling an object store.
- A shallow embedding can be used for developing formal tool support for a specification language used in industry.
- Defining the semantics, and also building tools, in an conservative way, i. e., without using (unproven) axioms, is feasible; even for such a rich language as OCL. Moreover, we provide evidence that the consistency guarantee of this approach outweighs the larger effort compared to an axiomatic approach.
- A conservative embedding technique is useful to compare different semantical variants and possible language extensions in a logical safe way, i. e., without the risk of introducing inconsistencies.
- Developing a machine-checked formalization of a real-world, i. e., defined by an industrial committee, standard of a specification language is feasible.

For supporting our thesis, we have conducted work that is both of theoretical and of practical importance. On the theoretical side, we provide:

- A type-safe representation of object-oriented data structures in the typed λ-calculus and in particular higher-order logic (HOL).

This includes a typed, extensible UML data model supporting inheritance and subtyping inside the typed λ-calculus with parametric polymorphism. As a consequence of the conservativity with respect to HOL, we can guarantee the consistency of the semantic model.

- A formal semantics of object-oriented data structures with invariants enriched with a precondition/postcondition style specifications for operations.
- An extensible encoding of object-oriented data structures into HOL. Moreover, we developed automatic support for our encoding. This includes also the development of a datatype package for extensible, object-oriented data-structures with invariants.
- Proof calculi for a Strong Kleene Logic over path expressions. In particular, we provide several derived calculi for UML/OCL that allow for formal derivations establishing the validity of UML/OCL formulae. Automated support for such proofs is also provided. However, since our embedding, called HOL-OCL, comprises predicate logic with equality and a typed set theory, the validity of a formula is undecidable and the logic is inherently incomplete with respect to the class of standard models of HOL [2].

On the practical side, we developed HOL-OCL, an interactive proof environment for object-oriented specifications, in particular using UML/OCL. The main design goals of HOL-OCL can be summarized as follows:

- HOL-OCL implements a generic framework for object-oriented specification languages and means to compare different semantic variants thereof.
- HOL-OCL follows semantically the OCL 2.0 standard [88].
- HOL-OCL provides technical support for importing UML/OCL models in the XMI-formats generated by conventional UML/OCL modeling tools.
- HOL-OCL allows for analyzing UML/OCL models with all the means provided by an up-to-date interactive, tactic-driven theorem proving environment such as Isabelle [86].

HOL-OCL also provides contributions for improving the definition and development of the language UML/OCL itself:

- It defines a machine-checked formalization of the semantics as described in the standard for OCL 2.0 [88]. This is implemented as a conservative, shallow embedding of OCL into HOL.

- The standard postulates requirements on the semantics of OCL operators. The OCL semantics—which is contained in an informative, i. e., non-normative, appendix of the standards documents—is not formally related to these requirements. We provide formal proofs that our formalization of the OCL semantics indeed meets the requirements.
- Our work detected formal contradictions of the OCL standard, in particular with respect to the derived calculi. Overall, our embedding strives for compliance with the OCL 2.0 standard, i. e., whenever our embedding differs from the standard, we give reasons for it and document this fact clearly.
- It represents a technical framework (including a graphical front-end based on Proof General [5] and a programming interface for SML) enabling one to implement particular *formal methods* based on UML/OCL.

1.3 RELATED WORK

In this section, we give a broad but brief overview of related work. This discussion is acting as a first classification of the overall subject of this thesis but should neither be understood as being complete nor as providing a detailed comparison. We give an overview of formal specification languages, formal tools, and a report on the state of the art regarding OCL. A detailed discussion of closely related work will be given in Chapter 7.

1.3.1 *Formal Specification Languages*

There is a variety of formal software specification languages, mainly developed by academia. Most of the well known formal notations, e. g., Z [108], are not geared toward object-orientation. For example, Z is based on set theory and first-order predicate logic without special support for object-oriented concepts. Nevertheless, there are various object-oriented extension for Z [106, 109] available but they fail to provide a strong link to object-oriented methodologies as they are used in industry. Moreover, these efforts for integrating object-orientation into well-known formal methods are faced with the same criticism from industry as their ancestors: the notation used by formal methods is difficult to understand, there is a lack of tools supporting formal methods, and the costs of using formal methods are high.

In contrast, object-oriented specification languages like UML are highly accepted in industry. Moreover, with Object Constraint Language (OCL), which is part of UML, a semi-formal constraint language is also provided. OCL allows for the specification of constraints which were not directly

expressible within UML. Besides OCL, there are other light-weight formal methods for object-oriented systems; the most prominent ones are Alloy [59], the Java Modeling Language (JML) [70], and Spec# [6].

Alloy was one of the first languages that faced the object-oriented community by advertising itself as being compatible with graphical object models. Its development was influenced by Z, but Alloy is geared towards fully automatic decision procedures, i. e., parallel to defining the semantics of Alloy, a specialized model-checker was developed [60]. Alloy provides a composition that is based on adding fields which is somewhat similar to inheritance and also a concept of reuse of formulae by explicit parametrization, similar to functions in a functional programming language. But this is not sufficient to model object-orientation in all its aspects.

JML is an interface specification language that can be used to specify the behavior of Java modules. Among other, it allows the annotation of Java source with preconditions, postconditions and invariants using a Java-like syntax. It combines the design by contract approach of Eiffel [78] and the model-based specification approach of the Larch family [47] of interface specification languages, with some elements of the refinement calculus. More informally, one can state that JML has a similar relation to Java as OCL has to UML or Spec# has to C#. As JML and Spec# are tightly connected to a programming language, the formal tool support for both are geared towards program verification.

1.3.2 *Formal Tools for Object-oriented Systems*

While object-oriented programming is a widely accepted programming paradigm, theorem proving over object-oriented-programs or object-oriented-specifications is far from being a mature technology. Classes, inheritance, subtyping, objects, and references are deeply intertwined and represent complex concepts that are quite remote from the platonic world of first-order logic or HOL. For this reason, there is a tangible conceptual gap between the verification of functional and imperative programs on the one hand and object-oriented-programs on the other.

Among the existing implementations of proof environments dealing with subtyping and references, two categories can be distinguished:

1. into standard logic and
2. deep embeddings into a meta-logic.

As pre-compilation tools, for example, we consider Boogie for Spec# [6, 72] or Krakatoa [75] for JML. The underlying idea is to compile object-oriented programs into standard imperative ones and to apply a verification condition generator on the latter. While technically sometimes

very advanced, the foundation of these tools is quite problematic: The compilation in itself is not verified and it is not clear if the generated conditions are sound with respect to the (usually complex) operational semantics.

Among the tools based on deep embeddings, there is a large body of literature on formal models of Java-like languages, e. g., [17, 39, 41, 84, 114]. In a deep embedding of a language semantics, syntax and types are represented by free datatypes. As a consequence, derived calculi inherit a heavy syntactic bias in form of side conditions over binding and typing issues. This is unavoidable if one is interested in meta-theoretic properties such as type-safety; however, when reasoning about applications and not over language properties, this advantage turns into a major obstacle for efficient deduction. Thus, while proofs for type-safety, soundness of Hoare Calculi and even soundness of verification condition generators are done, none of the mentioned deep embeddings has been used for substantial proof work in applications.

In contrast, the *shallow embedding* technique has successfully been used, including large applications, for semantic representations of non object-oriented languages. Examples for such embeddings are HOL itself (in Isabelle/Pure), HOLCF (in Isabelle/HOL) [80], or HOL-Z; an embedding of Z into Isabelle/HOL [26]. These embeddings have been used for substantial applications, e. g., Basin et al. [8] present an analysis of a security architecture using HOL-Z [26]. The essence of a shallow embedding is to represent object-language binding and typing directly in the binding and typing machinery of the meta-language. Thus, many side conditions are simply unnecessary; type-safety, for example, is proven implicitly when deriving computational rules from semantic definitions. Since implicit side conditions are "implemented" by built-in mechanisms, they can be handled orders of magnitude faster than an explicit treatment.

1.3.3 *Formal Semantics and Tools for OCL*

From its very first appearance, OCL gained much interest in the research community. These interests resulted in several tools supporting OCL and also in many open questions about the formal foundation of OCL.

Beside several works [37, 54, 101] discussing details of the OCL semantics, there are also early attempts for providing a formal semantics for OCL or a subset thereof. For example, Richters and Gogolla [102] present a formal semantics of OCL based on an untyped set theory and Cengarle and Knapp [31] present a type inference system and a big-step operational semantics for OCL 1.4. All these proposals for a formal OCL semantics are based on "mathematical notation" in the style of "naïve set theory," which in our opinion, especially for a typed object-language like OCL, is inadequate to cover subtle subjects such as inheritance and class invariants.

In particular, this also holds for the semantics presented by Richters [100] (a derivative of [102]) which is the basis of the informative (i. e., non-normative) semantics chapter included in recent versions of the OCL standard [88, Appendix A]. Moreover, none of these works provide a link between their formalization and the normative part of the standard, i. e., a formal proof showing that these formal semantics fulfill the pairs of preconditions and postconditions given in the normative part of the standard. The development of proof calculi and automated deduction for OCL has not been in the focus of interest so far. Furthermore, none of the presented works aims for a formal tool support that is guaranteed to follow the given semantics.

The formal tool support for OCL is still limited. Besides the integration of OCL type-checkers into several CASE tools, there are mainly two different categories of tools that are compliant to the OCL semantics of the standard [88]:

- Tools for runtime checking of OCL specifications, e. g., based on the OCL toolkit from the University of Dresden [38]. This suite consists of a Java library for representing OCL datatypes, a type-checker, and a code generator (constraint checker) that can check OCL constraints (invariants, precondition and postconditions) at runtime.

- Tools, namely USE [102] and OCLE [34], for animation of OCL specifications, allowing for the evaluation of OCL expression in the context of a UML model. Overall, these environments can be used to validate a UML against an OCL specification, i. e., one can check that a given model is well-formed where the well-formedness rules are expressed using OCL.

In particular, there is no proof environment for OCL, or a subset thereof, that is based on a semantics that is compliant to the standard, e. g., supporting a three-valued logic.

1.4 OVERVIEW

This thesis is structured as follows: In Chapter 2 we present the foundations of this thesis, i. e., first we give a brief introduction into object-orientation with a particular focus on UML/OCL. Second, we introduce the higher-order logic (HOL), the interactive theorem prover Isabelle/HOL, and how it can be used for providing both a machine checked formal semantics for a language and a formal tool for that language.

In Chapter 3, we develop a framework for defining the semantics for object-oriented specification languages. This framework comprises building blocks for the semantics of object-oriented data structures, i. e., an

object store, and an object-oriented constraint language for reasoning over these data-structures.

We use this framework in Chapter 4 for defining a formal semantics of OCL that is compliant with the standard. Moreover, we present several extensions of OCL and propose changes of the OCL semantics. These extensions and changes will make OCL easier to understand and more suitable for formal analysis.

In Chapter 5, we present several calculi for three-valued logics that reason over path expressions, i. e., object-oriented data-structures. This includes also the development of several sub-calculi, e. g., specialized for the reasoning over definedness. We also discuss their use for semi-automatic reasoning.

In Chapter 6, we give a brief overview of HOL-OCL. HOL-OCL is an interactive theorem prover for UML/OCL specifications that is implemented using the formal framework of Chapter 4. Moreover, we show the usability of HOL-OCL in a case study.

We give a detailed discussion of related work in Chapter 7. Finally, we conclude the thesis and discuss future work in Chapter 8.

1.5 TYPOGRAPHIC CONVENTIONS

The following typographic conventions are adopted in this thesis:

- Within UML/OCL specifications, OCL expressions are either written inline, like `self.s->includes(5)`, or together with their context specification:

```
context A:
  inv: self.s->includes(5)
```

  Keywords are printed in a blue typeface.

- OCL formulae that are interpreted within HOL-OCL are written inline as *self*.*s*`->includes`(5), or alternatively in mathematical syntax as: $5 \in (self.s)$. We provide this mathematical notation for OCL as an alternative concrete syntax. Overall, HOL-OCL supports both notations, but we prefer the mathematical one for semantic definitions and proof work. Appendix A provides a brief comparison of both concrete syntaxes for OCL.

- We use a color coding to distinguish OCL and HOL sub-expressions in formulae containing both e. g.,

$$\cup \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda\, X.\ \mathrm{strictify}(\lambda\, Y.\ \mathrm{Abs}_{\mathrm{Set}\,\iota}\ulcorner \mathrm{Rep}_{\mathrm{Set}}\, X\urcorner \cup \ulcorner \mathrm{Rep}_{\mathrm{Set}}\, Y\urcorner_{,})\big)\Big). \quad (1.1)$$

Overall, HOL expressions are printed using the default color, i. e., black. We resolve ambiguities between the underlying mathematical syntax (i. e., HOL) and the OCL level by using colors: Expressions that are internally used within HOL-OCL, like the lifting operator $\llcorner _ \lrcorner$ are printed in a black typeface. Using our mathematical OCL syntax, expressions on the OCL level, like _ ∧ _, are written in a purple typeface. For the concrete syntax presented in the standard, e. g., _ and _, we use a purple typeface.

- HOL formulae are written using the usual mathematical notion, i. e., $s \in S$.
- Theory files for HOL-OCL and Isabelle/HOL are printed as follows:

```
theory royals_and_loyals
imports
  OCL
begin
  import_model "royals_and_loyals.xmi" "royals_and_loyals.ocl"
end
```

Keywords are printed in a blue typeface.

2 FOUNDATIONS AND BACKGROUND

In this chapter, we introduce the basic concepts needed for this thesis. As this thesis is positioned between object-orientation on the one side and formal methods on the other, this chapter is twofold: in Section 2.1 we introduce the notion and concepts of object-orientation as it is used in this thesis. In particular, we introduce concepts like classes, objects, and inheritance. As an example of an object-oriented specification and modeling language, we give a short overview of the Unified Modeling Language (UML) and the Object Constraint Language (OCL). In Section 2.2, we introduce the formal concepts this thesis is based on. In particular, we explain the key concepts of the interactive theorem prover Isabelle and introduce higher-order logic (HOL). We also show how Isabelle/HOL can be used both for defining semantics and for the development of tools supporting formal methods.

2.1 OBJECT-ORIENTED SPECIFICATIONS

In this section, we give a brief introduction to the concepts of object-oriented specification formalisms. We assume, that the reader is somewhat familiar with object-orientation, e. g., in a way it is used in programming languages like Java. We will only introduce the key concepts and notions used in this thesis.

2.1.1 *The Object-oriented Paradigm*

While developed in the database community, object-orientation gained much of its success in the programming languages community. Many of the widely used programming languages, e. g., Java, feature object-orientation. The object-oriented paradigm emphasizes the following key concepts:

Class: A *class* defines a unit consisting of a data specification, i. e., defined by its *attributes*, and its behavior, e. g., defined by its *methods* and *operations*. The attributes, methods, and operations are also called

class
attribute

member

members of a class. A class is the basis of modularity and structure in an object-based or object-oriented setting.

object
object identifier
reference

Object: An *object* is an instance, i. e., a runtime representation, of a class. Usually, every object is uniquely identified by its *object identifier* (oid) or *reference*.

primitive type
boxing
basic types

Primitive Type: A *primitive type* is a type whose instances are directly represented by their value. Many object-oriented programming language provide primitive types for integers, real numbers and strings. Wrapping a primitive type into a class is called *boxing*, e. g., Java for example provides both the primitive type `int` for an unboxed and the class type `Integer` for a boxed implementation of the integers. Sometimes, primitive types are also called *value types* or *basic types*.

encapsulation
access specifier

Encapsulation: Using *encapsulation*, a class can hide details of its implementation, i. e., methods, operations, and attributes. It ensures that an object can be changed only through well-defined interfaces. The accessibility of class members is specified using *access specifiers*. Common access specifiers are *private*, *protected*, and *public*. Whereas private members of a class are only accessible inside the class itself, protected parts are also accessible by subclasses. Public members are accessible from everywhere.

abstraction

Abstraction: The ability of a program to ignore the details of an object's (sub)class and work at a more generic level when appropriate is called *abstraction*.

inheritance
single inheritance
multiple inheritance

Inheritance: Using *inheritance*, a more specific element can incorporate structure and behavior defined by a more general element. Inheritance is the preferred way of extending classes and establishes a "is-a" relationship. If a class is derived from exactly one superclass, the relationship is called *single inheritance*. Otherwise, i. e., if a class has several superclasses, it is called *multiple inheritance*.

subtype
supertype
single subtyping
multiple subtyping

Subtyping: A *subtype* is a datatype that is related to another, more general, one (the *supertype*) by some notion of substitutivity, meaning that computer programs written to operate on elements of the supertype can also operate on elements of the subtype. More specifically, the supertype-subtype relation is often taken to be the one defined by the *Liskov substitution principle* [73]. As for inheritance, we call this relation *single subtyping* if a subtype is related to exactly one supertype. Otherwise, i. e., if a subtype has several supertypes, we call this relation *multiple subtyping*.

Polymorphism: Using *polymorphism*, the same method can be provided with different types. Two common types of polymorphism are *overloading polymorphism* and *overriding polymorphism*. The former is based on the *overloading* of operations, i. e., different variants of the same operation of method, only differing in the types of their arguments, are defined within the same class. The latter is based on the *overriding* of operations, i. e., the re-definition of an operation or method having the same arguments within a subtype of inheritance hierarchy. In the case of overriding polymorphism, the behavior of the operation of method varies depending on the class in which the behavior is invoked.

polymorphism

overloading

overriding

Whereas these terms define different concepts, they are often mixed, i. e., subtyping is often implemented via inheritance. But, this is not necessarily the case, e. g., subtyping can be implemented without using inheritance. For a good understanding of object-orientation, it is important to keep these two concepts separate. In Java, for example, implementing, also called realizing, an interface establishes a subtype relation that is not implemented by inheritance. Moreover, as multiple inheritance can introduce several kinds of ambiguities, it is not supported in many modern object-oriented programming languages like Java or C#. In contrast to a class, an *interfaces* consists of a set of operation specifications. As all classes implementing an interface have to provide implementations for these operations that comply to the specification given in the interface, multiple subtyping based on interface realization is considered to less problematic and is also supported by languages like Java and C#.

interface

Further, we call a language *object-based* if it supports the most of the above described properties but does not support inheritance. An object-based language that also supports inheritance is called *object-oriented*.

object-based

object-oriented

In the following, we distinguish between methods and operations. An *operation* is a possibly non-executable specification of a behavioral aspect of a class; for example, pairs of preconditions and postconditions specify an operation. A *method* is an executable implementation in a programming language, thus an operation can be implemented by several methods (i. e., using different algorithms or even different programming languages). Usually, one requires that a method (implementation) is a refinement of an operation (specification).

operation

method

Furthermore, in an object-oriented setting with subtyping, an expression has always a dynamic and a static type:

DEFINITION 2.1 (STATIC TYPE) A type that can be checked statically, i. e., at compile time, is called *static type*. □

static type

DEFINITION 2.2 (DYNAMIC TYPE) A type that is inferred during runtime is called *dynamic type*. □

dynamic type

At a given execution point, the dynamic type must always conform to the static type. Moreover, the static type and dynamic type of an expression may identical. Informally, the dynamic type of an object is the type as which the object was initially created, e. g., by calling a constructor. The static type is the type the object is acting as. Moreover, the static type can be changed, via *type-casts*, along the subtype hierarchy. For example, assume an expression that refers to an instance of the class `Account`: thus the static type of this expression is `Account`. Nevertheless, at a given execution point, this expression may be assigned to an object of class `LimitedAccount` (which must be a subclass of `Account`); in this case, the dynamic type of the expression is `LimitedAccount`.

type-cast

The dynamic type of an object determines which concrete implementation of an overridden method or operation is invoked. In more detail, we distinguish between operation or method calls and invocations. An operation *call* can be statically resolved, i. e., already at compile time the concrete implementation that is called can be determined. In contrast, an operation *invocation* cannot be resolved statically, i. e., due to overriding polymorphism the concrete operation to call can only be determined during runtime; this is also called *late-binding*.

call

invocation

late-binding

2.1.2 *A Short Introduction to UML/OCL*

In industry, the Unified Modeling Language (UML) is probably the most widely used object-oriented specification language. It is mainly known as a diagrammatic specification language providing a variety of diagram types describing the static structure, the behavior, and the interaction of an object-oriented system. The structure diagrams like class diagrams, component diagrams or object diagrams allow one to model the structure and data model of a system. Using behavior diagrams like state-machine diagrams or activity diagrams, one can also specify the intended behavior of the system. Further, a special variant of the behavioral diagrams are the interaction diagrams (e. g., sequence diagrams or collaboration diagrams) for modeling the control and data flow of a system.

OBJECT-ORIENTED DATA MODELING USING UML. The core part of the *Unified Modeling Language* (UML) is concerned with the modeling of object-oriented data models, or the structure of an object-oriented system, especially using class diagrams. A *class diagram* is a *structural diagram* showing the classifiers (classes, interfaces, etc.) and the various static relationships between them. A concrete class diagram usually only shows a limited view of the overall structural system model (i. e., not every class of a model must be visualized using a class diagram). Thus, a class diagram is only a partial visualization of the underlying object-oriented *data model*.

Unified Modeling Language (UML)

class diagram

data model

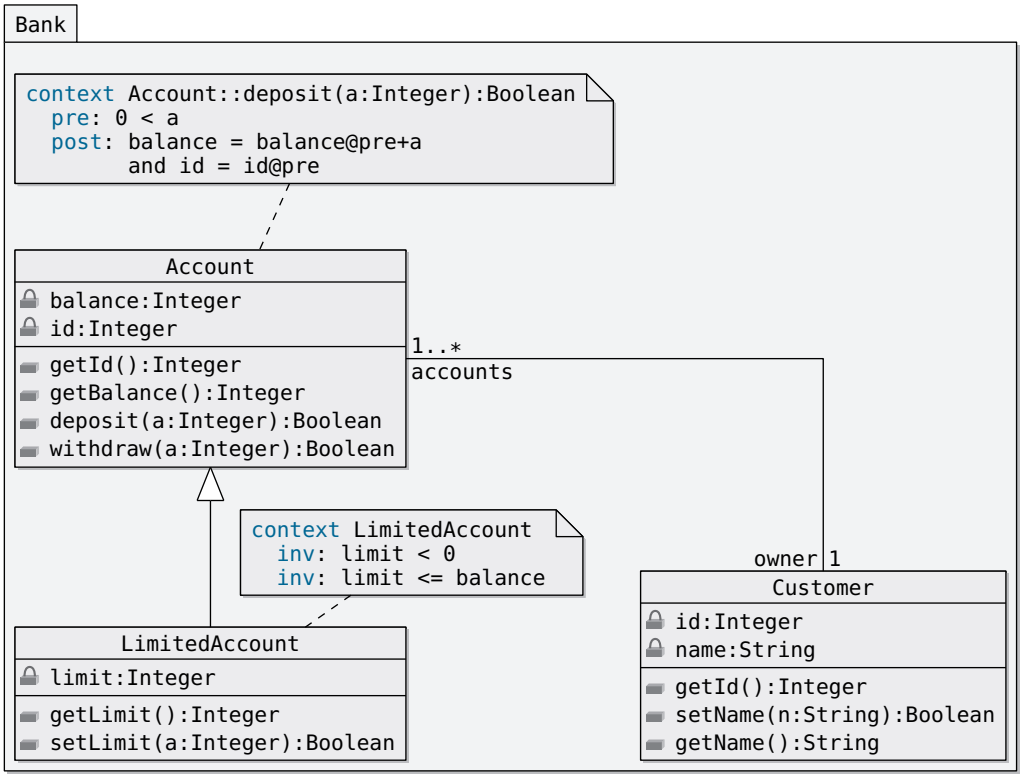


Figure 2.1: Modeling a simple banking scenario with UML/OCL. An `Account` is owned by a `Customer`, which itself can have one or more accounts. An account can either be a generic account, or one which can only be debited up to a specific limit; this is expressed using OCL invariants. The behavior of the operation `Account::deposit(amount:Integer):Boolean` is specified by an OCL precondition and postcondition pair.

The class diagram in Figure 2.1 on the preceding page illustrates the data model of a simple accounting scenario where customers can own different kinds of accounts and transfer money between them.

In more detail: customers are modeled as a *class* `Customer`. There are further classes modeling the different account types, a regular account called `Account` and an account `LimitedAccount` allowing credits only up to a specific limit. A class does not only describe a set of *object instances*, i. e., record-like data consisting of *attributes* such as `balance`, but also of behavioral aspects, i. e., *operations* (e. g., `getBalance()`) defined on them.

object instance
attribute
operation

generalization

The different account types are organized in a hierarchy of *generalizations*, e. g., the class `Account` generalizes the class `LimitedAccount`. An UML generalization, denoted by an outlined arrow, is implemented by *inheritance* and establishes a *subtype* relation.

association

Relations, as the one between customers and accounts, can be modeled in UML using *associations*. An association is constrained by a multiplicities, i. e., a constraint describing how many objects can be part of an association. In our example, the multiplicities of the association requires that every object instance of `Account` is associated with exactly one object instance of `Customer`. Usually, the annotation of the multiplicity one is omitted in graphical representations. In the other direction, the association models that an instance of class `Customer` is related to a (non-empty) set of instances of class `Account` or its subtypes.

package
namespace
pathname

For structuring the design model, UML introduces a generic concept of packages. A *package* allows for organizing model elements (e. g., classes, packages) and diagrams in a hierarchy. Further, many model elements, in particular classes and packages, introduce a *namespace*. Thus, the hierarchy of such model elements induces a hierarchy of namespaces. Elements within such a hierarchy of namespaces can be accessed using *pathnames*, i. e., a path over several nested namespaces, is obtained by concatenating the names of the namespaces (such as packages or classes) separated by pairs of double colons, e. g., `Bank::Account`.

visibility

The concept of access specifier is called *visibility* in UML. The visibility of an attribute or operation can either be specified textually or graphically. In Figure 2.1 all class attributes are private (denoted by a closed lock) and all operations are public (denoted by an open lock). As an alternative, private members of a class are denoted with a prefixed "`-`"-sign, e. g., `-getId():Integer` and public members with a "`+`"-sign, e. g., `+balance:Integer`. Moreover, protected members are denoted with a prefixed "`#`"-sign, e. g., `#id:Integer`.

THE OBJECT CONSTRAINT LANGUAGE. Pure UML diagrams are not precise enough for supporting a formal software development process. To

close this gab, the UML provides the *Object Constraint Language* (OCL). It is a constraint language for object-oriented designs trying to mimic the syntax of object-oriented programming languages and thus hiding the probably unfamiliar mathematical notions from its users. An overview of the concrete syntax for OCL is given in Table 2.1 on the next page in EBNF notation. This fragment, in particular, omits some syntactic variants. For example, quantified variables must be named explicitly whereas the OCL standard allows for omitting them.

Object Constraint Language (OCL)

The core of OCL is based on a three valued logic reasoning over UML path expressions. Additionally, the OCL standard also provides a library of basic data structures (e. g., `Boolean`, `Integer`, `Real`, `String`, `Set`, or `Sequence`). An overview of the types (classes) of the OCL library is given in Figure 2.2 on page 29. In principle, OCL allows for the annotation of arbitrary UML models. Nevertheless, the main focus of OCL is the annotation of models describing the static structure (e. g., class diagrams) of a system. In this context, it seems natural to make class diagrams more precise using OCL for specifying invariants, preconditions and postconditions of operations.

An *invariant* is an OCL formula attached to a class, which must, informally, evaluate to true in "all" possible system states for all instances of that class [88, p. 8]. As we will see later, requiring for an invariant that it evaluates to true for all system states is too strong, i. e., we will relax that requirement in certain situations (see Section 3.5.4). Using an invariant, we can describe in OCL that the attribute `id` is a unique identifier for all objects of type `Account`:

invariant

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

Moreover, we can also constrain the effects of operations. Using a *precondition*, we can state that only positive amounts can be deposited:

precondition

```
context Account::deposit(a:Integer):Boolean
  pre: a > 0
```

Additionally, one can describe the system state after the successful execution of the operation using a *postcondition*, e. g.:

postcondition

```
context Account::deposit(amount:Integer):Boolean
 post: balance = balance@pre+amount
       and id = id@pre
```

In postconditions, the operator `@pre` allows for accessing the previous state. If several constraints of the same type, e. g., invariants, are specified they are semantically equivalent with their conjunction. For example, the specification

contextDeclList ::= [*classifierContextDecl* | *operationContextDecl*] *contextDecl*
classifierContextDecl ::= `context` *pathName invDecl*
invDecl ::= [*invDecl*] `inv` [*simpleName*] `:` *expr*
operationContextDecl ::= `context` *operation prePostDecl*
prePostDecl ::= [*prePostDecl*] `pre` [*simpleName*] `:` *expr*
| [*prePostDecl*] `post` [*simpleName*] `:` *expr*
operation ::= [*pathName* `::`] *simpleName* `(` [*varDeclList*] `)` [`:` *type*]
varDeclList ::= [*varDeclList* `,`] *varDecl*
varDecl ::= *simpleName* [`:` *type*] [`=` *expr*]
type ::= *pathName* | *collKind* `(` *type* `)`
expr ::= *literalExp* | *pathName* [`@pre`]
| *expr*`.`*simpleName* [`@pre`] | *expr*`->`*simpleName*
| *expr*`(`[{*expr*`,`}*expr*]`)`
| *expr*`(`*expr*[`:`*type*][`=`*expr*]`,`*varDecl*`|`*expr*`)`
| *expr*`(`*varDecl* `|` *expr*`)`
| *expr*`[`{*expr*`,`}*expr*`]`[`@pre`]
| *expr*`->forAll(`*varDecl*[`;`*varDecl*]`|`*expr*`)`
| *expr*`->exists(`*varDecl*[`;`*varDecl*]`|`*expr*`)`
| *expr*`->iterate(`*varDecl*[`;`*varDecl*]`|`*expr*`)`
| *prefixOperator expr* | *expr infixOperator expr*
| `if` *expr* `then` *expr* `else` *expr* `endif`
| `let` *varDeclList* `in` *expr*
infixOperator ::= `*` | `/` | `div` | `mod` | `+` | `-` | `<` | `>` | `<=` | `>=` | `=` | `<>`
| `and` | `or` | `xor` | `implies`
prefixOperator ::= `-` | `not`
literalExp ::= *collLiteralExp* | *primitiveLiteralExp*
collLiteralExp ::= *collKind*`{`{*collLiteralPart*`,`}*collLiteralPart*`}`
collKind ::= `Set` | `Bag` | `Sequence` | `Collection` | `OrderedSet`
collLiteralPart ::= *expr* | *expr*`..`*expr*
primitiveLiteralExp ::= `Boolean` | `Integer` | `Real` | `String`
| `true` | `false` | `oclUndefined`
pathName ::= [*pathName*`::`]*simpleName*
simpleName ::= *SIMPLE_NAME*

Table 2.1: A fragment of the formal grammar of OCL, omitting syntactic variants like implicit quantified variables. We also omit some types we do not consider in this thesis, e. g., `OclMessage`.

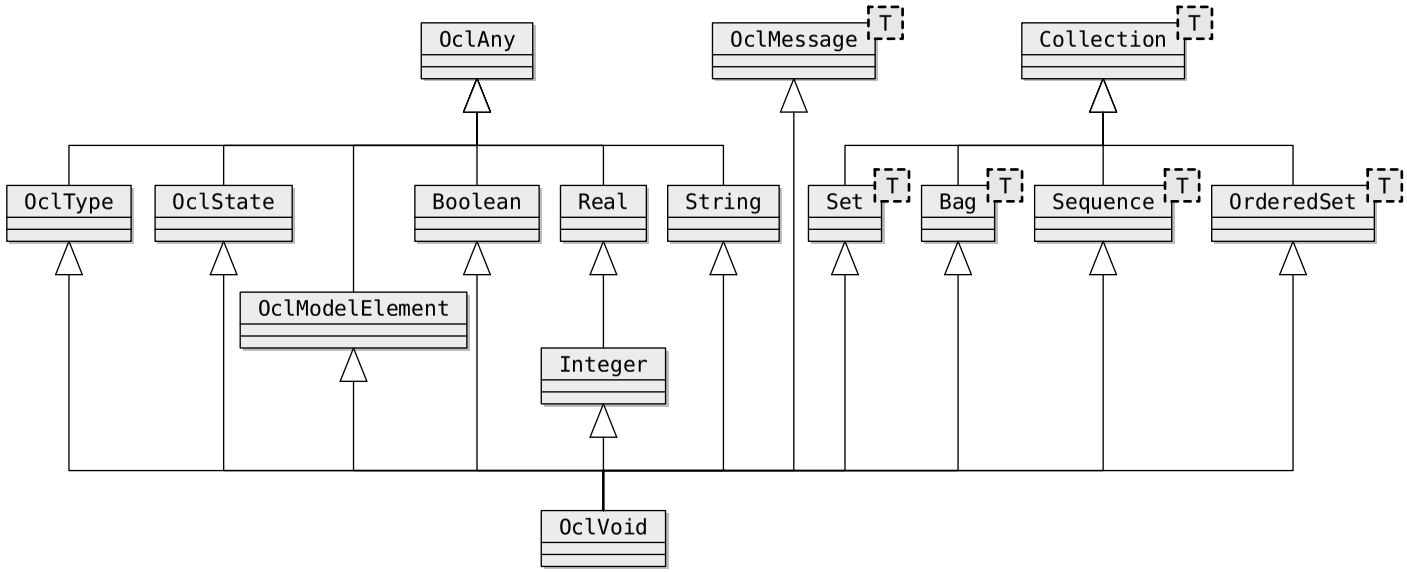


Figure 2.2: The types of the OCL standard library, except the collection types and `OclMessage`. All types are subclasses of `OclAny`. Moreover, all types are superclasses of `OclVoid`, i. e., instances of all types can be undefined. All user-defined classes are also subclasses of `OclAny` and superclasses of `OclVoid`.

```
context LimitedAccount
  inv: limit < 0
  inv: limit <= balance
```

is semantically equivalent with the following specification

```
context LimitedAccount
  inv: limit < 0 and limit <= balance
```

Further, a means of structuring the OCL specification is the possibility to name constraints explicitly. For example, by specifying

```
context LimitedAccount
  inv limitNegative: limit < 0
```

we can later refer to this constraint by its symbolic name `limitNegative`.

Many diagrammatic UML-features can be translated to OCL expressions without losing any information, e. g., associations can be represented by introducing implicit attributes into the objects with a suitable data invariant describing the multiplicities. Some of these transformations are already described in the UML standard [90].

2.2 FORMAL BACKGROUND

In this section, we give a brief overview of the formal foundations our work is based on. Overall, we assume some familiarity with mathematical notions and their use in defining language semantics formally. Thus we will limit ourselves to a brief introduction of the key concepts of the interactive theorem prover Isabelle/HOL and the specific method of how we use it for both defining the semantics of a language and providing tool support for this language.

2.2.1 *Formal Analysis: Validation and Verification*

We start our introduction to formal methods with a short excursion explaining where formal tools can be used during software development. The central point of a formal software development process is to ensure that an implementation fulfills its specification. The techniques for ensuring that a given software meets the specified requirements are called *verification* and *validation*.

validation

Software *validation* is the process of executing or evaluating the software and checking its behavior to ensure that it complies with its requirements. Often, validation is done by the utilization of various testing approaches. Software *verification* is the process of determining if software fulfills its specification or not, e. g., by proving its correctness with

verification

respect to a formal specification. Thus, verification does usually not rely on executing the software. Validation is an incomplete method, i. e., it can only find errors but cannot guarantee the correct behavior for all possible execution traces; verification provides this assurance.

Whereas verification is rarely used in "large-scale" software development, *testing*, as a validation technique, is widely used, but normally these efforts are not based on a formal specification. This strict separation between verification and validation techniques is not obvious: testing can be based on a formal specification, and thus a successful test together with explicitly stated test hypotheses [24] is not fundamentally different from program verification, see [24] for more details. Overall, tools that build upon a formal machine-checked semantics, a technique this thesis is centered around, can be used for both the development of tools for formal verification and formal validation of (object-oriented) systems.

testing

2.2.2 *Interactive Theorem Proving and Logical Frameworks*

In this section, we introduce the key concepts of higher-order logic (HOL), logical frameworks, and interactive theorem proving using Isabelle.

Interactive theorem proving deals with the *machine-supported* development of formal proofs. An *interactive proof editor* allows for guiding a semi-automatic proof search where all formal details are checked and stored by a special computer program called *theorem prover*. Such a theorem prover can either be limited to one specific logic, or it can be generic in the sense that it does not only support one built-in logic, but rather is a logical framework [97] for building new tools supporting various logics. While using a logical embedding for defining the semantics of a language it is important to distinguish between the language (logic) that is already known and the new language being defined. The newly defined logic is called *object-logic* or object-language. Its operators and inferences rules are described using an already supported logic, called the *meta-logic* or meta-language. For example, we use higher-order logic (HOL) as meta-logic for defining the semantics of OCL, one of our object-languages.

theorem prover

object-logic
meta-logic

THE LOGICAL FRAMEWORK ISABELLE. The generic theorem prover *Isabelle* [86] is a logical framework based on an LCF [45] style kernel. The proof engine of Isabelle can directly process *natural deduction rules*. The generic rule "from assumptions A_1 to A_n, infer conclusion A_{n+1}" is formally written as

Isabelle

$$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A_{n+1} \tag{2.1}$$

or using the notation of Isabelle it is written as

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow A_{n+1}\,, \tag{2.2}$$

meta-implication

where _ $\Longrightarrow$ _ denotes the built-in *meta-implication* of Isabelle. Using the usual mathematical notation, this rule is written as

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}\,. \tag{2.3}$$

Also more complex rules like "if assumption B can be inferred from assumption A, infer $A \to B$" can be expressed in Isabelle:

$$(A \Longrightarrow B) \Longrightarrow A \longrightarrow B\,. \tag{2.4}$$

This rule, called *implication introduction* and in the mathematical literature it is written as:

$$\frac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \longrightarrow B}\,. \tag{2.5}$$

In this thesis, we prefer the mathematical notion, even if we describe a system that is formalized in Isabelle.

proof state
subgoal
proof goal

A *proof state* in Isabelle contains an implicitly conjoint sequence of Horn-clause-like rules $\phi_1, \ldots, \phi_n$, called *subgoals*, and ϕ, which represents the actual *proof goal*. Logically, the subgoals and the goal form together a theorem of the form

$$\frac{\phi_1 \qquad \ldots \qquad \phi_n}{\phi}\,. \tag{2.6}$$

meta-quantifier

To cope with quantifiers, subgoals have a slightly more general form than just Horn-clauses: variables may be bound by a built-in *meta-quantifier*, e. g., as in the following rule:

$$\frac{\bigwedge x_1, \ldots, x_m.\ A_1; \ldots; A_n}{A_{n+1}}\,. \tag{2.7}$$

meta-variable

The meta universal quantifier $\bigwedge$_. _ can capture the usual side-constraint "the variables x_1, …, x_m must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be logically considered as free variables. Further, Isabelle supports *meta-variables* (e. g., written as ?x or ?y), which can be seen as "holes in a term" that can still be substituted. Meta-variables are instantiated by Isabelle's built-in higher-order unification and occur only inside proofs.

The initial proof state is built from the theorem

$$\frac{\phi}{\phi}\,, \tag{2.8}$$

which is trivially true for any (typed) formula ϕ. A theorem is proven if a final proof state of the form ϕ is reached by *tactics*, i. e., SML functions allowing for the transformation of proof states. It is a key feature of Isabelle's design that all tactics are based on a few operations provided by the logical core engine of Isabelle. Moreover, these core operations log all logical operations in a derivation tree called *proof object*; thus, if someone has serious doubts on the correct implementation of Isabelle, he may generate the proof objects and check the derivations by an independent program or just store the proof objects for archival reasons.

tactics

proof object

Isabelle supports user-programmable extensions in a logically safe way. Several generic tactics (proof procedures) have been developed; namely a simplifier based on higher-order rewriting and a tableaux-based proof-search procedures based on higher-order resolution. Building upon this basis one can, in a logically safe way, extend Isabelle to support new languages, e. g., Z [26]. Such an extension of Isabelle, if done conservatively, provides both a logical consistent semantics for the object-logic and a reliable interactive theorem prover environment for the object-logic.

The rationale behind Isabelle is to encode other logical languages, both with respect to their syntax and to their deductive system. The syntax of a language can be described using higher-order syntax and powerful pretty-printing mechanisms. The deductive system may be specified by logical rules in the built-in logical core language either by axioms or derived rules (theorems).

The distinction between axioms and theorems is important. An *axiom* is an unproven fact that is *defined* to be true. In contrast, a *theorem* (or derived rule) is a proven statement.

axiom

theorem

LOGICAL FRAMEWORKS AND HIGHER-ORDER LOGIC. Classical higher-order logic (HOL) [2, 35] is a classical logic with equality enriched by total (parametric) polymorphic higher-order functions. HOL is based on a typed version of the *λ-calculus*. The types τ are defined as

λ-calculus

$$\tau := \alpha :: \xi \mid \chi(\tau, \ldots, \tau)\,, \tag{2.9}$$

where α is a *type variable* (the set of type variables is ranging over α, β, γ, ...) and where the set of *type constructors* χ contains, among others, $_ \Rightarrow _$, bool, int, and α set. Moreover, the type system of Isabelle/HOL is two-staged: types are classified in *type classes*, e. g., ξ. The set of type classes is ranging over bot, term, ... Annotations with the default type class term can be omitted, i. e., instead of $\alpha :: \text{term}$ we may just write α. Type classes are also called *sorts*.

type variable

type constructor

type class

sort

The terms of HOL are *λ-terms* defined as

λ-term

$$\Lambda := C \mid V \mid \lambda\, V.\ \Lambda \mid \Lambda\ \Lambda\,, \tag{2.10}$$

symbol	meta-type	description
$\neg$	$\text{bool} \Rightarrow \text{bool}$	negation
true	bool	tautology
false	bool	absurdity
if	$[\text{bool}, \alpha, \alpha] \Rightarrow \alpha$	conditional
let	$[\alpha, \alpha \Rightarrow \beta] \Rightarrow \beta$	let binder

Table 2.2: Syntax and types of the HOL constants.

symbol	meta-type	description
ε	$(\alpha \Rightarrow \text{bool}) \Rightarrow \alpha$	Hilbert description
$\forall$	$(\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool}$	universal quantification
$\exists$	$(\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool}$	existential quantification
$\exists!$	$(\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool}$	unique existence

Table 2.3: Syntax and types of the HOL binders.

where C is the set of *constants* like true, false and where V is the set of *variables* like x, y, z. *Abstractions* and *applications* are written $\lambda\, x.\ e$ and $e\ e'$ or $e(e')$. A subset of λ-terms may be *typed*, i. e., terms may be associated to *types* by an inductive type inference system similar to the programming language Haskell or (to a lesser extent) SML. We do not give a formal definition of the type inference system here and refer the interested reader to [82], where also a type inference algorithm is described. In the following, we will only show type-checked λ-terms and use an intuitive understanding of types.

type

The logical terms of HOL (see Table 2.2, Table 2.3, and Table 2.4) are centered around the logical connectives. The *Hilbert operator* $\epsilon\, x.\ P\ x$ returns an arbitrary x that makes $P\ x$ true. The Hilbert operator turns HOL into a classical logic [95]. HOL may be interpreted in standard or non-standard models assigning types to carrier sets, logical operators in functions over them [46].

Hilbert operator

symbol	meta-type	description
$\circ$	$[\beta \Rightarrow \gamma, \alpha \Rightarrow \beta] \Rightarrow (\alpha \Rightarrow \gamma)$	composition
$=$	$[\alpha, \alpha] \Rightarrow \text{bool}$	equality
$\wedge$	$[\text{bool}, \text{bool}] \Rightarrow \text{bool}$	conjunction
$\vee$	$[\text{bool}, \text{bool}] \Rightarrow \text{bool}$	disjunction
$\rightarrow$	$[\text{bool}, \text{bool}] \Rightarrow \text{bool}$	implication

Table 2.4: Syntax and types of the HOL infix operators.

The logic of HOL is based on a few axioms or elementary inference rules: the implication introduction, modus ponens, and tertium non datur:

$$\frac{\begin{array}{c}[P]\\ \vdots\\ Q\end{array}}{P \longrightarrow Q}\,, \qquad \text{(implication introduction)}$$

$$\frac{P \longrightarrow Q \qquad P}{Q}\,, \text{ and} \qquad \text{(modus ponens)}$$

$$\frac{}{(P = \text{true}) \vee (P = \text{false})}\,. \qquad \text{(tertium non datur)}$$

Further only the usual laws for axiomatizing equality, reflexivity, symmetry, transitivity, extensionality, and substitutivity) are necessary:

$$\frac{}{t = t}\,, \qquad \text{(reflexivity)}$$

$$\frac{s = t}{t = s}\,, \qquad \text{(symmetry)}$$

$$\frac{r = s \quad s = t}{r = t}\,, \qquad \text{(transitivity)}$$

$$\frac{\bigwedge x.\ f\,x = g\,x}{f = g}\,, \text{ and} \qquad \text{(extensionality)}$$

$$\frac{s = t \qquad P(s)}{P(t)}\,. \qquad \text{(substitutivity)}$$

The substitutivity rule exploits the fact that term contexts (e. g., $C = A \wedge (B \vee \square)$ with the "hole" $\square$) can be directly represented inside the term language by a *λ-abstraction* (e. g., $C = \lambda\, x.\ A \wedge (B \vee x)$), while the usual substitution in a context $C[s]$ is captured by the β-reduction of the λ-calculus and the application $C(s)$.

λ-abstraction

The modules of larger logical systems built on top of HOL are Isabelle *theories*. Among many other constructs, they contain type and constant declarations as well as axioms. Stating arbitrary axioms in a theory is extremely error-prone and should therefore be avoided. Thus only a limited form of extension mechanisms, called *conservative extensions*, should be used. Using a conservative extension scheme ensures that the extended theory is consistent ("has models") provided the original theory is consistent. Four different conservative extensions have been discussed in the literature: constant definition, type definition, constant specification, and type specification [46]. For example, the most widely used *constant definition* consists of a constant declaration

theories

conservative extension

constant definition

$$c :: \tau \qquad (2.11)$$

and an axiom of the form:

$$c \equiv E\,. \tag{2.12}$$

We require that c has not been previously declared, the axiom is well-typed, E is a closed expression (i. e., does not contain free variables) and E does not contain c (no recursion). A further restriction forbids type variables in the types of constants in E that do not occur in the type τ. As a whole, a constant definition can be seen as an "abbreviation" which makes the conservativity of the construction plausible [46], and the syntactic side-conditions are checked by Isabelle automatically. The idea of an "abbreviation" is also applied to the conservative *type definition* of a new type $(\alpha_1, \ldots, \alpha_n)T$ based on its representation as a set, i. e., $\{x \mid P(x)\}$.

type definition

In this case, the set of type constructors is extended by the constructor T of arity n. The predicate P of type $\tau \Rightarrow \text{bool}$ for a base type τ constructs a set of elements τ set; the new type is defined to be isomorphic to this set. Technically, this isomorphism is stated by the declaration of two constants representing the abstraction and the representation function and by two axioms over them. More precisely, the constant Abs_T of type $\tau \Rightarrow (\alpha_1, \ldots, \alpha_n)T$ and the constant Rep_T of type $(\alpha_1, \ldots, \alpha_n)T \Rightarrow \tau$ are declared. The two isomorphism axioms have the form:

$$\text{Abs}_\text{T}(\text{Rep}_\text{T}\ x) = x \tag{2.13}$$

and

$$P(x) \Longrightarrow \text{Rep}_\text{T}(\text{Abs}_\text{T}\ x) = x\,. \tag{2.14}$$

The type definition is conservative if the proof obligation $\exists x.\ P(x)$ holds; this assures that the type is non-empty as required by the semantics of HOL.

An example for a definition of a simple but eminently useful datatype of Isabelle/HOL is

None, Some

$$\alpha\ \text{option} \coloneqq \text{None} \mid \text{Some}\ \alpha\,. \tag{2.15}$$

This datatype allows for adding a distinguished element to an already existing type. For example, this can be used for modeling partial maps. In fact, the type constructor _ ⇀ _ for partial maps in Isabelle/HOL is defined as a type synonym:

$$\alpha \rightharpoonup \beta \coloneqq \alpha \Rightarrow \beta\ \text{option}\,. \tag{2.16}$$

On top of the HOL core language, a rich set of theories can be built entirely by conservative definitions. In particular, one can derive a typed set theory including least fixed-point theory, a theory of ordering, including well-founded recursion, number theory including real number theory and theories for data-structures like pairs, type sums and lists. A large part of these theories consists in deriving rules over the defined operators, in particular those that allow for simplification and (recursive) computation.

2.2.3 *Comparing Textbook and Combinator Style Semantics*

Besides the distinction between formal and informal semantic definitions, one can also distinguish several styles for writing a formal semantics of a programming or specification language. In this section, we will discuss two widely used styles for writing formal semantics: *textbook-style* and *combinator-style*.

In textbooks, e. g., Winskel [117], formal semantics are described using a semantic interpretation function I. For example, consider the following definition for the addition over the Integers from the OCL standard [88, page A-11]:

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \tag{2.17}$$

This definition is chosen by the authors of [88, page A-11] as a representative, i. e., it is the only definition given for all *strict operations*. An operation is called strict, if evaluating the operation results in undefined, denoted by $\bot$, if at least one argument is undefined. We call the style of semantics shown in Equation 2.17 a *textbook-style semantics*. Normally, such semantics are "paper and pencil" works that are on the one hand easy to read and very useful to communicate. On the other hand, they usually lack the formalization of rules and laws, contain informal or meta-logic definitions. Moreover, it is easy to write inconsistent semantic definitions.

textbook-style semantics

Defining a *machine-checked semantics*, i. e., by embedding it into a logic supported by a theorem prover, overcomes these problems: here, formalization is enforced by the underlying theorem prover and the consistency can by guaranteed by defining the semantics conservatively. Nevertheless, formalizing rules and laws, especially when using conservative theory extensions only, is a lot of manual work. Using a *combinator-style* approach to formal semantics can reduce this work dramatically. A combinator-style formalization of semantics factorizes common properties into specific combinators, for example consider the following definition:

machine-checked semantics

combinator-style

$$_ + _ \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner \ulcorner x \urcorner + \ulcorner y \urcorner \lrcorner)\big)\Big), \tag{2.18}$$

where strictify _ is a combinator for constructing strict operations, i. e., it is defined as

$$\mathrm{strictify}\ f\ x \equiv \mathrm{if}(x = \bot)\ \mathrm{then}\ \bot\ \mathrm{else}\ f\ x \tag{2.19}$$

and lift_2 _ is a combinator for the context lifting of binary operations (for more details, see Section 3.2.1), i. e., it is defined as

$$\mathrm{lift}_2\ f \equiv \lambda\, x\ y\ \tau.\ f\ (x\ \tau)\ (y\ \tau)\,. \tag{2.20}$$

For supporting types with an explicit undefined element, we assume a type constructor $\tau_\perp$ that assigns to each type τ a type lifted by $\perp$. Moreover, the function $\llcorner_\lrcorner$ denotes the injection for lifted types, the function $\ulcorner_\urcorner$ its inverse for defined values.

Such a combinator-style approach emphasizes the semantical structure of the language being defined and is in particular well suited for machine-readable semantics and machine-supported proof calculi development. Exploiting the common structure of the definitions, one can automatically derive, from the rules of the meta-logic, a wealth of rules for the object-logic. Moreover, for a textbook semantics that is concise enough, one can prove the equivalence with a machine-checked formalization using a combinator-style semantics approach.

In the following, we will present our semantics in combinator-style, because it is more suitable for a machine-checked semantics. In particular, one can optimize an automatic proof procedure for all definitions that are based on the same setup of combinators. Moreover, this construction allows for the automatic lifting of theorems from the meta-level (e. g., HOL) to the object-level (e. g., OCL) [20]. Overall, we aim for a conservatively developed, machine-checked semantics, because we see the following advantages:

A Consistency Guarantee. If one only uses conservative definitions and only derived rules for defining the formal semantics, the consistency of the defined semantics is reduced to the consistency of HOL for the *entire language*.

A Technical Basis for a Proof Environment. Based on the derived rules, proof procedures (i. e., *tactics*) implement automated reasoning over formulae of the defined language and the correctness proof for the new proof systems is reduced to the correctness of a (well-known) HOL theorem proving system.

Formalization Experience. Since our semantics is machine-checked, we can easily change definitions and check their properties; allowing for deepening the knowledge of the language semantics as a whole.

2.2.4 *Logical Embeddings and Semantics*

embedding
higher-order abstract syntax (HOAS)

A theory representing syntax and semantics of a programming or specification language in another specification language is called an *embedding*. Further, *higher-order abstract syntax* (HOAS) [98] is an important concept for representing bindings in logical rules and program transformations [57] and for implementations [98].

As an example, we define the universal quantifier of HOL: it is represented in HOAS by a constant $\mathrm{All} :: (\alpha \Rightarrow \mathrm{bool}) \Rightarrow \mathrm{bool}$, where the term

$\mathrm{All}(\lambda\, x.\ P(x))$ is paraphrased by the usual notation $\forall\, x.\ P(x)$. This is in contrast to the usual textbook definition for predicate logic, where a free datatype for terms and predicates, explicit substitution and well-typedness functions over them is provided. This conventional representation requires explicit side-conditions in logical rules over quantifiers preventing variable clashes and variable capture. The representation using HOAS has two advantages:

1. The substitution required by logical rules like $\forall\, x.\ P(x) \Longrightarrow P(t)$ can be directly implemented by the β-reduction underlying the λ-calculus.
2. The typing discipline of the typed λ-calculus can be used to represent the typing of the represented language. For example, a multi-sorted first-order logic (having syntactic categories for arithmetic terms, list terms, etc.) is immediately possible by admitting expressions of type nat and α list.

In short, HOAS has the advantage of "internalization" of substitution and typing into the meta-language, which can therefore be handled significantly more generally and substantially more efficiently by means of the meta-logic. This is a prerequisite for using Isabelle as an implementation platform. When using HOAS style semantic definitions, this is called a shallow embedding [18] of an object-language, as opposed to a deep embedding:

- A *deep embedding* represents the abstract syntax as a datatype and defines a semantic function I from syntax to semantics. *deep embedding*
- A *shallow embedding* defines the semantics directly; each construct is represented by some function on a semantic domain. *shallow embedding*

Assume we want to embed the Boolean operators `and` and `or` into HOL. The semantics function I maps object-language expressions and environments to bool, where *environments* map variables to bool values. Using a shallow embedding, we define directly:

$$x\ \mathtt{and}\ y \equiv \lambda\, e.\ x\ e \wedge y\ e \tag{2.21}$$

and

$$x\ \mathtt{or}\ y \equiv \lambda\, e.\ x\ e \vee y\ e\,. \tag{2.22}$$

Shallow embedding allows for direct definitions as semantic domains and operations on them. In contrast, in a deep embedding, we have to define the syntax of our object-language as a recursive datatype:

$$expr = \mathtt{var}\ var \mid expr\ \mathtt{and}\ expr \mid expr\ \mathtt{or}\ expr \tag{2.23}$$

and the explicit semantic function I:

$$I[\![\texttt{var}\ x]\!] = \lambda\, e.\ e(x)\,, \tag{2.24}$$

$$I[\![x\ \texttt{and}\ y]\!] = \lambda\, e.\ I[\![x]\!]\ e \wedge I[\![y]\!]\ e\,, \text{and} \tag{2.25}$$

$$I[\![x\ \texttt{or}\ y]\!] = \lambda\, e.\ I[\![x]\!]\ e \vee I[\![y]\!]\ e\,. \tag{2.26}$$

This example reveals the main difference: compared to a shallow embedding, in a deep embedding the language is clearly separated from the underlying meta-language HOL. Moreover, semantic functions represent obstacles for deduction that are not present in a shallow embedding. The explicit syntax of deep embeddings enables induction proofs, however; for some meta-theoretic analysis, this may have advantages. Since we are interested in a concise semantic description of object-oriented specification languages and prototypical proof support, but not meta-theory, we have chosen a shallow embedding.

Another example for a shallow embedding is definition of the universal quantifier:

$$\text{All}\ P \equiv (P = \lambda\, x.\ \text{true})\,. \tag{2.27}$$

The propositional function of the "body" of the quantifier must be equal to the function that yields true for any argument. A deep representation of the universal quantifier follows usual textbooks:

$$\text{Sem}[\![\forall\, x.\ P(x)]\!]\gamma \equiv \begin{cases} \text{true} & \text{if Sem}[\![P(x)]\!]\gamma[x := d] \text{ for all } d \\ \text{false} & \text{otherwise.} \end{cases} \tag{2.28}$$

Here, we assume a meta-language with well-defined concepts such as "if," "otherwise," and "for all," e. g., Zermelo-Fraenkel set theory.

As can be seen, shallow embeddings can have a remarkably different flavor in their semantic presentation, in particular when striving for conservativity as in the example above. However, the usual inference rules are derived from these definitions. Thus they are equivalent in the sense that they describe semantically the same language.

conservative embedding

If a shallow embedding is built entirely by conservative theory extensions, it is called a *conservative embedding*. A conservative extensions ensures the consistency of the language definition if the underlying meta-language is consistent.

In the following chapter, we will use these techniques for building a formal framework for object-oriented languages. This framework, implemented as a conservative shallow embedding into Isabelle/HOL, provides both a combinator-style formal semantics and an interactive proof environment for object-oriented specifications.

A FRAMEWORK FOR OBJECT-ORIENTED SPECIFICATION

3

In this chapter, we present the key concepts of our framework for objected-oriented specifications. The complete formalization is over 500 pages long and contains several thousand definitions and theorems. This formalization, including all technical details, is presented in a separate document [21]. The foundations for our framework are motivated by the UML/OCL scenario, i. e., we model an object-oriented system supporting subtyping using single inheritance. Further, we model an object-oriented constraint language for specifying state transition using invariants, preconditions and postconditions. As we have to cope with undefined elements, e. g., path expressions that are invalid in a specific system state, it seems natural to base our specification language on a three-valued logic. Moreover, we aim for an embedding that supports the extension of existing data models without the need of re-proving everything and thus breaking up the closed-world assumption that is present many in state of the art proof environments.

3.1 CHALLENGES

We aim for a shallow embedding that captures the essence of object-orientation as it is understood in the object-oriented community. For example, our framework should not only be object-based but truly object-oriented in sense of Section 2.1. In particular such an embedding in HOL must provide:

- Support for subtyping and inheritance in a type system that does not provide a notion of subtypes.
- A notion of *state*, i. e., a mapping of object-references to objects and its possible *state transitions*. *state* *state transitions*
- Support for expressions that contain operations that refer to a pair of pre-state and post-state $(\sigma, \sigma_{\text{pre}})$, as for example in OCL.

path expression

- Support for a semantic representation of *path expressions* for addressing objects within the object store (memory).

undefinedness

- Support for *undefinedness*. This is important, because any syntactically correct path expression can, for a specific system state, be semantically undefined. Moreover, this also allows for a consistent logical support for undefined expressions like $\frac{0}{0}$. But, in such a setting, many rules can only be applied for "defined" values. In comparison to a two-valued setting, this results in additional case splits and side-conditions for many proofs. These side-conditions must be established by *subcalculi*; for example, we need rules that infer facts like "if $a + b$ is defined, then a and b must be defined."

subcalculus

- A semantics for operation calls and invocations supporting late-binding, and, if possible, overriding in a setting without closed-world assumption, i. e., we aim for an extensible framework.

To meet these challenges we have to provide for the object-oriented constraint language:

primitive type

- An technique for defining the semantics of the *primitive types*, e. g., `Boolean`, `Integer`, `Set` and collection types such as `Set`.
- A technique for defining the semantics for the built-in operation, capturing the arithmetic, logic and collection theories.
- A technique for giving semantics for user-defined operations.

With respect to the underlying object-oriented data model, which results in a semantics for path expressions, we have to provide:

object structure

- A mechanism to generate formal theories of typed *object structures* associated to classes and their relationships (e. g., inheritance).
- A technique for giving semantics for user-defined operations in the context of classes, leading also to a formal semantics of path expressions.
- We have to bring both embeddings together. Among others, this includes the definition of a semantics for path expressions and also the definition of a semantics for invariants and operation specifications consisting of preconditions and postconditions.

Further, we aim for mechanisms providing modularization and extensibility in different ways:

- The object store should allow for *modular proofs*, i. e., one should be able to add new classes without the need of re-proving the properties of existing classes. This breaks the closed-world assumption normally made in analysis tools for object-oriented systems. In our view, as object-oriented systems are normally extensible supporting such open-world scenarios is a corner stone for an usable object-oriented proof environment. In particular, such systems can be, even after the analysis, extended later. *modular proof*
- The embedding of the object-oriented constraint language should be easily usable with another kind of object store. One idea is to use a Java-like object store allowing, e. g., a distinction between $\perp$ and the "null" or "void" reference (pointer).
- The object store should be usable without the specification language, e. g., for a programming language description that allows for method definitions associated to an operation. Thus, one could verify a method with respect to its operation specification in a Hoare-logic style of reasoning.

Each of the mentioned techniques and encoding mechanisms can be organized into *levels*. The core of these levels is formally defined by theory morphisms called *layers*. Figure 3.1 on the next page gives an overview of this modular architecture: on the first two levels, the encoding of the object store and the object-oriented constraint language can be used independently. For example, by providing an encoding of state-machines, one could provide a constraint language for constraining state-machines, on level 2. In more detail: *level* *layer*

Level 0: This level defines the ground work for the embeddings. It consists of two layers:

New Datatypes: In this layer, we define HOL types, in particular auxiliary types for classes.

Datatype Adaption: In this layer, the HOL datatypes are adapted as needed, e. g., we glue basic datatypes together to objects or extend all datatypes by a special "undefined" element.

In summary, this level defines both an extensible object store and the datatypes for the constraint language.

Level 1: This adapts the functional behavior and finalizes the embeddings. It consists of two layers:

Functional Adaption: This layer adapts and extends the functional behavior of our embeddings, e. g., it defines the strictness of operations and defines the semantics of operation invocations in the context of our object store.

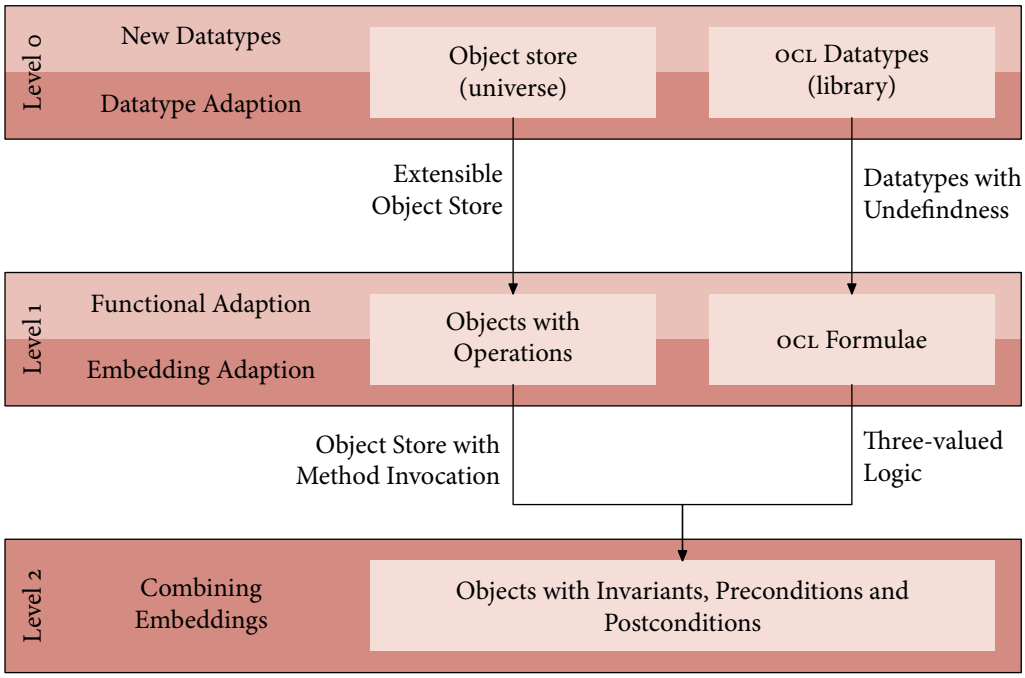


Figure 3.1: Structuring embeddings into functional layers and levels improves the reusability: after level 0 our architecture provides a rich library of datatypes supporting undefinedness, after level 1 we provide an object store with method invocation on the first hand and a three-valued logic on the other. Finally, we combine the two embeddings into a constrained object store with method invocation.

Embedding Adaption: This layer adds infrastructure for the treatment of contexts, i. e., the underlying pair of pre-state/post-state.

In summary, this level provides an embedding of an extensible object store with operation invocation, and constrained language (i. e., in the case of OCL, there are only OCL formulae without context declarations).

Level 2: This level combines the two embeddings, i. e., it introduces the context of the constraints and defines the semantics of objects and method invocations with respect to the validity of the corresponding preconditions, postconditions and invariants.

One can view level 1 objects as raw "structural" objects, while level 2 objects preserve the semantics of class invariants.

In the remainder of this chapter, we will explain our framework in more detail, using the architectural overview (Figure 3.1) as a kind of road-map. In Section 3.2, we will explain the concepts of theory morphism and show how theory morphism can be used for implementing the level structure. In Section 3.3, we show how our framework can be used for defining a constraint-language for object-oriented specifications, in particular, we present several semantic alternatives. By choosing a concrete semantics for these alternatives, one can define the semantics of concrete object-oriented constraint language, e. g., OCL. Our extensible encoding of object-oriented data-structures (object store) is explained in Section 3.4. In the remaining sections of this chapter, we explain how the constraint language and our object store interact, e. g., we explain our encoding of invariants and introduce statements into the constraint language that reason over the state of the object store.

3.2 THEORY MORPHISM AND STRUCTURING

Using a *conservative* embedding for defining semantics or for developing formal tools has, in comparison to an axiomatic approach, one disadvantage: one has to prove several thousand theorems for the object-language. At the first look, this large amount of proof requirements seems to make a conservative approach unfeasible for a language like OCL that comes with a rich library of datatypes. Thus it seems tempting to throw the conservatism over board and just postulate what we want. Nevertheless, in our opinion the consistency guarantee we gain from being conservative compensates for the toil. Moreover, we show in this section, how one can automate this work, and thus delegate some of this honest toil to the machine.

In particular we present an approach for deriving the mass of these theorems mechanically from the existing HOL library (our meta-language), i. e., based on the already proven theorems on the HOL level, we automatically try to prove similar properties for the meta-language. Our approach assumes a layered *theory morphism* mapping library types and library functions to new types and new functions of the constraint language (i. e., OCL) while uniformly modifying some semantic properties. The key idea is to represent the structure of the theory morphism by *semantic combinators* that are organized into layers (see Figure 3.1). Further, we introduce the concept of layered theory morphism to structure situations where a theory morphism can be decomposed in the application of several semantic combinators.

But first, we introduce the core notion of conservative theory morphism. Recall the notions for HOL as introduced in Section 2.2.2, i. e., the set of types τ, the set of type classes or sorts ξ, and the set of type constructors χ. We also introduced the inductively defined set of terms Λ, built over

the set of constants C and the set of variables V. Moreover, we introduce

type arity

a *type arity* ar, i. e., a finite mapping from type constructors to non-empty lists of sorts $ar : \chi \Rightarrow \mathrm{list}_{\geq 1}(\xi)$, where $\mathrm{list}_{\geq 1}$ denotes the type constructor for non-empty lists.

signature
environment

A *signature* is a quadruple $\Sigma = (\xi, \chi, ar, c :: C \Rightarrow \tau)$ and analogously the quadruple $\Gamma = (\xi, \chi, ar, v :: V \Rightarrow \tau)$ is called an *environment*.

typed term
typed formula

The following assumption incorporates a type inference and a notion of well-typed term: we assume a subset of terms called *typed terms* (written $\Lambda_{\Sigma,\Gamma} \subseteq \Lambda$) and a subset of typed terms, called *typed formulae* (written $F_{\Sigma,\Gamma} \subseteq \Lambda_{\Sigma,\Gamma}$); we require that in these notions, ar, ξ and χ agree in Σ and Γ. For example, the set of formulae can be defined as the set of typed terms of type bool.

theory
theory closure

We call $S = (\Sigma, A)$ with the axioms $A \subseteq F_{\Sigma,\Gamma}$ a *theory*. The following assumption incorporates an inference system: with a *theory closure* $\mathit{Th}(S) \subseteq F_{\Sigma,\Gamma}$ we denote the set of formulae derivable from A; in particular, we require $A \in \mathit{Th}(S)$ and Th to be monotonous in the axioms, i. e., $S \subseteq S' \to \mathit{Th}(S) \subseteq \mathit{Th}(S')$ (we also use $S \subseteq S'$ for the extension of subsets on tuples for component-wise set inclusion).

signature morphism
theory morphism

A *signature morphism* is a mapping $\Sigma \Rightarrow \Sigma$ which can be naturally extended to a *specification morphism* and a *theory morphism*.

In the following, the concepts of conservative theory morphisms are rephrased more abstractly. If S' is the constructed by extending S conservatively, we write

$$S' = S \uplus E\,; \tag{3.1}$$

where E is either a constant definition or a type definition (as introduced in Section 2.2.2). Recall that $S' = ((\xi, \chi', ar', C'), A')$ is defined as follows: We assume $S = ((\xi, \chi, ar, C), A)$, and $P(x)$ of type $P :: R \Rightarrow \mathrm{bool}$ for a base type R in χ. C' is constructed from C by adding $\mathrm{Abs}_T : R \Rightarrow T$ and $\mathrm{Rep}_T :: T \Rightarrow R$. χ' is constructed from χ by adding the new type T (i. e., which is supposed to be not in χ). The axioms A' is constructed by adding the two isomorphism axioms

$$\forall x.\ \mathrm{Abs}_T(\mathrm{Rep}_T\, x) = x \tag{3.2a}$$

and

$$\forall x.\ P\, x \to \mathrm{Rep}_T(\mathrm{Abs}_T\, x) = x \tag{3.2b}$$

to the set A, i. e.:

$$A' = A \cup \left\{ \begin{array}{c} \forall x.\ \mathrm{Abs}_T(\mathrm{Rep}_T\, x) = x\,, \\ \forall x.\ P\, x \to \mathrm{Rep}_T(\mathrm{Abs}_T\, x) = x \end{array} \right\}. \tag{3.3}$$

The type definition is conservative if the proof obligation $\exists x.\ P\, x$, holds.

Technically, conservative language embeddings are represented as *specification increments E* that contain the type definitions and constant definitions for the language elements and give a semantics in terms of a specification *S*.

specification increment

We use this possibility of extending a theory piecewise for structuring a theory morphism into layers. The main idea is to collect similar extension into the same layer. Formally, we define a layered theory morphism as follows:

DEFINITION 3.1 (LAYERED THEORY MORPHISM) A theory morphism is called a *layered theory morphism* if and only if in each form of conservative extension the following decomposition into elementary theory morphisms (*layers*) is possible:

layered theory morphism

1. for type synonyms $(\alpha_1, \ldots, \alpha_m)T$, there must be type constructors C_1 to C_n such that

$$(\alpha_1, \ldots, \alpha_m)T = C_n\Big(\cdots\big(C_1(T')\big)\Big), \tag{3.4}$$

2. for conservative type definitions $(\alpha_1, \ldots, \alpha_m)T$, there must be functions C_1 to C_n such that

$$(\alpha_1, \ldots, \alpha_m)T = \Big\{x :: C_n\Big(\cdots\big(C_1(T')\big)\Big) \;\Big|\; P(x)\Big\}, \text{ and} \tag{3.5}$$

3. for constant definitions c, there must be functions E_1 to E_n such that

$$c = (E_n \circ \cdots \circ E_1)(c'); \tag{3.6}$$

where each C_i or E_i are type constructors or expressions build from *semantic combinators* of layer L_i and the type expression T' is built from the meta-logic. Also c' is a construct from the meta-logic. A *layer* L_i is represented by a specification defining the *semantic combinators*, i. e., constructs that perform the semantic transformation from meta-level definitions to object-level definitions. □

Figure 3.2 on the following page illustrates the concepts of a structured theorem morphism: based on a meta-language (e. g., HOL) with datatypes the semantics of the object-language is defined, whereby common adaption techniques are exploited and are classified into different adaptions. Namely, we distinguish datatype adaption, the functional adaption, and the embedding adaption.

In the following, we will present a collection of layers and their combinators in more detail. We will associate the semantic combinators one by one to the specific layers and collect them in a distinguished set *SemCom*.

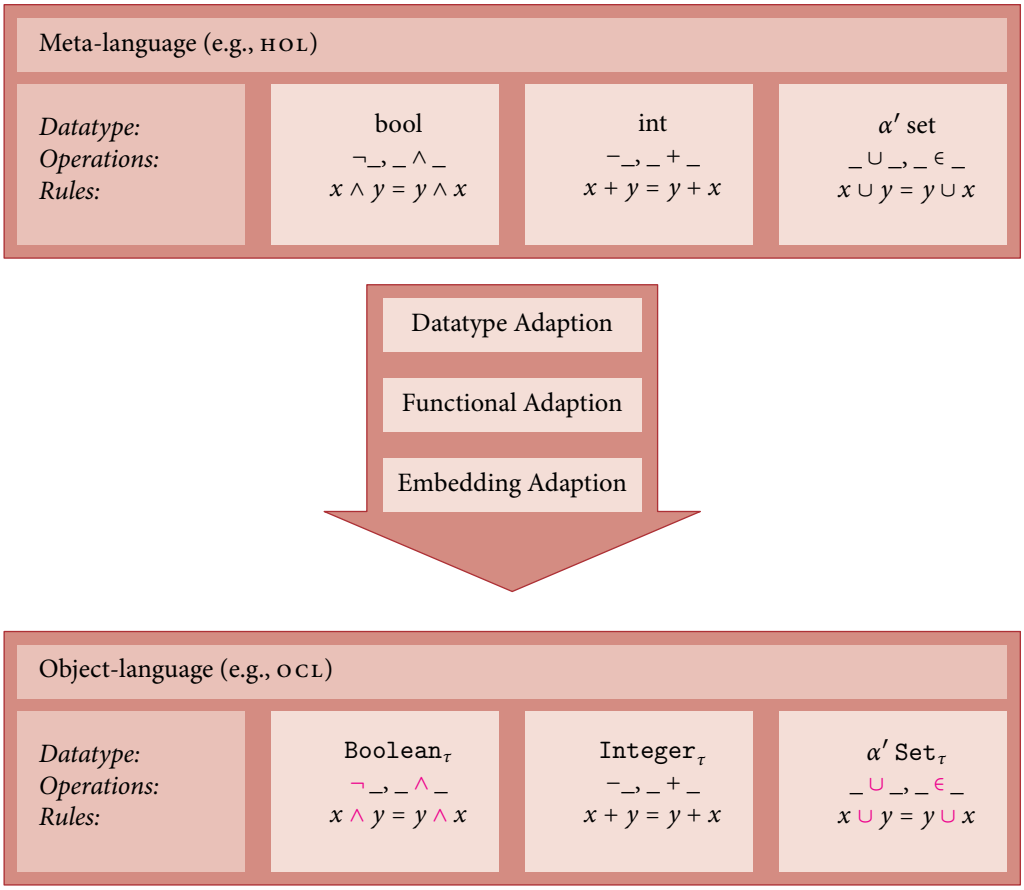


Figure 3.2: A structured theory morphism allows for mechanically deriving libraries of theorems for the object-language (e. g., OCL) out of already proven theorem libraries for the meta-language (e. g., HOL).

3.2.1 *Datatype Adaption*

datatype adaption

The *datatype adaption* establishes the link between meta-level types and object-level types, and between meta-level constants and object-level constants. While meta-level definitions in libraries of existing theorem provers are usually optimized in a way that is most suitable for automatic proof support, object-level definitions are often tied to a particular computational model. Thus, a gap between these two has to be bridged. For example, in Isabelle/HOL, the head-function applied to an empty list is underspecified. In a typical executable object-language such as SML or Haskell, this function should be defined to yield an exception element. Datatype adaption copes with such failure elements, the introduction of boundaries (e. g., maximal and minimal numbers in usual machine representation of numbers), congruences on raw data (such as *smashing*; see below) and the introduction of additional semantic structure on a type such as being member of a specific type class.

semantic combinator
bot

Concepts like definedness and strictness play a major role in our framework. We capture them using *semantic combinators*. We used Isabelle's concept of a type class to specify the class of all types bot (written as $\alpha :: \text{bot}$) that contain the undefinedness element $\bot$. Additionally, we required from this class the postulate "all types must have one element different from the undefined value" to rule out certain pathological cases revealed during the proofs. For all types in this class, concepts such as *definedness*, i. e.,

definedness

$$\text{def}\,(x :: \alpha :: \text{bot}) \equiv (x \neq \bot) \tag{3.7}$$

or *strictness* of a function, i. e.,

strictness

$$\text{isStrict}\, f \equiv (f\ \bot) = \bot \tag{3.8}$$

are introduced.

Further, we use the type constructor $\tau_\bot$ that assigns to each type τ a type lifted by $\bot$. Since any type in HOL contains at least one element, each type $\tau_\bot$ is in fact in the type class bot. The function $\llcorner_\lrcorner :: \alpha \to \alpha_\bot$, also called "lift," denotes the injection, the function $\ulcorner_\urcorner :: \alpha_\bot \to \alpha$, also called "drop," its inverse for defined values.

$\llcorner_\lrcorner$
$\ulcorner_\urcorner$

We encode operations that refer to a pair of system states (σ, σ_{pre}) as functions from (σ, σ_{pre}) to their semantic value. Technically, this means that we have to *lift over the context* any function occurring in the semantics of our object-language, e. g., $_ \wedge _$ or $_ + _$. Therefore, we introduce the type synonym $V_\tau(\alpha)$ and some combinators that capture the semantic essence of context lifting. The type synonym $V_\tau(\alpha)$ is defined as follows:

$V_\tau(\alpha)$

$$V_\tau(\alpha) := \tau \Rightarrow \alpha\,. \tag{3.9}$$

For example, using these type constructors, we can define a lifted and context-aware variant of the type Boolean:

$$\texttt{Boolean} = \text{bool}_\perp \text{ , and} \tag{3.10}$$

$$\texttt{Boolean}_\tau = V_\tau(\texttt{Boolean})\,, \tag{3.11}$$

which precisely correspond to layer 0 and layer 1.

context lifting

On the expression level, *context lifting* combinators for the distribution of contexts are defined as follows:

$$\text{lift}_0\, f \equiv \lambda\, \tau.\ f \tag{3.12}$$

with type $\alpha \Rightarrow V_\tau(\alpha)$,

$$\text{lift}_1\, f \equiv \lambda\, x\ \tau.\ f\ (x\ \tau) \tag{3.13}$$

with type $(\alpha \Rightarrow \beta) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)$, and

$$\text{lift}_2\, f \equiv \lambda\, x\ y\ \tau.\ f\ (x\ \tau)\ (y\ \tau) \tag{3.14}$$

with type $\big([\alpha, \beta] \Rightarrow \gamma\big) \Rightarrow \big[V_\tau(\alpha), V_\tau(\beta)\big] \Rightarrow V_\tau(\gamma)$. The types of these combinators reflect their purpose: they "lift" operations from HOL to semantic functions that are operations on contexts.

context passing

Operations constructed by context lifting pass the context τ unchanged. We call an operation *context passing* if it satisfies exactly this property, which is expressed formally as follows:

$$\text{cp}(P) \equiv \big(\exists\, f.\ \forall\, x\ \tau.\ P\, x\ \tau = f\ (x\ \tau)\ \tau\big) \tag{3.15}$$

with type $\big(V_\tau(\alpha) \Rightarrow V_\tau(\beta)\big) \Rightarrow \text{bool}$. Context invariance of expressions will turn out to be a key concept allowing for converting an equivalence on object-logic expressions (constraints) into a congruence; thus cp will play a major role in subcalculi for our object-oriented constraint language.

Definition 3.2 (Datatype Adaption Combinator) The combinators for the datatype adaption are semantic combinators, i. e., they are included in the set *SemComSemCom*:

$$\big\{_\!\perp, V_(_), \llcorner_\lrcorner, \ulcorner_\urcorner, \text{lift}_0, \text{lift}_1, \text{lift}_2, \text{lift}_3\big\} \subseteq \mathit{SemCom}\,. \tag{3.16}$$

□

smashing

As another example for a congruence construction, we will show the *smashing* on sets like data structures. For a language with a semantic domain providing $\perp$-element it is not clear, how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can define

$$(\perp, x) \neq \perp \qquad\qquad \{a, \perp, b\} \neq \perp \tag{3.17}$$

or we can define

$$(\bot, x) = \bot \qquad\qquad \{a, \bot, b\} = \bot\,. \tag{3.18}$$

The variant in Equation 3.18 is called *smashed product* and *smashed set*. The constant definition for the semantic combinator for *smashing* reads as follows:

$$\text{smash}\ f\ x \equiv \text{if}\ f\ \bot\ x\ \text{then}\ \bot\ \text{else}\ x \tag{3.19}$$

with type $\big[[\beta :: \text{bot}, \alpha :: \text{bot}] \Rightarrow \text{bool}, \alpha\big] \Rightarrow \alpha$. On this basis, the type `Set`, for example, is built via the type definition

$$\alpha\ \texttt{Set} = \Big\{x :: (\alpha :: \text{bot set}_\bot)\Big|\big(\text{smash}(\lambda\ x\ X.\ \text{def}\ X \wedge x \in \ulcorner X \urcorner)\ X\big)\ X = X\Big\} \tag{3.20}$$

and the type synonym

$$\alpha\ \texttt{Set}_\tau := V_\tau(\alpha\ \texttt{Set})\,. \tag{3.21}$$

Alternatively, we could define smashed sets directly:

$$\alpha\ \texttt{Set} = \Big\{x :: (\alpha :: \text{bot set}_\bot)\Big|\bot \notin \ulcorner x \urcorner\Big\}\,. \tag{3.22}$$

This representation is easier to read whereas the representation in Equation 3.20 is based on our layered theory morphism. Therefore we prefer the representation of Equation 3.20 as it makes the automatic derivation of theorems over smashed sets easier.

Overall, this quotient construction for smashed data structures identifies all sets containing $\bot$ in one class which is defined to be the $\bot$ of the type α set. All other sets were injected into an own class. Thus, using the (overloaded) constant definition

$$\bot \equiv \text{Abs}_{\text{Set}}\ \bot \tag{3.23}$$

we embed smashed sets into the class bot. The injection Abs_{Set} (together with the projection Rep_{Set}) is a consequence of the conservative type definition above (cf. Section 2.2.2).

For nested collection types such as `Set(Set(Integer))`, the HOL type is `Integer Set` $\texttt{Set}_\tau$ and not $\texttt{Integer}_\tau\ \texttt{Set}_\tau\ \texttt{Set}_\tau$ since context lifting is only necessary on the topmost level for each argument of an operation.

DEFINITION 3.3 (DATATYPE ADAPTION COMBINATOR (SET)) The combinators for the datatype adaption of sets are semantic combinators, i. e., they are included in the set *SemComSemCom*:

$$\big\{\text{smash}, \bot :: \alpha\ \texttt{Set}, \text{Abs}_{\text{Set}}, \text{Rep}_{\text{Set}}\big\} \subseteq \mathit{SemCom}\,. \tag{3.24}$$

□

Similar definitions can be derived for other set-like data structures like multisets (bags) or sequences, see [21] for the formal details of these definitions.

3.2.2 *Functional Adaption*

functional adaption

The *functional adaption* is concerned with the semantic transformation of a meta-level function into an object-level operation. Functional adaption may involve, for example, the

- *strictification* of an operation, i. e., its result is undefined if one of its arguments is undefined, or
- *late-binding invocation* semantics for operations. This semantic conversion process is necessary for converting a function into an operation using supporting overriding.

Technically, this is achieved by the strictify combinator. Overriding and late-binding can be introduced by the combinators invoke and invokeS described in this section.

strictify

We define a combinator strictify by

$$\text{strictify}\ f\ x \equiv \text{if}\ x = \bot\ \text{then}\ \bot\ \text{else}\ f\ x \tag{3.25}$$

with type $(\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \alpha \Rightarrow \beta$. The operator strictify yields a strict version of an arbitrary function f defined over the type class bot.

For example, we can define a strictified version of the union operator of HOL over the smashed type α `Set` as follows:

$$\text{union} \equiv \text{strictify}\big(\lambda\, x.\ \text{strictify}(\lambda\, y.\ \text{Abs}_{\text{Set}_\perp}\ulcorner\text{Rep}_{\text{Set}}\, x\urcorner \cup \ulcorner\text{Rep}_{\text{Set}}\, y\urcorner_\perp)\big) \tag{3.26}$$

with type $[(\alpha :: \text{bot})\ \texttt{Set}, \alpha\ \texttt{Set}] \Rightarrow \alpha\ \texttt{Set}$.

The treatment of late-binding (see Section 2.1.1) requires a particular pre-compilation step concerning the declaration of overridden methods discussed in Section 3.10 in more detail; in this section, we will concentrate on the caller aspect of method invocations, i. e., how to represent sub-expressions occurring in post conditions representing an invocation of a user-defined operation specification for method

$$m(a_1 :: t_1, \ldots, a_n :: t_n) :: t \tag{3.27}$$

as feature of class A. In an invocation, e. g., a sub-expression of the form $self.m(a_1, \ldots, a_n)$, the semantic value of the dynamic type of *self* is detected. This dynamic type helps to look-up the concrete operation specification in a look-up table. This specification can be turned into a function (by picking *some* function satisfying the specification), which is applied to *self* as first argument together with the other arguments. This semantics for *invoke* is captured in the n-indexed family of semantic combinators invoke (non-strict invocation) and invokeS (strict invocation); the former define call-by-name semantics, the latter call-by-value semantics. Since

invoke

invokeS is in principle only a strictified version of invoke we only explain invoke.

The invoke-combinator is defined for the case $n = 1$, for example, as follows:

$$\text{invoke}\ C\ tab\ a\ result \equiv \lambda\,\tau.$$
$$\begin{cases} \text{arbitrary} & \text{if } tab\ (\text{Least}\ x.\ x \in \text{dom}\ tab \wedge C(a\ \tau) \in x) = \text{None}; \\ f\ a\ result\ \tau & \text{if } tab\ (\text{Least}\ x.\ x \in \text{dom}\ tab \wedge C(a\ \tau) \in x) = \text{Some}\ f\,. \end{cases} \tag{3.28}$$

Here, Least is a HOL operator selecting the least set of a set of sets, that satisfies a certain property. In this case, this property is that *self* is contained in one of the domains of the look-up table OpTab_m generated during the processing of the declaration that will be discussed in Section 3.10), i. e., some set (of objects) characterizing a type. For such an element of the domain of the look-up table, the specification of the operation is selected and returned. Recall that Some _ is a constructor of the datatype α option.

The casting function C will be instantiated by a suitable coercion of a dynamic type to a class type to be discussed later (see Section 3.4).

The process of selecting an arbitrary, but fixed function from a specification (i. e., a relation) is handled by the Call-combinator, see Section 3.9 for details. It is defined essentially as the context-lifting of the Hilbert-Operator $\varepsilon\, x.\ P\ x$ that just gives one result element satisfying P; if this does not exist, the Hilbert-Operator picks an arbitrary element of this type.

The semantic code for the call-by-value invocation $c.m(a_1, \ldots, a_n)$ is given by:

$$\text{Call}(\text{invokeS}\ \text{C}_{[\text{A}]}\ \text{OpTab}_m\ c\ a_1\ \ldots\ a_n) \tag{3.29}$$

where c is assumed as an object of class C and to have a subtype of A and $\text{C}_{[\text{A}]}$ is a casting-function that converts C objects to A objects.

DEFINITION 3.4 (FUNCTIONAL ADAPTION COMBINATOR) The combinators for the functional adaption are semantic combinators, i. e., we included in the set *SemComSemCom*:

$$\{\text{strictify}, \text{Call}, \text{invoke}, \text{invokeS}\} \subseteq \mathit{SemCom}\,. \tag{3.30}$$

□

3.2.3 *Embedding Adaption for Shallow Embedding*

embedding adaption

The semantic combinators for the *embedding adaptions* are related to the embedding technique itself, namely the *lifting over contexts*. Any function

f with type $t_1, \ldots, t_n \to t_{n+1}$ of the object-language has to be transformed to a function:

$$I[\![f]\!] \qquad \text{with type } \big[V_\tau(t_1), \ldots, V_\tau(t_n)\big] \Rightarrow V_\tau(t_{n+1})\,. \tag{3.31}$$

As an example for a binary function like the built-in operation _ ∪ _ (based on union operator defined in Section 3.2.2), we present its constant definition:

$$_ \cup _ \equiv \text{lift}_2\ \text{union} \quad \text{with type } \big[(\alpha :: \text{bot})\ \texttt{Set}_\tau, \alpha\ \texttt{Set}_\tau\big] \Rightarrow \alpha\ \texttt{Set}_\tau\,. \tag{3.32}$$

Summing up the intermediate results of the local theory morphisms (i. e., the layers) in the previous subsections, the definition of our running example _ ∪ _ is given directly by:

$$_ \cup _ \equiv \text{lift}_2\Big(\text{strictify}\big(\lambda\, x.\ \text{strictify}(\lambda\, y.\ \text{Abs}_{\text{Set}\,\llcorner}{}^{\ulcorner}\text{Rep}_{\text{Set}}\, x^{\urcorner} \cup {}^{\ulcorner}\text{Rep}_{\text{Set}}\, y^{\urcorner}{}_{\lrcorner})\big)\Big)\,. \tag{3.33}$$

One easily recognizes our standard definition scheme, having Abs_{Set} and Rep_{Set} as additional semantic combinators. During mechanical lifting of HOL theorems to theorems of the object-logic (such as $A \cup B = B \cup A$), these operators require proofs for the invariance of the underlying quotient constructions; i. e., in this example, it must be proved that the union on representations of object-logic sets will again be representations of an object-logic set, (i. e., HOL sets not containing $\bot$).

3.3 DEFINING AN OBJECT-ORIENTED CONSTRAINT LANGUAGE

object-oriented constraint language

In this section, we motivate that our framework can easily be used for defining different semantics for an *object-oriented constraint language*. Intuitively, we define an object-oriented constraint language as a language based on logic and set theory which is used for reasoning about object-oriented data structures. Overall, such a language must provide a logic that able to reason over path expressions and also basic datatypes like Integers and Set. For the different building blocks, e. g., the logic, we provide different alternative proposals for a semantics. These different semantics can be examined and compared formally within our framework. The semantics of a specific object-oriented constraint language, e. g., OCL, can be easily defined by choosing the one concrete semantics for each needed building block. In particular we show the definitions for different three-valued logics, the definitions for basic datatypes and the definitions for smashed and non-smashed sets.

3.3.1 *The Logical Core*

Expression such as $\frac{0}{0} = 42$ are neither true nor false in mathematics. One way to treat this, is by underspecification, i. e., in each model of this formulae, $\frac{0}{0}$ has another interpretation. Instead, we introduce undefinedness (denoted by $\bot$) into the object-logic of our framework. This decision allows for the explicit treatment of undefined expressions while preserving an extended, three-valued, form of the law of the excluded middle, i. e., a Boolean value is either true, false, or undefined. Thus, we introduce a test for being defined into our object-logic, which can be easily defined based on the concepts introduces in Section 3.2. In particular, then we only need to lift the already introduced definedness-test def _:

$$\partial\, x \equiv \mathrm{lift}_{1\llcorner}\mathrm{def}\, x_{\lrcorner} \qquad \text{with type } \mathit{Val}_\tau(\alpha) \Rightarrow \mathtt{Boolean}_\tau. \tag{3.34}$$

As a shorthand, we also introduce a test for being undefined:

$$\not\partial\, x \equiv \neg\, \partial\, x \qquad \text{with type } \mathit{Val}_\tau(\alpha) \Rightarrow \mathtt{Boolean}_\tau. \tag{3.35}$$

Of course, both functions are non-strict, e. g., $\partial \bot$ is defined and, in particular, evaluates to f.

Following the overall scheme for operations, already presented in Section 3.2, one can also require strictness for the logic:

strict three-valued logic

DEFINITION 3.5 (STRICT THREE-VALUED LOGIC) For the *strict three-valued logic*, the semantics of the connectives are defined by the following truth tables:

$\dot\wedge$	f	t	$\bot$
f	f	f	$\bot$
t	f	t	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\dot\vee$	f	t	$\bot$
f	f	t	$\bot$
t	t	t	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\dot\longrightarrow$	f	t	$\bot$
f	t	t	$\bot$
t	f	t	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\neg$	
f	t
t	f
$\bot$	$\bot$

□

The truth tables in Definition 3.5 reveal that the binary operations of a strict three-valued logic are associative, commutative, and idempotent. Following the already described definition scheme for strict operations, this strict three-valued logic can be easily defined within our framework, see Table 3.1 for details.

lazy three-valued logic

DEFINITION 3.6 (LAZY THREE-VALUED LOGIC) The semantics of the logical connectives of the *lazy three-valued logic* are defined by the following truth tables:

$\vec\wedge$	f	t	$\bot$
f	f	f	f
t	f	t	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vec\vee$	f	t	$\bot$
f	f	t	$\bot$
t	t	t	t
$\bot$	$\bot$	$\bot$	$\bot$

$\vec\longrightarrow$	f	t	$\bot$
f	t	t	t
t	f	t	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\neg$	
f	t
t	f
$\bot$	$\bot$

□

$$\neg_ \equiv \mathrm{lift}_1\ \mathrm{strictify}(\lambda\, x.\ \llcorner \neg \ulcorner x \urcorner \lrcorner) \tag{3.36a}$$
$$_ \mathbin{\dot\wedge} _ \equiv \mathrm{lift}_2\big(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner \ulcorner x \urcorner \wedge \ulcorner y \urcorner \lrcorner))\big) \tag{3.36b}$$
$$_ \mathbin{\dot\vee} _ \equiv \lambda\, x\ y.\ \neg(\neg x \mathbin{\dot\wedge} \neg y) \tag{3.36c}$$
$$_ \mathbin{\dot\longrightarrow} _ \equiv \lambda\, x\ y.\ \neg x \mathbin{\dot\vee} y \tag{3.36d}$$

Table 3.1: The definitions for a *strict three-valued logic* follow the general scheme for strict operations. The binary operations of a strict three-valued logic are associative, commutative and idempotent.

$$\neg_ \equiv \mathrm{lift}_1\ \mathrm{strictify}(\lambda\, x.\ \llcorner \neg \ulcorner x \urcorner \lrcorner) \tag{3.37a}$$
$$\begin{aligned}_ \mathbin{\vec\wedge} _ \equiv \mathrm{lift}_2\ \big(\lambda\, x\ y.\ &\mathrm{if}\ (\mathrm{def}\ x)\\ &\mathrm{then\,if}\ (\mathrm{def}\ y)\ \mathrm{then}_{\llcorner} \ulcorner x \urcorner \wedge \ulcorner y \urcorner \lrcorner\\ &\qquad\qquad \mathrm{else\,if} \ulcorner x \urcorner\ \mathrm{then}\ \bot\ \mathrm{else}_{\llcorner} \mathrm{false} \lrcorner\\ &\mathrm{else}\ \bot\end{aligned} \tag{3.37b}$$
$$_ \mathbin{\vec\vee} _ \equiv \lambda\, x\ y.\ \neg(\neg x \mathbin{\vec\wedge} \neg y) \tag{3.37c}$$
$$_ \mathbin{\vec\longrightarrow} _ \equiv \lambda\, x\ y.\ \neg x \mathbin{\vec\vee} y \tag{3.37d}$$

Table 3.2: A *lazy three-valued logic* is often used by evaluation environments and programming languages. Due to the fact that algebraic properties like the commutativity of the conjunction do not hold, it is not well-suited for formal reasoning.

Note, that the truth tables are not symmetric which indicates that the binary operations are not commutative e. g., $\mathtt{f} \mathbin{\vec\wedge} \bot$ evaluates to $\mathtt{f}$, whereas $\bot \mathbin{\vec\wedge} \mathtt{f}$ evaluates to $\bot$. Obviously, in a proof environment, a non-commutative "and" is not common and leads to complicated calculi. Nevertheless, it can easily be defined (see Table 3.2) and it also motivates the idea of canceling undefinedness, if the result of an operation can be uniquely determined from its defined arguments.

A logic supporting undefinedness, while preserving the usual algebraic properties was introduced by Kleene [66] in which he argues that the informal meaning for the third value ($\bot$) should be "unknown" or "undefined."

Strong Kleene Logic

Definition 3.7 (Strong Kleene Logic) The semantics of the connectives for the *Strong Kleene Logic* are defined by the following truth tables:

$$\neg_ \equiv \mathrm{lift}_1\ \mathrm{strictify}(\lambda x.\ \llcorner\neg\ulcorner x\urcorner\lrcorner) \tag{3.38a}$$

$$\begin{aligned}_ \wedge _ \equiv \mathrm{lift}_2\ \big(\lambda\, x\ y.\ &\text{if } (\mathrm{def}\ x)\\ &\text{then if } (\mathrm{def}\ y) \text{ then}_{\llcorner}\ulcorner x\urcorner \wedge \ulcorner y\urcorner\lrcorner\\ &\qquad\text{else if}\ulcorner x\urcorner \text{ then } \bot \text{ else}_{\llcorner}\mathrm{false}_{\lrcorner}\\ &\text{else if } (\mathrm{def}\ y) \text{ then if}\ulcorner y\urcorner \text{ then } \bot\\ &\qquad\text{else}_{\llcorner}\mathrm{false}_{\lrcorner} \text{ else } \bot\big)\end{aligned} \tag{3.38b}$$

$$_ \vee _ \equiv \lambda\, x\ y.\ \neg(\neg\, x \wedge \neg\, y) \tag{3.38c}$$

$$_ \longrightarrow _ \equiv \lambda\, x\ y.\ \neg\, x \vee y \tag{3.38d}$$

Table 3.3: The connectives of a Strong Kleene Logic have the usual lattice properties, while allowing reasoning over three-valued formulae where undefinedness is canceled whenever possible.

$\wedge$	f	t	$\bot$
f	f	f	f
t	f	t	$\bot$
$\bot$	f	$\bot$	$\bot$

$\vee$	f	t	$\bot$
f	f	t	$\bot$
t	t	t	t
$\bot$	$\bot$	t	$\bot$

$\longrightarrow$	f	t	$\bot$
f	t	t	t
t	f	t	$\bot$
$\bot$	$\bot$	t	$\bot$

$\neg$	
f	t
t	f
$\bot$	$\bot$

□

Of course, the connectives for the Strong Kleene Logic can be defined in our framework, see Table 3.3. Here we exploit that the implication can be defined using only the logical negation and disjunction, i. e.:

$$A \longrightarrow B \equiv \neg\, A \vee B\,. \tag{3.39a}$$

Moreover, for Strong Kleene Logics two other definitions for the implications are discussed in the literature [52, 56]:

$$A \overset{1}{\longrightarrow} B \equiv (\not\partial\, A) \vee (\neg\, A) \vee B \tag{3.39b}$$

and

$$A \overset{2}{\longrightarrow} B \equiv (\neg\, A) \vee (A \wedge B)\,. \tag{3.39c}$$

For a first comparison of these three definitions, we derived their truth tables, see Table 3.4. Notably, their behavior differs only for undefined operands. The difference is that the variant (3.39b) evaluates to t if the assumption is undefined, while (3.39c) evaluates to $\bot$.

Moreover, for all three logics, we can prove easily within our framework, using Isabelle/HOL, that the definitions given in Table 3.1, Table 3.2 and

$\longrightarrow$	f	$\bot$	t
f	t	t	t
$\bot$	$\bot$	$\bot$	t
t	f	$\bot$	t

$\overset{1}{\longrightarrow}$	f	$\bot$	t
f	t	t	t
$\bot$	t	t	t
t	f	$\bot$	t

$\overset{2}{\longrightarrow}$	f	$\bot$	t
f	t	t	t
$\bot$	$\bot$	$\bot$	$\bot$
t	f	$\bot$	t

Table 3.4: For Strong Kleene Logic, three different definitions of the implications are discussed in literature. Their truth tables show, that their behavior differs only for undefined operands.

Table 3.3 fulfill the corresponding truth tables given in the definitions; [21] presents the proofs in detail. Moreover, the definition of the negation $(\neg _)$ is in all three logics identical, i. e., it is in all three definitions a strict operation. Further it is interesting that in all three settings it is sufficient to define the negation and the conjunction, since the definitions for the disjunction and implications can then be expressed as negations and conjunctions.

3.3.2 *Primitive Datatypes*

Primitive datatypes, or value types (see Section 3.6), e. g., Boolean, Integer, String, are defined using the previous explained datatype adaption combinators. For example, recall the following definitions:

$$\begin{aligned} \texttt{Boolean} &:= \mathrm{bool}_{\bot}\,, \\ \texttt{Boolean}_{\tau} &:= V_{\tau}(\texttt{Boolean})\,, \\ \texttt{Integer} &:= \mathrm{int}_{\bot}\,, \text{ and} \\ \texttt{Integer}_{\tau} &:= V_{\tau}(\texttt{Integer})\,. \end{aligned}$$

The constant definitions for `t` (true), `f` (false), and $\bot$ (undefined) of $\texttt{Boolean}_{\tau}$ or `0`, `1`, …of $\texttt{Integer}_{\tau}$ are straight-forward, using the lifting combinators:

$$\begin{aligned} \bot &\equiv \mathrm{lift}_0(\llcorner \bot \lrcorner) && \text{with type } \texttt{Boolean}_{\tau}\,, \\ \texttt{t} &\equiv \mathrm{lift}_0(\llcorner \mathrm{true} \lrcorner) && \text{with type } \texttt{Boolean}_{\tau}\,, \\ \texttt{f} &\equiv \mathrm{lift}_0(\llcorner \mathrm{false} \lrcorner) && \text{with type } \texttt{Boolean}_{\tau}\,, \\ \bot &\equiv \mathrm{lift}_0(\llcorner \bot \lrcorner) && \text{with type } \texttt{Integer}_{\tau}\,, \\ \texttt{0} &\equiv \mathrm{lift}_0(\llcorner 0 \lrcorner) && \text{with type } \texttt{Integer}_{\tau}, \text{ and} \\ \texttt{1} &\equiv \mathrm{lift}_0(\llcorner 1 \lrcorner) && \text{with type } \texttt{Integer}_{\tau}\,. \end{aligned}$$

The definition for undefinedness is done for the *polymorphic constant* $\bot$.

For Boolean, we have already seen the definitions for the basic operations. The operations for the primitive datatypes are very similar to these,

e. g., the strict addition on integers is defined as:

$$_ + _ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner \ulcorner x \urcorner + \ulcorner y \urcorner \lrcorner))) \,. \tag{3.40}$$

Moreover, from these definitions, computational rules on numbers can be derived, which perform computations like $3+4$ on binary representations of numbers.

Adding an explicit element denoting undefinedness to the "mathematical" integers and real numbers, i. e., lifting them, changes their algebraic structure. Whereas mathematical integers are a commutative ring with unity and real numbers are a field, the lifted versions do not share such a rich structure:

- The lifted integers ($\mathtt{Integer}_\tau$) contain neither and additive nor a multiplicative inverse element for $\bot$. Both $(\mathtt{Integer}_\tau; +)$ and $(\mathtt{Integer}_\tau; \cdot)$ form only a half-group with unity (monoid), and thus $(\mathtt{Integer}_\tau; +; \cdot)$ does not even form a semiring ($a \cdot 0 = 0$ does not hold). Nevertheless, $(\mathtt{Integer}_\tau \setminus \{\bot\}; +; \cdot)$ is a subring of $(\mathtt{Integer}_\tau; +; \cdot)$.
- Similarly, $(\mathtt{Real}_\tau; +; \cdot)$ do not form a semiring either, but contain the subfield $(\mathtt{Real}_\tau \setminus \{\bot\}; +; \cdot)$.

For the real numbers ($\mathtt{Real}_\tau$), this seems to be a dramatic loss of algebraic structure. But one should keep in mind that still most laws required for rings (in the case of $\mathtt{Integer}_\tau$) and fields ($\mathtt{Real}_\tau$) hold. For example, Table 3.5 summarizes the variants of the ring laws that can be proven for $\mathtt{Real}_\tau$, only Equation 3.41d and Equation 3.41h differ from the usual laws for fields. The situation for $\mathtt{Integer}_\tau$ is similar. Moreover, the idea of canceling undefinedness whenever possible, e. g., by defining $\bot \cdot \mathtt{0}$ to be $\mathtt{0}$, is not sufficient for establishing a richer algebraic structure.

Another interesting possibility is the definition of "machine arithmetic," i. e., a bounded integers based on two's-complement representation. Rauch and Wolff [99] presents such a formal semantics for bounded integers, based on the specification of the Java virtual machine (JVM), as a shallow embedding into Isabelle/HOL. This work would fit nicely into our framework.

3.3.3 *Collections*

Using our framework, it is easy to define both a theory of non-smashed collection types and a theory of smashed collection types. The underlying datatype definitions are already described in Section 3.2.1. As an example, see Table 3.6 for the definition of the core operations for non-smashed sets. Introducing the injection $\mathrm{Abs}_{\mathrm{Set}}$ and projection $\mathrm{Rep}_{\mathrm{Set}}$ results in the corresponding definitions for the smashed sets (see Table 3.7).

$$\frac{}{(x_0 + x_1) + (x_2 :: \mathtt{Real}_\tau) = x_0 + (x_1 + x_2)} \tag{3.41a}$$

$$\frac{}{0 + (x_0 :: \mathtt{Real}_\tau) = x_0} \tag{3.41b}$$

$$\frac{}{(x_0 :: \mathtt{Real}_\tau) + x_1 = x_1 + x_0} \tag{3.41c}$$

$$\frac{(x :: \mathtt{Real}_\tau) \neq \bot}{\exists y.\ x + y = 0} \tag{3.41d}$$

$$\frac{}{(x_0 \cdot x_1) \cdot (x_2 :: \mathtt{Real}_\tau) = x_0 \cdot (x_1 \cdot x_2)} \tag{3.41e}$$

$$\frac{}{1 \cdot (x_0 :: \mathtt{Real}_\tau) = x_0} \tag{3.41f}$$

$$\frac{}{x_0 \cdot (x_1 :: \mathtt{Real}_\tau) = x_1 \cdot x_0} \tag{3.41g}$$

$$\frac{(x :: \mathtt{Real}_\tau) \neq \bot \qquad x \neq 0}{\exists y.\ x \cdot y = 1} \tag{3.41h}$$

$$\frac{}{(x_0 :: \mathtt{Real}_\tau) \cdot (x_1 + x_2) = x_0 \cdot x_1 + x_0 \cdot x_2} \tag{3.41i}$$

$$\frac{}{(x_0 + x_1) \cdot (x_2 :: \mathtt{Real}_\tau) = x_0 \cdot x_2 + x_1 \cdot x_2} \tag{3.41j}$$

Table 3.5: The lifted real numbers ($\mathtt{Real}_\tau$) enjoy not all laws required for fields: there is neither an inverse element for the addition (Equation 3.41d) nor for the multiplication (Equation 3.41h).

$$_ \in _ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda X.\ \mathrm{strictify}(\lambda x.\ \llcorner x \in \ulcorner X \urcorner \lrcorner))) \tag{3.42a}$$

$$_^{-1} \equiv \mathrm{lift}_1(\mathrm{strictify}(\lambda X.\ \llcorner -(\ulcorner X \urcorner - \bot) \lrcorner)) \tag{3.42b}$$

$$_ \cup _ \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \llcorner \ulcorner X \urcorner \cup \ulcorner Y \urcorner \lrcorner)\big)\Big) \tag{3.42c}$$

$$_ \cap _ \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \llcorner \ulcorner X \urcorner \cap \ulcorner Y \urcorner \lrcorner)\big)\Big) \tag{3.42d}$$

Table 3.6: Defining the basic operation on *non-smashed sets* straight forward. Non-strict version can be obtained by omitting the strictify combinator.

$$_ \in _ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda X.\ \mathrm{strictify}(\lambda x.\ \llcorner x \in \ulcorner \mathrm{Rep}_{\mathrm{Set}}\ X \urcorner \lrcorner))) \tag{3.43a}$$

$$_^{-1} \equiv \mathrm{lift}_1(\mathrm{strictify}(\lambda X.\ \mathrm{Abs}_{\mathrm{Set}\llcorner} - (\ulcorner \mathrm{Rep}_{\mathrm{Set}}\ X \urcorner - \bot) \lrcorner)) \tag{3.43b}$$

$$_ \cup _ \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \mathrm{Abs}_{\mathrm{Set}\llcorner} \ulcorner \mathrm{Rep}_{\mathrm{Set}}\ X \urcorner \cup \ulcorner \mathrm{Rep}_{\mathrm{Set}}\ Y \urcorner \lrcorner)\big)\Big) \tag{3.43c}$$

$$_ \cap _ \equiv \mathrm{lift}_2\Big(\mathrm{strictify}\big(\lambda X.\ \mathrm{strictify}(\lambda Y.\ \mathrm{Abs}_{\mathrm{Set}\llcorner} \ulcorner \mathrm{Rep}_{\mathrm{Set}}\ X \urcorner \cap \ulcorner \mathrm{Rep}_{\mathrm{Set}}\ Y \urcorner \lrcorner)\big)\Big) \tag{3.43d}$$

Table 3.7: The definition for the basic operation on *smashed sets* can be easily derived from the definition for non-smashed sets (see Table 3.6) by introducing the injection $\mathrm{Abs}_{\mathrm{Set}}$ and projection $\mathrm{Rep}_{\mathrm{Set}}$.

In a similar way, we also could define datatype adaption combinators for *finite sets*, albeit this is not necessary: using a suitable definition for the size of an unbounded set, i. e., for an infinite set the size is undefined, we can easily define a test for sets being finite

$$\mathrm{isFinite}\ \mathit{self} \equiv \partial \| \mathit{self} \| \qquad \text{of type } \alpha\ \mathtt{Set}_\tau \Rightarrow \mathtt{Boolean}_\tau. \tag{3.44}$$

In assumptions, this test function isFinite _ can be easily used for restricting theorems to finite sets.

3.4 FORMALIZING OBJECT-ORIENTED DATA STRUCTURES

Systems for formally analyzing object-oriented systems can be mainly classified into systems based on a closed-world assumption (the majority) and systems based on an open-world-assumption:

Closed-world: In a *closed-world* scenario, a fixed data model is assumed, i. e., a fixed set of classes that cannot be extended. In particular, it is neither possible to introduce new top-level classes nor to inherit from an existing class. *closed-world*

Open-world: In an *open-world* scenario, the underlying data-mode is extensible, i. e., new classes can be introduced into the system and these new classes can, in particular, override already defined operations or methods. *open-world*

Extensibility is both a key feature of object-orientation, and also a corner stone of modular theorem proving. Therefore, we aim for an embedding

supporting extensibility. Moreover, the construction we present in this section allows for open-worlds, closed-worlds and even partially closed-worlds.

Instead of constructing a "universe of all objects" (which is either untyped or "too large" for a (simply) typed set theory, where all type sums must be finite), one could think of generating an object universe for each given set of classes. For example, assume a model with the classes A, B, and C. Moreover, we ignore subtyping and inheritance for a moment. In this case, we would construct the universe $\mathcal{U}^0 = A + B + C$. Unfortunately, such a construction is not extensible: If we add a new class, e. g., D, then the "obvious" construction $\mathcal{U}^1 = A + B + C + D$ results in a type that is *different* to the type $\mathcal{U}^0$. Thus, the two types $\mathcal{U}^0$ and $\mathcal{U}^1$ (and all values constructed over them) are incomparable. Therefore, such a representation rules out a modular, incremental construction of larger object systems. In particular, properties that have been proven over $\mathcal{U}^0$ will not hold over $\mathcal{U}^1$. Practically, this means that all proof scripts will have to be rerun over an extended universe.

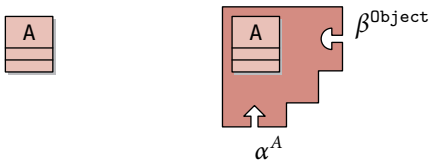
We solve this problem by using parametric polymorphisms for representing *families* for universes $\mathcal{U}^i$, see Figure 3.3 on the next page for a first overview of this idea. Such a family of universes represents the "possible class diagram extensions". Further, we extend the scheme sketched above by assigning to Classes not directly *objects*, but merely *object extensions*. This "incremental" object view (also used in many implementations) allows for the representation of object inheritance and leads, as we will see, to a smooth integration of inheritance into the world of parametric polymorphism.

3.4.1 *Foundations*

Object universes are the core of our notion of state, which is the building block of our notion of context τ, which is again the building block of the semantic domain of our expressions: $\tau \Rightarrow \alpha :: \text{bot}$. In this section, we focus on families of object universes $\mathcal{U}^i$, each of which corresponds to a class diagram. Each $\mathcal{U}^i$ comprises all *primitive types* (`Real`, `Integer`, `String`, `Boolean`, ...) and an extensible *class type representation* induced by a class hierarchy. To each class in a given class diagram, a *class type* is associated which represents the set of *object instances* or *objects*. The *structure* of a $\mathcal{U}^i$ is to provide a family of injections and projections to and from each class type. More precisely, if we assume a class A, this results on level 0 in:

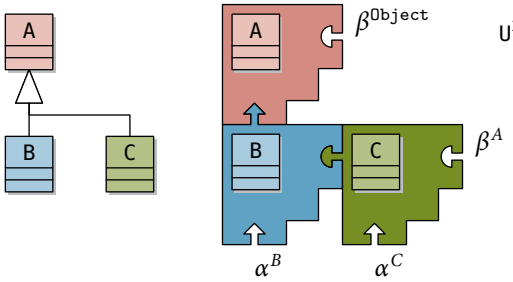
$$\text{mk}_A^{(0)} \qquad \text{with type } \mathcal{U}^i \Rightarrow A, \text{ and} \tag{3.45}$$

$$\text{get}_A^{(0)} \qquad \text{with type } A \Rightarrow \mathcal{U}^i. \tag{3.46}$$



$$\mathsf{U}^1_{(\alpha^A,\beta^{\texttt{Object}})} = A \times \alpha^A_\perp + \beta^{\texttt{Object}}$$

(a) A single class A represented by the type sum $A \times \alpha^A_\perp + \beta^{\texttt{Object}}$. The type variable $\alpha^A_\perp$ allows for introducing subclasses of A and the type variable $\beta^{\texttt{Object}}$ allows for introducing new top-level classes.



$$\mathsf{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})} = A \times \overbrace{(B \times \alpha^B_\perp + C \times \alpha^C_\perp + \beta^A)_\perp}^{=\alpha^A} + \beta^{\texttt{Object}}$$

(b) Extending the previous class model simultaneously with two direct subclasses of A is represented by instantiating the type variable α^A of $\mathcal{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$.

Figure 3.3: Assume we have a model consisting only of one class `A` which "lives" in the universe $\mathcal{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$ that we want to extend simultaneously with two new subclasses, namely `B` and `C`. As both new classes are derived from class `A`, we construct a local type polynomial $B \times \alpha^B_\perp + C \times \alpha^C_\perp + \beta^A$. This type polynomial is used for instantiating type variable α^A. This process results in the universe $\mathcal{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$ for the final class hierarchy. In particular, the universe $\mathcal{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$ is a type instance of $\mathcal{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$. Thus, properties that have been proven over the initial universe $\mathcal{U}^1_{(\alpha^A,\beta^{\texttt{Object}})}$ are still valid over the extended universe $\mathcal{U}^2_{(\alpha^B,\alpha^C,\beta^A,\beta^{\texttt{Object}})}$.

This functions allow us to inject any semantic value of our object-language into some $\mathcal{U}^i$. Note, as we need also lifted versions of these definitions, we will mark the different versions by different superscripts. This in turn makes a family of states (containing "object systems") possible:

$$\text{state} \qquad \text{with type oid} \rightharpoonup \mathcal{U} \tag{3.47}$$

where $_ \rightharpoonup _$ denotes the partial mapping. From this state, concrete values may be accessed via an oid and then be projected via $\text{get}_{\text{A}}^{(0)}$. On this basis, the accessor functions composing path expressions can be built.

The extensibility of a universe type is reflected by "holes" (polymorphic variables) that can be filled when "adding" extensions to class objects, which means adding subclasses to the class hierarchy. Our construction will ensure that $\mathcal{U}^{i+1}$ (corresponding to a particular class diagram) is just a type instance of $\mathcal{U}^i$ (where $\mathcal{U}^{(i+1)}$ is constructed by adding new classes to $\mathcal{U}^i$). Thus, properties proven over object systems "living" in $\mathcal{U}^i$ remain valid in $\mathcal{U}^{i+1}$.

There are essentially two choices for an operational semantics of object universes and thus object constructors:

Objects are references: every object has a *unique identifier*, sometimes also called *reference*. This construction is well-known from many widely used object-oriented programming languages like Java or Eiffel or specification languages with *copy semantics*

copy semantics

Objects are values: the identifying representation of an objects is its value, i. e., two objects representing the same value, are indistinguishable. This is often called a *sharing semantics*.

sharing semantics

Naturally, the choice if objects identifiers are used to identify objects has several consequences on the meaning of "being equal." This is discussed in more detail in Section 3.6. Having object-oriented programming language in mind, a non-referential setting can be unintuitive. For example, a constructor does not necessarily generate a fresh object.

In our framework, the distinction between the sharing semantics and copy semantics is reflected in two alternative universe constructions, namely the *non-referential universe* and the *referential universe*.

3.4.2 *Type Constructions*

In the following, we define several type sets which all are subsets of the types of the HOL type system. This set, although denoted in usual set-notation, is a meta-theoretic construct, i. e., it cannot be formalized in HOL. We start by defining all possible types for class attributes.

attribute type

DEFINITION 3.8 (ATTRIBUTE TYPE) The set $\mathfrak{A}$ of *attribute types* is inductively defined as follows:

1. $\{\texttt{Boolean}, \texttt{Integer}, \texttt{Real}, \texttt{String}, \text{oid}\} \subset \mathfrak{A}$, and
2. $\{a\ \texttt{Set}, a\ \texttt{Sequence}, a\ \texttt{Bag}, a\ \texttt{OrderedSet}\} \subset \mathfrak{A}$ for all $a \in \mathfrak{A}$. □

Attributes with class types are encoded using the type oid. These object identifiers (i. e., references) will be resolved by accessor functions for a given state; an access failure will be represented by $\bot$.

Similar to the description in the OCL standard we represent a class by a tuple, which is built by pairing the attribute types of the class. Moreover, we extend this tuple by an abstract datatype for each class. This construction guarantees that each class type is unique. Thus we provide a strongly typed universe (with regard to the object-oriented type system).

DEFINITION 3.9 (TAG TYPE) For each class C we assign a *tag type* $t \in \mathfrak{T}$ which is just an abstract type that makes class types unique. The set $\mathfrak{T}$ is called the set of tag types. □ *tag type*

Further, we introduce for each class a base type:

DEFINITION 3.10 (BASE CLASS TYPE) The set of *base class types* $\mathfrak{B}$ is defined as follows: *base class type*

1. classes without attributes are represented by $(t \times \text{unit}) \in \mathfrak{B}$, where $t \in \mathfrak{T}$ and unit is a special HOL type denoting the empty product.
2. if $t \in \mathfrak{T}$ and $a_i \in \mathfrak{A}$ for $i \in \{0, \ldots, n\}$ then $(t \times a_0 \times \cdots \times a_n) \in \mathfrak{B}$.□

Without loss of generality, we assume in our object model a common supertype of all objects. In the case of OCL, this is `OclAny`. In the case of Java this would be `Object`. We can assume such a supertype without loss of generality, because such a common supertype can always be added to a given class structure without changing the overall semantics of the original object model.

DEFINITION 3.11 (OBJECT) Let $\text{Object}_{\text{tag}} \in \mathfrak{T}$ be the tag of the common supertype *Object* and oid the type of the object identifiers, *Object*

1. in the *non-referential* setting, we define

$$\alpha\ \texttt{Object} := (\text{Object}_{\text{tag}} \times \alpha_\bot) \tag{3.48}$$

   as the common supertype.

2. in the *referential* setting, we define

$$\alpha\ \texttt{Object} := \big((\text{Object}_{\text{tag}} \times \text{oid}) \times \alpha_\bot\big) \tag{3.49}$$

   as the common supertype. □

Now we have all the foundations for defining the type of our family of universes formally:

universe type

DEFINITION 3.12 (UNIVERSE TYPE) The set of all *universe types* $\mathfrak{U}_{\text{ref}}$ and $\mathfrak{U}_{\text{nonref}}$ (abbreviated $\mathfrak{U}_{\text{x}}$) is inductively defined by:

1. $\mathcal{U}^0_\alpha \in \mathfrak{U}_{\text{x}}$ is the initial universe type with one type variable (hole) α.
2. $\mathcal{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)} \in \mathfrak{U}_{\text{x}}$, $n, m \in \mathbb{N}$, $i \in \{0,\ldots,n\}$ and $c \in \mathfrak{B}$ then
$$\mathcal{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)}\Big[\alpha_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}_{\text{x}}\,. \quad (3.50)$$
This definition covers the introduction of "direct object extensions" by instantiating α-variables.
3. $\mathcal{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)} \in \mathfrak{U}_{\text{x}}$, $n, m \in \mathbb{N}$, $i \in \{0,\ldots,m\}$, and $c \in \mathfrak{B}$ then
$$\mathcal{U}_{(\alpha_0,\ldots,\alpha_n,\beta_1,\ldots,\beta_m)}\Big[\beta_i := \big((c \times (\alpha_{n+1})_\perp) + \beta_{m+1}\big)\Big] \in \mathfrak{U}_{\text{x}}\,. \quad (3.51)$$
This definition covers the introduction of "alternative object extensions" by instantiating β-variables. □

The initial universe $\mathcal{U}^0_\alpha$ represents the common supertype (i. e., `Object`) of all classes, i. e., a simple definition would be

$$\mathcal{U}^0_\alpha := \alpha\ \texttt{Object}\,. \quad (3.52)$$

Along the class hierarchy, class types are type instances of the types of the superclasses.

Alternatively one can also encode values *Values* := `Real` + `Integer` + `Boolean` + `String` within the initial universe type, e. g.,

$$\mathcal{U}^0_\alpha := \alpha\ \texttt{Object} + \mathit{Values}\,. \quad (3.53)$$

As we will see later, for our framework we choose to represent also values within the universe which makes extensions possible that need to store values within the store. Thus we define the universes as follows:

referential universe

DEFINITION 3.13 (REFERENTIAL UNIVERSE TYPE) The universe type $\mathfrak{U}_{\text{ref}}$ of the *referential universe* is constructed using item 1 from Definition 3.11 in the definition of the initial universe:

$$\mathcal{U}^0_\alpha := \alpha\ \texttt{Object} + \texttt{Real} + \texttt{Integer} + \texttt{Boolean} + \texttt{String}\,. \quad (3.54)$$

□

DEFINITION 3.14 (NON-REFERENTIAL UNIVERSE TYPE) The type $\mathfrak{U}_{\text{nonref}}$ of the *non-referential universe* is constructed using item 2 from Definition 3.11 in the definition of the initial universe: *non-referential universe*

$$\mathcal{U}_\alpha^0 := \alpha\ \texttt{Object} + \texttt{Real} + \texttt{Integer} + \texttt{Boolean} + \texttt{String}. \tag{3.55}$$

□

We pick up the idea of a universe representation without values for a class with all its extensions (subtypes). We construct for each class a type that describes a class and all its subtypes. They can be seen as "paths" in the tree-like structure of universe types, collecting all attributes in Cartesian products and pruning the type sums and β-alternatives.

DEFINITION 3.15 (CLASS TYPE) The set of *class types* $\mathfrak{C}$ is defined as follows: Let $\mathcal{U}$ be the universe covering, among others, class C_n, and let $C_0, \ldots, C_{n-1}$ be the supertypes of C_n, i. e., C_i is inherited from C_{i-1}. The class type of C_n is defined as: *class type*

1. $C_i \in \mathfrak{B}, i \in \{0, \ldots, n\}$ then

$$\mathcal{C}_\alpha^0 = \Big(C_0 \times \big(C_1 \times \big(C_2 \times \cdots \times (C_n \times \alpha_\bot)_\bot\big)_\bot\big)_\bot\Big)_\bot \in \mathfrak{C}. \tag{3.56}$$

2. $\mathfrak{U}_\mathfrak{C} \supset \mathfrak{C}$, where $\mathfrak{U}_\mathfrak{C}$ is the set of universe types with $\mathcal{U}_\alpha^0 = \mathcal{C}_\alpha^0$. □

Alternatively, one could omit the lifting of the base types of the supertypes in the definition of class types. This would lead to:

$$\mathcal{C}_\alpha^0 = \Big(C_0 \times \big(C_1 \times \big(C_2 \times \cdots \times (C_n \times \alpha_\bot)\big)\big)\Big)_\bot. \tag{3.57}$$

We see our definition as the more general one, since it allows for "partial objects" potentially relevant for other object-oriented semantics for programming languages. For example Java, for which partial class objects may occur during construction. This paves the way for establishing the definedness of an object "lazy." Furthermore, since the injections and projections are only built to define attribute accessors, partial objects can be hidden on level 2, e. g., the representation of OCL formulae.

In both cases the outermost $_\bot$ reflects that class objects may also be undefined, in particular after projection from some elements in the universe or from failing type-casts. This choice has the consequence that the arguments of constructors may be undefined.

3.4.3 *Instances*

We provide for each class injections and projections. In the case of `Object` these definitions are quite easy, e. g., using the constructors Inl and Inr for type sums we can easily insert an `Object` object into the initial universe via

$$\mathrm{mk}^{(0)}_{\mathrm{Object}}\, \mathit{self} \equiv \mathrm{Inl}\, \mathit{self} \qquad \text{with type } \alpha\ \texttt{Object} \to \mathcal{U}^0_\alpha \tag{3.58}$$

and the inverse function for constructing an `Object` object out of a universe can be defined as follows:

$$\mathrm{get}^{(0)}_{\mathrm{Object}}\, \mathit{univ} \equiv \begin{cases} k & \text{if } \mathit{univ} = \mathrm{Inl}\, k \\ \varepsilon\, k.\ \mathrm{true} & \text{if } \mathit{univ} = \mathrm{Inr}\, k \end{cases} \quad \text{with type } \mathcal{U}^0_\alpha \to \alpha\ \texttt{Object}. \tag{3.59}$$

In the general case, the definitions of the injections and projections is a little bit more complex, but follows the same schema: for the injections we have to find the "right" position in the type sum and insert the given object into that position. Further, we define in a similar way projectors for all class attributes.

In a next step, we define type test functions; for universe types we need to test if an element of the universe belongs to a specific type, i. e., we need to test which corresponding extensions are defined. For `Object` we define:

$$\mathrm{isUniv}^{(0)}_{\mathrm{Object}}\, \mathit{univ} \equiv \begin{cases} \mathrm{true} & \text{if } \mathit{univ} = \mathrm{Inl}\, k \\ \mathrm{false} & \text{if } \mathit{univ} = \mathrm{Inr}\, k \end{cases} \quad \text{with type } \mathcal{U}^0_\alpha \to \mathrm{bool}. \tag{3.60}$$

For class types we define two type tests: an exact one that tests if an object is exactly of the given *dynamic type* and a more liberal one that tests if an object is of the given type or a subtype thereof. Testing the latter one, *kind* which is called *kind* in the OCL standard, is quite easy. We only have to test that the base type of the object is defined, e. g., not equal to $\bot$:

$$\mathrm{isKind}^{(0)}_{\mathrm{Object}}\, \mathit{self} \equiv \mathrm{def}\, \mathit{self} \qquad \text{with type } \alpha\ \texttt{Object} \to \mathrm{bool}. \tag{3.61}$$

An object is exactly of a specific dynamic type, if it is of the given kind and the extension is undefined, e. g.:

$$\mathrm{isType}^{(0)}_{\mathrm{Object}}\, \mathit{self} \equiv \mathrm{isKind}_{\mathrm{Object}} \wedge \neg\big((\mathrm{def} \circ \mathrm{base})\, \mathit{self}\big) \tag{3.62}$$

with type α `Object` $\to$ bool. Where base is a kind of strict operator for accessing the second element of a pair:

$$\mathrm{base}\, x \equiv \begin{cases} b & \text{if } x = {}_\llcorner(a, b)_\lrcorner, \\ \bot & \text{otherwise,} \end{cases} \qquad \text{with type } (\alpha \times \beta_\bot) \to \beta_\bot. \tag{3.63}$$

The type tests for user defined classes are defined in a similar way by testing the corresponding extensions for definedness.

Finally, we define coercions, i. e., ways to *type-cast* classes along their subtype hierarchy. Thus we define for each class a cast to its direct subtype and to its direct supertype. We need no conversion on the universe types where the subtype relations are modeled by polymorphism. Therefore we can define the type-casts as simple compositions of projections and injections, e. g., consider a direct subclass `Node` of `Object`, then we can define directly: *type-cast*

$$\mathrm{Node}^{(0)}_{[\mathrm{Object}]} \equiv \mathrm{get}_{\mathrm{Object}} \circ \mathrm{mk}_{\mathrm{Node}} \tag{3.64}$$

with type (α_1, β) Node $\to (\alpha_1, \beta_1)$ `Object`, and

$$\mathrm{Object}^{(0)}_{[\mathrm{Node}]} \equiv \mathrm{get}_{\mathrm{Node}} \circ \mathrm{mk}_{\mathrm{Object}} \tag{3.65}$$

with type (α_1, β_1) `Object` $\to (\alpha_1, \beta_1)$ Node. These type-casts are changing the *static type* of an object, while the *dynamic type* remains unchanged.

Note, for a universe construction without values, e. g., $\mathcal{U}^0_\alpha := \alpha$ `Object`, the universe type and the class type for the common supertype are the same. In that case there is a particularly strong relation between class types and universe types on the one hand and on the other there is a strong relation between the conversion functions and the injections and projections function. In more detail, see also Figure 3.4 on the following page, one can understand the projections as a cast from the universe type to the given class type and the injections as their inverse.

As reusabilty and extensibility are key concepts of object-orientation, we aim for an *open-world* within our framework. Recall that our universe construction ensures that theorems proven a given universe $\mathcal{U}$ will remain valid for extensions of $\mathcal{U}$.

Moreover, our construction allows to close a model in a fine-granular way: We can block further extensions by instantiating the α's and β's related to this class by instantiating them by the unit type. We consider this fact as a solution to the long-standing problem of extensionality for object-oriented languages, enabling to represent "open-world" and "closed-world" assumptions as polymorphism on data universes.

Our solution is more fine-grained than the concept of finalization available in several object-oriented programming languages: First, we can *finalize* a class which inhibits the inheritance from a class completely, i. e., a class cannot have any subclasses. Technically, this is done by instantiating the α of this class with unit. Second, we can *sterilize* a class, i. e., inhibit further direct subclassing but allowing "sub-subclassing." This is done by instantiating the β of the last direct subclass with unit. *finalize* *sterilize*

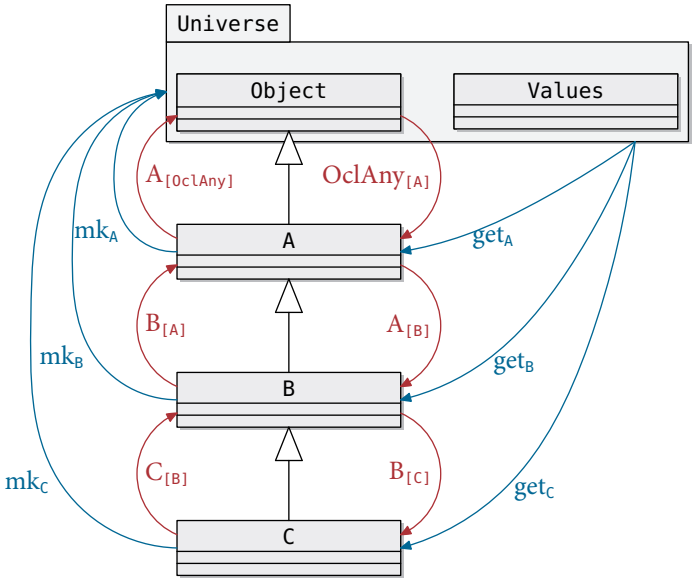


Figure 3.4: The type-casts, e. g., $B_{[C]}$ allow the conversion of a type to its direct successor or predecessor in the type hierarchy. The injections, e. g., mk_B convert a class type to the universe type and the projections, e. g., get_B, convert a universe type to a concrete class type. For a universe without values, the class type and the universe type of the top most class are identical. Here, the package `Universe` represents the universe, i. e., the top level class (`Object`) and the primitive types (`Values`).

3.4.4 *Adaption to Higher Embedding Layers*

The previous presented definitions are on the lowest layer, i. e., the introduction of new datatypes. Just as the HOL definition are adopted, over several layers, to match the object-language definitions, we have to adopt the new definitions for object-oriented datatypes.

3.5 TOWARDS A CONSTRAINED OBJECT STORE

In this section, we bring the specifications of *class invariants* and the data specifications together, i. e., we enrich the structural data model with class invariants. Often, in object-oriented modeling, invariants are a *local* property of an object, i. e., it can be locally decided if an object fulfills its invariant or not. In this setting, a class defines a *structural type*. Invariants are checked locally in a post-hoc manner, i. e., somehow it is required that an invariant holds for *all* instances of a structural type. Overall, this is similar to programming languages with runtime checking or systems based on that idea like [79] or Spec# [6, 72].

class invariant

structural type

We aim for a *deep integration of invariants*, i. e., a strong link between invariants and objects. In our understanding, there is no object (on the user level) not satisfying its invariant. We ensure this by a co-recursive scheme that we present in this section. Our co-recursive encoding scheme supports the encoding of *recursive* object structures with class invariants, introducing the concept of a *semantic type*. This is in opposite to the weak link provided in systems that do post-hoc checking of invariants.

semantic type

To illustrate why we break up with the idea of post-hoc invariant checking, consider the model of a linked list given in Figure 3.5 on the next page together with a simple invariant "positive" stating that the value of the attribute `i` must be positive and a second one "flip" stating that within a given linked list, the Boolean attribute of the nodes flips while traversing the list.

Further, consider the system state depicted in the object diagram in Figure 3.6: Is the given system state legal, i. e., do the invariants hold for object `n1`? For the first invariant, called "positive," this can be trivially decided only by interpreting the `n1` locally. But interpreting the second invariant, called "flip," only locally makes not much sense. This invariant has to be interpreted it in a global context, i. e., following the `next` links. Moreover, this also requires that `n2` is a valid instance of class `Node` and thus one has to consider all instances that are reachable from `n1` to decide if `n1` fulfills its invariant or not. Moreover, we need a construction that can also cope with cycles in the object structure, as illustrated in the two states in Figure 3.7 and Figure 3.8. Here, only the state in Figure 3.8 is consistent. The situation can become much more complex, involving inheritance and more complicated object structures. Such scenarios are

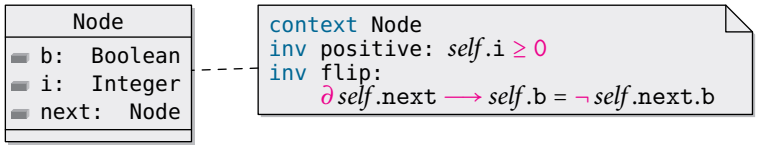


Figure 3.5: In this diagram we model a simple linked list: every instance of class `node` can have a successor (`node`) and has two attributes storing an Integer and a Boolean value. Using an invariant, we require that the Integer attribute is positive and that the value of the Boolean attribute is different from the Boolean value of the next node, i. e., the Boolean attributes should flip its value as we traverse an instance of the linked list.

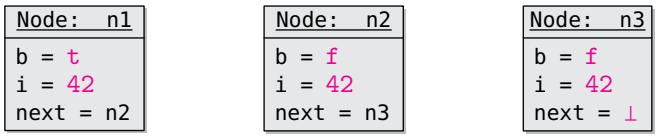


Figure 3.6: In this object diagram we illustrate an instance of our linked list structure. Consider the first node `n1` and decide if the invariant `flip` for class `Node` holds or not.

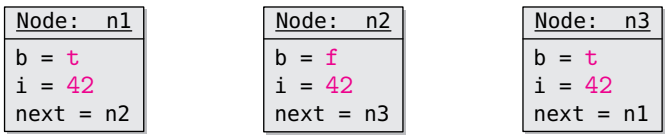


Figure 3.7: In this object diagram we illustrate a second instance of our linked list structure. Consider the first node `n1` and decide if the invariant `flip` for class `Node` holds or not.

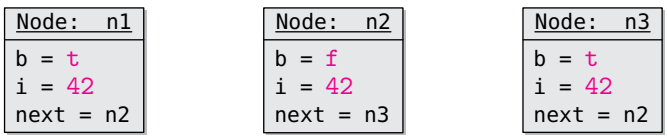


Figure 3.8: In this object diagram we illustrate a third instance of our linked list structure. Consider the first node `n1` and decide if the invariant `flip` for class `Node` holds or not.

our main motivation for a co-recursive construction of type and kind sets that are aware of class invariants. Moreover, if done conservatively, this construction will rule out invariants that are dangerous in the sense that they could introduce logical inconsistencies, which we will discuss this in more detail in Section 3.5.5. In Section 3.5.5 we will compare the approach of using structural types with post-hoc invariant checking, which we will introduce in Section 3.5.1, with semantic invariants as introduced in Section 3.5.2.

3.5.1 *Structural Types with Post-hoc Invariant Checks*

In this section, we will give a brief overview of the idea of post-hoc invariant checking, as it is for example used in the KeY [1] tool or USE [100]. These tools assume structural type and kind sets, e. g., Java classes. In this scenario there an instance of a class can fulfill an invariant or not, i. e., only a subset of all possible instances fulfill the class invariant. In the remainder of this section, we will show how structural type and kind sets can be defined within our framework and how formulae can be constructed that allow for testing if an instance fulfills the class invariant or not.

DEFINING STRUCTURAL TYPE AND KIND SETS. Based on the type and kind tests introduced in Section 3.4.3, we can easily define *structural* type and kind sets for each class. Our constructs on level 1, i. e., $\mathrm{isType}^{(1)}_{_}$ and $\mathrm{isKind}^{(1)}_{_}$ provide a test for structural types and kinds, e. g., for the class `Node` we define:

$$\mathrm{TypeSet}_{\mathrm{Node}} \equiv \{obj :: (\alpha_C, \beta_C)\,\texttt{Node} \mid \mathrm{isType}^{(1)}_{\mathrm{Node}}\, obj\} \tag{3.66}$$

and

$$\mathrm{KindSet}_{\mathrm{Node}} \equiv \{obj :: (\alpha_C, \beta_C)\,\texttt{Node} \mid \mathrm{isKind}^{(1)}_{\mathrm{Node}}\, obj\}. \tag{3.67}$$

These possibly infinite sets describe all structural objects of a specific type or kind. In particular, the elements of these sets are not required to fulfill the class invariants for the given type or kind.

POST-HOC INVARIANT CHECKING. Based on the structural projectors and injectors (i. e., the first-level definitions), we can directly give a semantics for formulae of our constraint language. For example, the invariant `positive` of class `Node` can be directly described as:

$$\mathrm{invPositive}_{\mathrm{Node}}\, self \equiv self.\mathrm{i}^{(1)} \geq \texttt{0} \tag{3.68}$$

where $self.\mathrm{i}^{(1)}$ is the access of attribute i on level 1.

Nevertheless, this is only a formula defined over our object store, in particular, it is not an invariant. Informally, the meaning of being an

invariant is often described as follows; an invariant of a class must evaluate to true for all instances (objects) of that class at any time. Operational semantic definitions as used in evaluation environments, e. g., [34] or validation environments, e. g., [100], require implicitly (on the meta-level) that all instances are satisfying their invariants.

Consider again our linked-list example, shown in Figure 3.5 on page 72. For environments based on a post-hoc invariant checking, the instances `n1` of Figure 3.7 satisfies its invariants and thus is valid. Nevertheless, if due to a meta-level requirement that all instances of a concrete system state have to fulfill their invariant, the state would be rejected because the instance `n3` does not fulfill its *local* invariant, i. e., `n3.b = n3.next.b` contradicts the requirements of the invariant `flip`. However, in this concrete example, the the consistency check will not terminate for most implementations of evaluation environments. This is caused by the fact that these environments are usually not designed with recursive data structures in mind.

In a proof environment, postulating that all instances fulfill their invariants can easily falsify the assumption of proofs (which makes them trivial) or even introduce deep logical inconsistencies. We will discuss this problem in more detail in Section 3.5.5.

In the next section, we present a co-recursive construction that provides a semantic definition of types and kind sets which can guarantee, by construction, that only objects fulfilling their invariants are of a given type.

3.5.2 *Defining Semantic Type and Kind Sets*

In this section, we present a co-recursive encoding that ensures, by construction, that each instance of a class fulfills the class invariants. We extend the type discipline to be aware of invariants, i. e., being of a specific type also ensures that the invariants of this type are fulfilled. In particular, we introduce type sets (which can be seen as a kind of first level invariant) which later on define a second level construct, providing a representation that looks familiar to people involved in object-oriented modeling.

A CO-RECURSIVE TYPE AND KIND SET CONSTRUCTION. In a setting with subtyping, we need two characteristic type sets, a sloppy one, the *characteristic kind set*, and a loose one, the *characteristic type set*. We define these sets co-recursively. As basis for our co-recursive construction, we built for each invariant a HOL representation, i. e., in each formula where we replace recursively the logical connectives of our object-language with their HOL counterpart by requiring the validness of the sub-formula. This is done using the *logical judgment* $\tau \vDash P$ which means that the object-logical formula P *is valid* (i. e., evaluates to `t`) in context

τ. As we want to use these invariants for a co-recursive construction we parametrize them over the current state τ, the object *self* and the type set C we are constructing.

Recall our previous example (see Figure 3.5), where the class `Node` describes a potentially infinite recursive object structure. The invariant of class `Node` constrains the attribute `i` to values greater or equal than 0. For this constraint, we generate

$$\text{hol_inv_positive}\ \tau\ C\ \mathit{self} \equiv \tau \vDash \mathit{self}.\,\mathrm{i}^{(1)} \geq 0\,. \tag{3.69}$$

Further, we generate an invariant expressing the fact that `next` be either undefined or of type `Node`:

$$\begin{aligned}\text{hol_inv_type_next}\ \tau\ C\ \mathit{self} \equiv\ & \tau \vDash \not\partial\,\mathit{self}.\,\mathrm{next}^{(1)} \\ & \vee\ \tau \vDash \mathit{self}.\,\mathrm{next}^{(1)} \in (\lambda\, f.\ f\,\tau)\ \text{`}\ C\end{aligned} \tag{3.70}$$

where _ ` _ denotes the point-wise application.

Now we define a function for construction the kind set for `Node` which approximates the set of possible instances of the class `Node` and its sub-classes:

$$\begin{aligned}&\text{NodeKindF} :: \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node set} \\ &\qquad\qquad \Rightarrow \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node set} \\ &\text{NodeKindF} \equiv \lambda\,\tau.\ \lambda\, X.\ \{\mathit{self} \mid \text{hol_inv_type_next}\ \tau X \mathit{self} \\ &\qquad\qquad\qquad\qquad \wedge\ \text{hol_inv_type_next}\ \tau\ X\ \mathit{self}\,.\end{aligned} \tag{3.71}$$

By adding the conjunct $\tau \vDash \mathrm{isType}^{(1)}_{\mathrm{Node}}\ \mathit{self}$, we can construct another approximation function (which has obviously the same type as NodeKindF):

$$\begin{aligned}&\text{NodeTypeF} :: \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node set} \\ &\qquad\qquad \Rightarrow \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node set} \\ &\text{NodeTypeF} \equiv \lambda\,\tau.\ \lambda\, X.\ \{\mathit{self} \mid (\mathit{self} \in (\text{NodeKindF}\ \tau\ X)) \\ &\qquad\qquad\qquad\qquad \wedge\ \tau\,\mathrm{isType}^{(1)}_{\mathrm{Node}}\ \mathit{self}\}\,.\end{aligned} \tag{3.72}$$

Thus, the characteristic kind set for the class `Node` can be defined as the greatest fixed-point over the function NodeKindF:

$$\begin{aligned}&\text{NodeKindSet} :: \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow \mathcal{U}^1_{(\alpha_C,\beta_C,\beta_N)}\ \mathrm{St} \Rightarrow (\alpha_C,\beta_C)\ \texttt{Node set} \\ &\text{NodeKindSet} \equiv \lambda\,\tau.\ (\mathrm{gfp}(\text{NodeKindF}\ \tau))\,.\end{aligned} \tag{3.73}$$

For the characteristic type set we proceed analogously. Further, we prove automatically, using the monotonicity of the approximation functions, the point-wise inclusion of the kind and type sets:

$$\text{NodeTypeSet} \subset \text{NodeKindSet}\,. \tag{3.74}$$

This property represents semantically the subtype relation. This kind of theorem remains valid if we add further classes in a class system.

Now we relate class invariants of subtypes to class invariants of supertypes. The core of the construction for characteristic sets taking the class invariants into account is a greatest fixed-point construction (reflecting their co-algebraic properties). We proceed by defining a new approximation for a subclass Cnode on the basis of the approximation function of the superclass:

$$\begin{aligned}\text{CnodeF} \equiv \lambda\,\tau.\ \lambda\,X.\\ \big\{\mathit{self} \;\big|\; \mathit{self}^{(1)}_{[\text{Node}]} \in (\text{NodeKindF}\ \tau\ (\lambda\,o.\ o^{(1)}_{[\text{Node}]}) \setminus X)\\ \wedge\ (\varphi\ \tau\ X\ \mathit{self})\big\}\end{aligned} \tag{3.75}$$

where φ stand for the constraints specific to the subclass. Note φ must appropriately include $\tau \vDash \mathit{self}.\,\text{next}^{(1)} \in (\lambda\,f.\ f\,\tau) \setminus X$ to make the implicit recursion in the `Cnode` invariant explicit.

Similar to [94] we can support mutual-recursive datatype definitions by encoding them into a type sum. However, we already have a suitable type sum together with the needed injections and projections, namely our universe type with the make and get methods for each class. The only requirement is that a set of mutual recursive classes must be introduced "in parallel," i. e., as *one* extension of an existing universe.

Now we can easily introduce type and kind tests that are aware of the invariants, i. e., which test if a given object is structural from the requested type (or kind) *and* fulfills its invariant can be reduced to membership test for the type or kind sets. Moreover, our construction for type sets and kind sets provides for an object a tight connection between "being of a type" and "fulfilling its invariant," i. e., the invariant for `Node` can be defined semantically as follows:

$$\text{Node_sem_inv}\ \mathit{self} \equiv \mathit{self} \in \text{NodeKindSet} \tag{3.76}$$

with type $(\alpha_C, \beta_C)\,\texttt{Node} \to \texttt{Boolean}_\tau$.

AN OBJECT-ORIENTED INVARIANT REPRESENTATION. The semantic invariant definition introduced in the previous section is not a representation of invariants somebody used to object-oriented modeling

would expect. Therefore we define for each class a "user representation." This representation is based on the accessor introduced in the last section, e. g., for `Node` we define:

$$\text{Node_defined}\, \mathit{self} \equiv \partial\, \mathit{self}\ , \tag{3.77}$$

$$\text{Node_inv_positive}\, \mathit{self} \equiv \mathit{self}\,.\,\text{i} \geq 0\ , \text{and} \tag{3.78}$$

$$\begin{aligned} \text{Node_inv}\, \mathit{self} \equiv\ & \text{Node_defined}\, \mathit{self} \\ & \wedge\, \text{Node_inv_positive}\, \mathit{self}\ . \end{aligned} \tag{3.79}$$

The constraint Node_defined ensures the definedness of *self*, which allows us to prove the equivalence of both invariant representations, e. g., for `Node` we prove:

$$\text{Node_inv}\, \mathit{self} = \text{Node_sem_inv}\, \mathit{self}\ . \tag{3.80}$$

Thus, we can provide the users of our framework with an invariant representation that looks familiar to the user specification, e. g., expressed using OCL:

```
context Node
  inv positive: self.i >= 0
  inv flip: not self.next.isOclUndefined()
            implies self.b = not next.b
```

Moreover, this representations allows proving of many system properties without the need of using co-recursive induction schemes.

Using co-induction for defining recursive datatypes allows for specifications of infinite structures. Alternatively, the introduction of a measure function allows for restricting data-recursion to finite data-structures. On finite data-structures, the usual inductive proof techniques can be applied.

It would be desirable to provide support for the concept of co-recursion, e. g., by generating specialized (co)recursion theorems for user defined datatypes (object-structures). This would pave the way for powerful recursion concepts in the constraint language that are not limited to well-foundedness. Operationally, such recursion schemes would correspond to lazy evaluations over cyclic data-structures known in functional programming languages such as Haskell.

In particular, co-recursive operators over the user-defined types can be defined such as the deep value equality under which to objects are equal if they are describe (recursively) the isomorphic object structure and the corresponding attribute with primitive types are equal, i. e., the object structures are bi-similar. The issue is discussed in more detail in Section 3.6.

3.5.3 *Combining Embeddings*

Combining a constraint language with our store raises another question that is closely related to the chosen type of invariant encoding. Namely, what should be the formal semantics of operations returning instances (e. g., accessors, constructors); in more detail: should the returned instance fulfill the invariant of its type (class). In systems using post-hoc invariant checking, the underlying semantic construction normally does not guarantee that a returned instance fulfills its invariant. This property has either to be tested post-hoc or is required on the meta-level by an assumption "all objects of the current state fulfill their invariants."

Using our semantic invariant definition we can go one step further in combining our constraint language embedding with our embedding of the object-oriented store: we test if a structural object (on the first level) fulfills its invariant and if not, we just return undefined. This is easily achieved by defining the second level constructs using a wrapper testing if a first level constructs fulfills its invariant. For example, we define the accessor for attribute `i` of class `Node` as follows:

$$\mathit{self}.\mathrm{i}^{(2)} \equiv \texttt{if}\ \mathrm{Node_inv}\,\mathit{self}\ \texttt{then}\ \mathit{self}.\mathrm{i}^{(1)}\ \texttt{else}\ \bot\ \texttt{endif}\ . \tag{3.81}$$

As a consequence, we can easily derive the rule

$$\frac{\tau \vDash \partial(\mathit{self}.\mathrm{i}^{(2)})}{\tau \vDash \mathrm{Node_inv}\,\mathit{self}}, \tag{3.82}$$

which does not hold in a setting with structural types and post-hoc invariant checking.

All other, previously introduced first level constructs, are also adopted by wrapping them in an `if`-statement.

Note, for finite system state and a closed-word scenario such a second level construction is also possible for a system with post-hoc invariant checking. Relaxing one of those constraint, i. e., allowing the system to be extensible, or infinite system states, this construction can lead, if done naïvely, to logical inconsistencies. In that case, the monotonicity proofs needed for our semantic type sets will fail and reject the model, and thus protecting the user of our system against this situation. We will discuss this problem in more detail in Section 3.5.5.

3.5.4 *Varying Invariants*

For program verification projects the user defined invariants are often too strong, at least if the invariants are not syntactically restricted to formulae only constraining visible, i. e., being public, part of the class attributes. In settings allowing arbitrary invariants, often the need for temporary

disobeying an invariant, i. e., during the construction of object structures or inner calls, occurs. This observation by us and also by others [72] motivates our decision for providing means for varying invariants. In the remainder of this section, we will describe two mechanisms for providing such relaxed invariants within a proof environment.

PROVIDING DIFFERENT TYPE SETS AND KIND SETS. The discussed type sets and kind sets are of major importance when resolving overriding and late-binding: If we can infer from a class invariant that some object must be of a particular *type*, then late-binding method invocation can be reduced to a straight-forward procedure call with simplified semantics.

As a default we generate for each class three different type sets and kind sets:

1. a set based on the user-defined invariant,
2. a set allowing undefined references, i. e., all accessors to attributes of type oid are or-ed with a corresponding $\not\phi$-statement, and
3. a set allowing undefined references *and* undefined primitive types, i. e., all accessor to attributes are or-ed with a corresponding $\not\phi$-statement.

For example, recall our model of a linked list (see Figure 3.5 on page 72). For the last variant described above, i. e., item 3, the HOL representation of the invariant constraining the attribute `i` to positive values we change Equation 3.69 on page 75 to

$$\begin{aligned}\text{hol_inv_positive_relaxed}\ \tau\ C\ \mathit{self} \equiv \tau \vDash \mathit{self}.\,\mathrm{i}^{(1)} \geq 0 \\ \vee\ \tau \vDash \not\phi\ \mathit{self}.\,\mathrm{i}^{(1)}\end{aligned} \tag{3.83}$$

and repeat the construction described in Section 3.5.2.

The above enumeration is listed in ascending order, i. e., every object that is in the first set, is also included in the other two. Such an hierarchy of invariants allows for formally specifying relaxed variants of class invariants necessary during program verification. In a program state, for example, it is not possible to create an object graph at once where all references are defined. Rather, a program proceeds by steps with undefined references (assuming a relaxed invariant/characteristic set following item 3) ending up by establishing a stronger invariant following item 1). Thus, the support of different invariant versions is a corner stone of successful verification of object-oriented systems, see also [72] where the authors argue for weakened invariants during inner calls for systems specified using JML.

Moreover, there will be the need for even more fine-grained invariants in verification practice. Thus, in our proof environment we support two technical means for defining relaxed invariants:

1. define a function that converts structurally an invariant into a relaxed one. This technique is internally used for construction the default type and kind sets.

2. for specification languages allowing to specify conjuncts of the invariant separately, as it is the case for OCL, we allow to switch conjuncts individually on or off.

Overall, we see such an approach as a key feature for successful program verification, e. g., using a shallow embedding of a small object-oriented language like IMP++ [23].

3.5.5 *Advantages of Semantic Invariants*

Overall, the concept of semantic type and kind sets, as introduced in Section 3.5.2, allows for proving many side-conditions once and for all during the construction of these sets which otherwise lead to assumptions in nearly every proof over a given class. In particular for recursive data structures the advantage of semantic invariants will become clear. Here, the identification between "type" and the "set of objects satisfying a class invariant" makes invariants to *recursive* predicates whenever the object structure is recursive. As a consequence, the construction of a conservative model for a system of class invariants is far from trivial.

One might ask what benefit an end-user will get from conservativity after all. Its need becomes apparent when stating class invariants, thus stating recursive predicates, as *axioms*: this results in *logical inconsistency*. Consider the following constraint as an invariant of class `A`:

$$\neg\, \mathrm{isKind}_{\mathtt{A}}\ \mathit{self} \tag{3.84}$$

which requires for all instances of type `A` not to be of kind `A`, i. e., neither of type`A` or a subtype of `A`. Thus, it is in fact possible to state a variant of Russell's paradox which is known to introduce logical inconsistency in naive set theory. Inconsistency means that the logic of the constraint logic can derive any fact; this might be exploited by an automated tactic accidentally. Logical inconsistency is different from an *unsatisfiable* class invariant meaning "there is no instance." In particular, in an inconsistent system, each class invariant can be proven both satisfiable and unsatisfiable.

Moreover, similar problematic situations can occur, for recursive invariants. Therefore, systems using an axiomatic post-hoc invariant checking

approach, like Spec# [6, 72], restrict syntactically the allowed invariants in a very rigid way.

Instead of restricting the set of allowed invariants syntactically, Our conservative construction requires proofs of side-conditions which will fail in the above described situation. Technically, it provides a model for the mutual recursive predicates representing class invariants. This model, i. e., the set of legal states satisfying the invariant, is a result of a greatest fixed-point computation. The existence this greatest fixed-point is reduced to a monotonicity argument. If the latter cannot be produced automatically (or by user-interaction), a given class invariant will be rejected. Thus, paradoxical situations like the one above are ruled out while admitting the "useful" forms of recursion in class invariants.

Moreover, using semantic type sets allows for defining second level operations, i. e., from the definedness of result of a operation, we can directly conclude that it also fulfills its invariant. For example consider a class `A` with an attribute `b::B`. From the definedness of the result of the access to attribute `b` we can conclude already:

$$\frac{\tau \vDash \partial\, \mathit{self}\,.\,\mathrm{b}}{\tau \vDash \partial\, \mathit{self}}, \tag{3.85a}$$

$$\frac{\tau \vDash \partial\, \mathit{self}\,.\,\mathrm{b}}{\tau \vDash \mathrm{A_inv}\, \mathit{self}}, \text{ and} \tag{3.85b}$$

$$\frac{\tau \vDash \partial\, \mathit{self}\,.\,\mathrm{b}}{\tau \vDash \mathrm{B_inv}\, \mathit{self}\,.\,\mathrm{b}}. \tag{3.85c}$$

We are aware that this identification between the notion of a type and its semantics is theoretically involved and not widely used in proof environments for object-oriented languages; for example, systems like Boogie or Key do not have this interpretation of type. In these systems the type of a class is defined by its structure, i. e., its attributes. Thus, there can be both instances of a given type that fulfill the invariant of this type and those who not fulfill the invariant of this type. We prefer the concept of types that ensure the invariant as this eliminates the need for proving side-conditions like "in a given system state, the return value of an attribute accessor will return an object that fulfills its invariant." Our constructions rules out such situations and guarantees that in any visible state, all objects fulfill their invariant. Overall, this should lead to a more natural reasoning over object structures.

3.6 EQUALITIES AND OBJECT-ORIENTATION

Equality, in its broadest sense, is an important property for both programming and formal reasoning. Historically, object-oriented systems

are equipped with a variety of different "equalities" [65]. Answering the question if two objects are equal is not so obvious: e. g., are two objects equal only if their object identifier is equal (are they the same object?) or are two objects equal if their values are equal, or are they equal if they are observably equivalent with respect to the accessor functions? In this section, we assume that every object has a unique identifier, called object identifier or reference of an object.

Whereas in traditional specification formalisms the equality is defined over values, the most basic equality over objects is the reference quality or identity equality which is also the kind of equality that is usually provided as a default, i. e., "built-in," equality in object-oriented languages. Thus, there is usually a fundamental difference between values and objects.

primitive type

DEFINITION 3.16 (PRIMITIVE TYPES) The set of *primitive types* $\mathfrak{P}$ is defined inductively as follows:

1. $\{\texttt{Boolean}, \texttt{Integer}, \texttt{Real}, \texttt{String}, \} \subset \mathfrak{P}$, and
2. $\{v\ \texttt{Set}, v\ \texttt{Sequence}, v\ \texttt{Bag}, v\ \texttt{OrderedSet}\} \subset \mathfrak{P}$ for all $v \in \mathfrak{P}$. □

value

DEFINITION 3.17 (VALUES) An instance of a primitive type, e. g., $x :: v$ with $v \in \mathfrak{P}$ is called *value*. □

Normally one expects that an equality is an equivalence relation.

equivalence relation

DEFINITION 3.18 (EQUIVALENCE RELATION) An *equivalence relation* is a binary relation $_ \sim _$ over a set S for which the following properties hold:

- Reflexivity: $a \sim a$, for all $a \in S$.
- Symmetry: $a \sim b$ if and only if $b \sim a$, for all $a, b \in S$.
- Transitivity: if $a \sim b$ and $b \sim c$ then $a \sim c$, for all $a, b \in S$. □

Now we introduce in an abstract way the basic qualities of object-oriented systems, we ignore undefinedness in these definitions. In a second step, we will show that the treatment of undefinedness is orthogonal and can be combined with any of the following equalities.

Most object-oriented languages have the concepts of references or object identifiers where a reference uniquely identifies an object. Thus it seems a natural choice to use these references for defining an equality, namely the reference quality.

reference equality

DEFINITION 3.19 (REFERENCE EQUALITY) The referential equality or *reference equality* is defined as follows:

1. Two values are reference equal, if they are of the same type and represent the same value.

2. Two objects are reference equal, if their object identifiers (references) are equal. □

Thus, the reference equality tests if two objects represent in fact the same object in a store. Often, the reference equality is also called *identity equality*; informally, it identifies an object uniquely, i. e., even objects representing the same value can be distinguished with this type of equality.

If we want to test, if two objects represent the same value we have two options; a shallow and a deep one:

DEFINITION 3.20 (VALUE EQUALITY) The *shallow value equality* or just *value equality* is defined as follows: *shallow value equality*

1. Two values are shallow value equal, if they are of the same type and represent the same value.

2. Two objects are shallow value equal, if they are of the same type and all attributes with primitive types are pairwise shallow value equal. □

This definition is not recursive, hence the name shallow equality. The main idea behind the shallow equality is to compare two singular objects as values. In contrast to this, we can define the deep value equality for comparing the values of two object structures.

DEFINITION 3.21 (DEEP VALUE EQUALITY) The *deep value equality* is defined as follows: *deep value equality*

1. Two values are deep value equal, if they are of the same type and represent the same value.

2. Two objects are deep value equal, if they are of the same type and
   a) all attributes with primitive types are pairwise deep value equal.
   b) all attributes with type oid (object type) are pairwise deep value equal if the objects they refer to are deep value equal. □

Summarizing, we have already three different equalities:

1. the reference equality which checks if two objects are in fact the same object,

2. the shallow value equality which compares the values of the attributes on the first level, and

3. the deep value equality which compares recursively the object structure comparing the equality of the corresponding parts.

Assuming a setting, where all values and oid are defined, i. e., the classical two-valued view, each of them is an equivalence relation on objects. It seems to be obvious that in a universe without undefinedness Definition 3.19 refines Definition 3.21 and Definition 3.21 refines Definition 3.20. Thus, two objects that are reference equal are also shallow equal and deep equal. But if we have undefined values and object it is not clear how these equalities relate to each other. First, in a world with undefinedness we can apply the concept of strictness to equivalence relations:

Taking undefinedness into account, e. g., values and references can be undefined, the setting gets more complicated. First we generalize the concept of equality relations by introducing equivalence operators.

equality operator

DEFINITION 3.22 (EQUALITY OPERATOR) An *equality operator* $_ \backsim _$ is a binary operator that satisfies the following properties for a state τ and a context-passing P:

- Quasi-reflexivity:

$$\frac{\tau \vDash \partial\, x}{\tau \vDash x \backsim x} \tag{3.86}$$

- Quasi-symmetry:

$$\frac{\tau \vDash \partial\, x \quad \tau \vDash \partial\, y \quad \tau \vDash x \backsim y}{\tau \vDash y \backsim x} \tag{3.87}$$

- Quasi-transitivity:

$$\frac{\tau \vDash \partial\, x \quad \tau \vDash \partial\, y \quad \tau \vDash \partial\, z \quad \tau \vDash x \backsim y \quad \tau \vDash y \backsim z}{\tau \vDash x \backsim z} \tag{3.88}$$

- Quasi-substitutivity or quasi-congruence:

$$\frac{\tau \vDash \partial\, x \quad \tau \vDash \partial\, y \quad \tau \vDash x \backsim y \quad \tau \vDash P(x)}{\tau \vDash P(y)} \tag{3.89}$$

□

This definition uses the *logical judgment* $\tau \vDash P$, which means that the object-logical formula P *is valid* (i. e., evaluates to $\mathtt{t}$) in context τ; this judgment is defined and discussed in Chapter 5.

In the following, we characterize certain classes of three-valued *equality operators*.

strong equality

DEFINITION 3.23 (STRONG EQUALITY) An equality operator $_ \mathrel{\hat{=}} _$ is a *strong equality* if it satisfies the property: $(\bot \mathrel{\hat{=}} \bot) = \mathtt{t}$. □

	strict	strong (non-strict)
referential equality	$o_1 \doteq o_2$	$o_1 \mathrel{\hat{=}} o_2$
shallow value equality	$o_1 \mathrel{\dot{\simeq}} o_2$	$o_1 \mathrel{\hat{\simeq}} o_2$
deep value equality	$o_1 \mathrel{\dot{\cong}} o_2$	$o_1 \mathrel{\hat{\cong}} o_2$

Table 3.8: In an object-oriented setting one has to deal with several different equalities and all of them can be strict or non-strict. We mark the strict variants with a dot and the non-strict one with a triangle.

This strong equality is reflexive, symmetric, transitive and substitutive (even for undefined values) which explains its importance in deduction.

Applying the concept of strictness to an equality operator results in the following definition:

DEFINITION 3.24 (STRICT EQUALITY) An equality operator $_ \doteq _$ is a *strict equality* if it evaluates to undefined whenever one of its arguments is undefined, i. e., if the following properties hold: *strict equality*

$$(o \doteq \bot) = \bot\,, \qquad (\bot \doteq o) = \bot\,, \text{ and} \qquad (\bot \doteq \bot) = \bot\,. \tag{3.90}$$

□

Strictly speaking, these last definitions are merely algebraic *characterizations* and not definitions. These operation symbols were characterized by some properties, but they are obviously not *defined* up to isomorphism. In our context, two interpretations of the equalities into the semantic domain of universes are of particular importance: when comparing objects, we can define the equality operation via HOL-equality in the object representation in the referential or the non-referential universe (when comparing values, we compare them via HOL-equality anyway).

Thus, the concept of strictness is orthogonal to the semantics of equality if the arguments are defined. Thus we can combine this with all of our previous equality variants. In principle this results in six different equalities for the object-logic (see also Table 3.8). Albeit, with respect to the interpretation of the equality operators (assuming only objects in the range of the state whose reference field just contains the reference to the object in the store), strong and strict equality operators both coincide with referential equality since we have a bijective mapping between the values of an object and the object identifier. This is only true when comparing objects within one state and not in constructs such as: `x.a@pre = x.a`.

In case of a referential universe construction, our framework allows for the definition of the strict and strong referential equality directly, e. g., the strong equality

$$_ \mathrel{\hat{=}} _ \equiv \mathrm{lift}_2(\lambda\, x\; y.\; \llcorner x = y \lrcorner) \tag{3.91}$$

and the strict equality

$$_ \doteq _ \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ \llcorner x = y \lrcorner))) . \tag{3.92}$$

Both equalities have the type $[(\tau, \alpha :: \mathrm{bot}), (\tau, \alpha :: \mathrm{bot})] \Rightarrow \mathtt{Boolean}_\tau$. In the non-referential setting, these definitions result in the strict and strong value equality, here the distinction between the shallow and deep equality does not make sense. In the referential setting. both the shallow value and the value equality have to be defined for each class separately. While this can be done by a constant definition for the shallow value equality, the deep value equality requires a recursive definition.

3.7 OPERATIONS FOR ACCESSING THE SYSTEM STATE

In this section, we define several operations that are parametrized over the current system state, i. e., they provide a limited form of reflection. These properties allow for restricting a specific system state and also allows for describing the behavior of constructors within our object-oriented constraint language. Namely we define an operation for obtaining all object instances of a type in a given state and an operation for testing if an instance is new.

3.7.1 *Accessing all Instances of a State*

We define an operation allInstances _ that returns all instances (objects) of a class in a specific type. In principle, for a given type (represented by its kind-set), the operation allInstances _ just returns the intersection of this set with the range in the state $\sigma = (\tau, \tau')$. We define it using an overloaded constant definition for each type, for example the definitions for $\mathtt{Integer}_\tau$ (a primitive type) and $\alpha\ \mathtt{Object}_\tau$ looks as follows:

$$\mathrm{allInstances}\ (\mathit{self} :: \mathtt{Integer}_\tau) \equiv \mathit{self} \tag{3.93}$$

$$\mathrm{allInstances}\ (\mathit{self} :: \alpha\ \mathtt{Object}_\tau) \equiv \lambda(\tau, \tau').\ \mathrm{Abs}_{\mathrm{Set}}(\llcorner \mathrm{get}_{\mathrm{Object}}\ \backprime\ \tau \lrcorner) \tag{3.94}$$

of type $V_\tau(\alpha) \to \alpha\ \mathtt{Set}_\tau$. Similarly, we define an operation for accessing all instances of the previous state, e. g.:

$$\mathrm{allInstances@pre}\ (\mathit{self} :: \mathtt{Integer}_\tau) \equiv \mathit{self} \text{ and} \tag{3.95}$$

$$\mathrm{allInstances@pre}\ (\mathit{self} :: \alpha\ \mathtt{Object}_\tau) \equiv \lambda(\tau, \tau').\ \mathrm{Abs}_{\mathrm{Set}}(\llcorner \mathrm{get}_{\mathrm{Object}}\ \backprime\ \tau' \lrcorner), \tag{3.96}$$

also of type $V_\tau(\alpha) \to \alpha\ \mathtt{Set}_\tau$. For user-defined classes the returned set is finite, i. e., in every system state there are only finitely many instances of a class. For for primitive types, the returned set is infinite; for example,

for $\texttt{Integer}_\tau$ the result represents the set of all integers. One can avoid infinite sets by defining

$$\begin{aligned}&\text{allInstancesFin}\ T \equiv \\ &\quad \lambda(\tau, \tau').\ \text{if}(T = \texttt{Integer}) \vee r(T = \texttt{Real}) \vee (T = \texttt{String}) \\ &\qquad\qquad \text{then}\ \bot\ \text{else}\ \text{allInstances}\ T\ (\tau, \tau')\,.\end{aligned} \tag{3.97}$$

This definition avoids infinite results by explicitly returning undefined for the primitive datatypes. Additionally, we can directly define variants of these operations for accessing all instances of the previous state. Nevertheless, we prefer the first definitions as it allows for specifying algebraic laws like

$$\forall\, x, y \in \texttt{Integer}.\ x + y \triangleq y + x \tag{3.98}$$

within the object-oriented constraint language.

3.7.2 *Testing for New Instances*

Our constraint language does not provide the concept of constructors in the sense of an object-oriented programming language. Nevertheless, we define a test which can be used in postconditions to test if a concrete instance is new:

$$\begin{aligned}\mathit{isNew}\ \mathit{self} \equiv \lambda(\tau, \tau').\ \llcorner &\text{get}^{(0)}_{\text{Object}}(\mathit{self}_{[\text{Object}]}(\tau, \tau')) \notin \text{ran}\ \tau \\ &\wedge\ \text{mk}^{(0)}_{\text{Object}}(\mathit{self}_{[\text{Object}]}\ (\tau, \tau')) \in \text{ran}\ \tau' \lrcorner\end{aligned} \tag{3.99}$$

of type $V_\tau(\alpha) \to \texttt{Boolean}_\tau$. In this context "new" means that the instances does not exist in the previous (direct predecessor) state.

3.8 ON OPERATION SPECIFICATIONS

Using preconditions and postconditions for specifying operations, especially in a three-valued constraint language, raises the questions what the precise meaning of this "contract" should be. In particular what should happen if a precondition of an operation is not fulfilled, e. g., it is invalid or undefined? In a programming language one could think of several different behaviors: The implementation of the operation might

- raise an exception,
- diverge,
- terminate without changing the system state, or

- terminate leaving the system in an arbitrary state, even one fulfilling the postcondition.

For specification languages that are based on preconditions and postconditions for defining an operation specification, two different interpretations are found in the literature: either we require that the precondition implies the postcondition, or stronger, that both must hold. Formally, we define:

DEFINITION 3.25 (OPERATION SPECIFICATION) Let $\text{pre}_{op}\ \mathit{self}\ a_0 \cdots a_n$ be the precondition and $\text{post}_{op}\ \mathit{self}\ a_0 \cdots a_n\ \mathit{result}$ the postcondition of the operation *op*. The *operation specification* with implication semantics is defined as

operation specification

$$\begin{aligned} &S_{op}^{\text{imp}}\ \mathit{self}\ a_0 \cdots a_n\ \mathit{result} \\ &\quad \equiv \tau \vDash (\text{pre}_{op}\ \mathit{self}\ a_0 \cdots a_n) \longrightarrow (\text{post}_{op}\ \mathit{self}\ a_0 \cdots a_n\ \mathit{result}) \end{aligned} \tag{3.100}$$

and the *operation specification* with conjunct semantics is defined as

$$\begin{aligned} &S_{op}^{\text{conj}}\ \mathit{self}\ a_0 \cdots a_n\ \mathit{result} \\ &\quad \equiv \tau \vDash (\text{pre}_{op}\ \mathit{self}\ a_0 \cdots a_n) \wedge (\text{post}_{op}\ \mathit{self}\ a_0 \cdots a_n\ \mathit{result}) \end{aligned} \tag{3.101}$$

where in both definitions we replace all accessor occurring in the precondition pre_{op} by their `@pre`-variant. □

The precondition pre_{op} is a predicate function depending on the input parameters including the implicit input parameter *self*. The postcondition post_{op} is a predicate function depending on the input parameter and the implicit *result* parameter. The implicit parameter *self* represents the object (class instance) for which the operation is called. Therefore, *self* is also called *context object*. The return value of the operation is described by the parameter *result*.

self

context object

result

In case of S_{op}^{conj} the preconditions and postconditions are conjoint. Thus, a false precondition simply says that there is "no transition" from a state to its successor; this corresponds to the operational behavior of an exception, a divergence or a deadlock. Moreover, an undefined precondition may either result in an undefined or false operation specification; in the former case, no statement on the implementation is made, i. e., it may behave arbitrarily. Thus, when writing a specification, there is the possibility to explicitly distinguish these possibilities: a precondition $a > 5$ makes no statement for the case that a is an undefined object in a particular state (provided the postcondition is valid), in $\partial\, a \wedge a > 5$; however, it is explicitly specified that there is no successor state, if a is not defined in the previous state.

In case of S_{op}^{imp} the precondition implies the postcondition. Thus, a false precondition allows any transition. Moreover, an undefined precondition results in an undefined operation specification.

Furthermore, we define a totalized operation specification, enforcing the result of the operation call or invocation to be undefined, if the precondition is undefined:

DEFINITION 3.26 (TOTALIZED OPERATION SPECIFICATION) Let *op* be an operation specified by the precondition $\text{pre}_{op}\ self\ a_0 \cdots a_n$ and the postcondition $\text{post}_{op}\ self\ a_0 \cdots a_n\ result$. The *totalized operation specification* semantics is defined as

totalized operation specification

$$
\begin{aligned}
S_{op}^{tot}\ self\ a_0 \cdots a_n\ result \equiv \tau \vDash \texttt{if}\ & \partial(\text{pre}_{op}\ self\ a_0 \cdots a_n) \\
& \wedge (\text{pre}_{op}\ self\ a_0 \cdots a_n) \\
\texttt{then}\ & (\text{post}_{op}\ self\ a_0 \cdots a_n\ result) \qquad (3.102) \\
\texttt{else}\ & \not\partial\ result \\
\texttt{endif}\ &
\end{aligned}
$$

where in both definitions we replace all accessor occurring in the precondition pre_{op} by their `@pre`-variant. □

The totalized operation specification can be used alternatively. It is preferable for methodological issues, namely for proofs of specification consistency.

3.9 OPERATION CALLS

We distinguish built-in operations (i. e., all library operations such as the logical operation $\neg\, x$, the arithmetical operation $x + y$ or the operation $X \cup Y$ on the collection types) and user-defined operations declared in class diagrams. From a perspective of a user, our framework forbids the overriding of the built-in operations. Thus it is obvious that the decision which operation has to be "executed" can always be resolved statically. We call this an *operation call*.

operation call

In contrast, user-defined operations can be overridden. Moreover, this is considered to be a main feature of object-oriented programming. Overriding results in situations where it is not possible to decide statically which implementation should be executed. We speak in such situations of an *operation invocation*, which we will discuss in more detail in Section 3.10. But also for user-defined operations there are situations where we can resolve the implementation statically, and thus only have to address an operation call. In this section, we will discuss which support our framework offers for operation calls to user-defined operations.

operation invocation

For operation calls we define

$$\text{Call}\, f \equiv \lambda\, \tau.\ \varepsilon\, x.\ f(\lambda\, \tau.\ x)\ \tau = \mathtt{t}\ \tau \tag{3.103}$$

with type $Val_\tau(\alpha) \Rightarrow \mathtt{Boolean}_\tau \Rightarrow Val_\tau(\beta)$. Here, Hilbert's ε-operator selects an eligible "implementation" fulfilling the operation specification.

For supporting recursive operation calls we extend the theory of well-founded orders and the well-founded recursor wfrec from HOL. This recursor allows for the conservative definition of a particular class of recursive functions, i. e., functions that fulfill the principles of well-founded recursion. Informally, a function is well-founded recursive, if the arguments of the recursive call are smaller and the ordering is well-founded. For example, Winskel [117] explains the formal details of well-founded recursion.

For introducing recursive calls in a conservative way, we extend the standard HOL methodology of well-founded recursion to object-oriented specifications. In particular, we define:

$$\text{Wfrec} \equiv \text{liftWf}\left(\lambda\, f\ x.\ \text{wfrec}\left\{(x, y)\middle|^{\ulcorner} r\ x\ y^{\urcorner}\right\}\ f\ x\right) \tag{3.104}$$

where liftWf is a specialized operator, similar to the already introduced lift_0, lift_1, …operators, lifting the type of wfrec. As its first argument, Wfrec expects a measurement, given as binary operation with a Boolean result and as second it expects a functional representing the recursive operation. Overall, the type of Wfrec is

$$\begin{aligned}\big(V_\tau(\alpha) \Rightarrow V_\tau(\alpha) \Rightarrow \mathtt{Boolean}_\tau\big) \\ \Rightarrow \big((V_\tau(\alpha) \Rightarrow V_\tau(\beta)) \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)\big) \\ \Rightarrow V_\tau(\alpha) \Rightarrow V_\tau(\beta)\,.\end{aligned} \tag{3.105}$$

The overall idea of both wfrec and Wfrec is to abstract away the occurrence of the recursive call.

As an example, let us assume the following postcondition for an operation m of class `A`, e. g., given as OCL specification:

```
context A::m(x:Integer):Integer
  post: result = if x > 0
                 then x * (self.asType(A)).m(x-1)
                 else 1
                 endif
```

By specifying `self.asType(A)` we always cast the context object to an instance of class `A` and thus we have a statically resolvable call here. We follow now the idea of abstracting the occurrence of the recursive call

away by defining a recursor-variant of this postcondition. The idea is to abstract away the occurrence of the recursive call:

$$\begin{aligned}&\mathrm{post}_m^{\mathrm{rec}}\ f\ \mathit{self}\ x\ \mathit{result} \equiv \\ &\qquad \mathit{result} \triangleq \texttt{if}\ x > 1\ \texttt{then}\ x \cdot (f\ \mathit{self}\ (x-1))\ \texttt{else}\ 1\ \texttt{endif}\end{aligned} \tag{3.106}$$

and to build the operation specification notions on top of it, based on the operation specifications defined in Definition 3.25 and Definition 3.26. For example, based on the operation specification with conjunct semantics (Equation 3.101), we define:

$$\begin{aligned}&\mathrm{S}_m^{\text{conj-rec}}\ f\ \mathit{self}\ x\ \mathit{result}\ \tau \equiv \\ &\qquad \tau \vDash \left(\mathrm{pre}_m\ \mathit{self}\ x \wedge \mathrm{post}_m^{\mathrm{rec}}\ f\ \mathit{self}\ x\ \mathit{result}\right).\end{aligned} \tag{3.107}$$

In this case, for direct recursive calls, the corresponding operation specification is defined as $\mathrm{S}_m^{\mathrm{conj}} \equiv \mathrm{Wfrec}\ M\ \mathrm{S}_m^{\text{conj-rec}}$ where M is an ordering, such as

$$\begin{aligned}M \equiv \lambda\, x\ y.\ &\texttt{if}(x < 0) \vee (y < 0) \\ &\texttt{then f} \\ &\texttt{else}\ x > y \\ &\texttt{endif}\ .\end{aligned} \tag{3.108}$$

If this ordering is well-founded, from this definition, the original user-specified postcondition follows from this definition. Since the critical call is now incorporated into the well-founded recursion construction, the definition is conservative; and provided the user gives a suitable ordering, it can be shown that the desired specification follows from the constructed definitions.

3.10 OPERATION INVOCATIONS

It is possible in our framework to describe any static resolution strategy explicitly using the previously defined type and kind test predicates on the arguments. While this would be probably sufficient for many verification tasks, we are also interested in the limits of a conservative strategy for late-binding. In this section, we show how the semantics of strict operation invocations is encoded using the semantic combinator for strict invoke defined in Section 3.2.2.

3.10.1 *The Invocation Encoding Scheme*

INITIAL OPERATION DEFINITION. In the following, we show the semantic representation scheme of invocation for, potentially overridden, user-defined operations by an example. Figure 3.9 on the next page

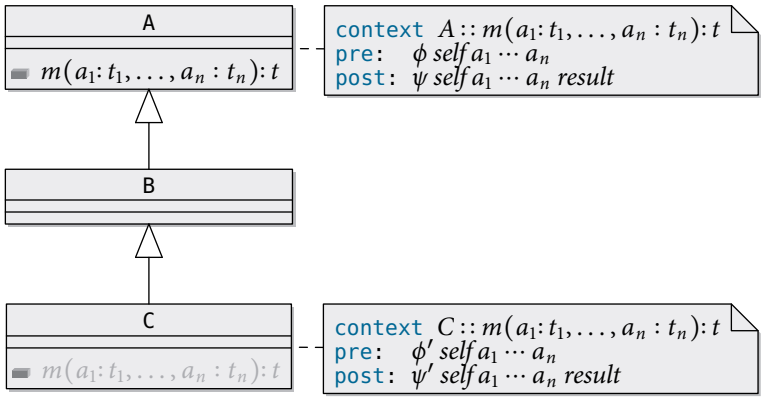


Figure 3.9: The specification of the method m of class C overrides the variant of that method already defined in class A.

illustrates our example: We assume the three classes A, B and C, where C inherits from B and B inherits from A. Further, we assume that an operation m, in the topmost class A, with arguments $a_1 : t_1, \ldots, a_n : t_n$, and return type t is specified using the precondition Φ, and postcondition Ψ.

operation table

While encoding the class model, we generate for each operation a *operation table*. The operation table is a look-up table collecting the overridden operation specifications. In our example, we define:

$$\mathrm{OpTab}_{\mathrm{m}} :: A \text{ set} \rightharpoonup [V_\tau(A), V_\tau(t_1), \ldots, V_\tau(t_n), V_\tau(t)] \Rightarrow \mathtt{Boolean}_\tau \,. \tag{3.109}$$

Here, $_ \rightharpoonup _$ stands for the type of *partial maps* from the HOL library. The main difference of partial maps compared to total functions $_ \Rightarrow _$ is that partial functions have a domain operator dom with type $\alpha \rightharpoonup \beta \Rightarrow \alpha$ set. Additionally, the axiom

$$\mathrm{OpTab}_{\mathrm{m}}\ A \equiv \mathrm{Some}(\mathrm{S}_m) \tag{3.110}$$

is generated, where S_m is one of the operation specifications defined in Section 3.8 and where A is the characteristic type set of the class A. In the concluding subsection, we will discuss the conservativity issue for this type of axioms which is similar, but technically unequal to a constant definition since the table is not defined once and for all, but point-wise for a finite set of arguments.

INHERITANCE OF OPERATION. Now we consider the case that the class B is declared, but the operation m is not overridden, i. e., inherited from class A. This leads to the axioms:

$$\mathrm{OpTab}_{\mathrm{m}}\ B \equiv \mathrm{Some}(\mathrm{S}_{\mathrm{m}}) \,. \tag{3.111}$$

Recall that due to our object universe construction, the type of B is an instance of the type of A even if the class `B` has been inserted into the system in a later stage than the compilation of `A`, i. e., A and B live in different universes. Moreover, in the later universe, the property $B \subset A$ holds and has been proven automatically.

OPERATION OVERRIDING. Now we consider the case of an operation overriding. Here, a new declaration introduces a new specification (for an already specified operation) for the operation m for class C (and its subclasses) with precondition P' and postcondition Q'. Again, see Figure 3.9 on the facing page for details. Analogously to the overriding case, the axiom

$$\text{OpTab}_{\text{m}}\ C \equiv \text{Some}(S'_{\text{m}}) \tag{3.112}$$

is constructed where S'_{m} is the operation specifications describing the new, overridden behavior.

3.10.2 *Considering Conservativity*

The axioms generated in the previous sections are conservative; however, they do *not* fit into one of the standard schemes such as constant definition (the argument of OpTab_{m} and $\text{OpTab}_{\text{m-tot}}$ are a changing constant not allowed in this scheme). Rather, it is a *finite family* of constant definitions, where the overall type is refined from universe to universe. In the following, we characterize the syntax of this axiom scheme and sketch a proof of conservativity for it.

finite family of constant definition

DEFINITION 3.27 (FINITE CONSTANT DEFINITION FAMILY) A *finite family of constant definitions* is a theory extension (Σ, A) where Σ is a constant declaration $c :: \tau_1 \to \tau_2$ and A is a finite sequence of axioms of the form $c\ D_1 \equiv E_1 \ldots c D_n \equiv E_n$ where D_i and E_i are closed expressions. One further optional rule, the *catch-all rule*, has the form: $X \notin \{D_1, \ldots, D_n\} \Longrightarrow c\ X \equiv E_{n+1}$. Furthermore, the following conditions must be satisfied:

1. c does not occur in E_1.
2. c can only occur in E_j (in fact: in no defining expression E in a definition except that the catch-all rule is known) in the form $c\ D_i$ with $i < j$.
3. The type of c in axiom j must be an instance of the type of c in axiom i with $i < j$.
4. All type variables occurring in any type of a sub-term of E_j must occur in the type of $c\ D_j$
5. $D_j = D_k \Longrightarrow j = k$, i. e., the D_j must be pairwise disjoint.

□

The catch-all rule is used when all classes that provide, either directly, by inheritance, or by overriding, an implementation of the operation, are *finalized*. Thus, none of these classes can be used as starting point for further extensions.

conservative

THEOREM 3.1 (CONSERVATIVITY) *A finite family of constant definitions is* conservative*, i. e., provided the original theory* (Σ, A) *is consistent ("has models"), the theory* $(\Sigma \cup \Sigma', A \cup A')$ *extended by the extension* (Σ', A') *is also consistent.* □

PROOF In case that there is a catch-all rule, translate the constant definition family into a family of constant definitions with constants c_{D_j}.

In case that there is no catch-all, the constant definition family can be replaced equivalently by the constant definition

$$c\ X \equiv \text{if } X = D_1 \text{ then } E_1 \text{ else if } \ldots \text{ else } E_{n+1}\,. \tag{3.113}$$

The axiom given in Equation 3.113 represents a constant definition family since partial maps $\alpha \rightharpoonup \beta$ are just a synonym for $\alpha \Rightarrow \beta$ option. The pairwise disjointness follows from the full inclusion of the characteristic sets assured by construction. ■

Thus, for non-recursive operation invocations the object-oriented concepts of overriding can be supported conservatively. In the next section, we will discuss the limitations of supporting of recursive operation calls and operation invocations conservatively.

3.11 LIMITS TO RECURSIVE INVOCATIONS AND CALLS

In this section, we will discuss how the use of recursive calls, and invocations in particular, must be restricted to ensure conservativity of our framework.

But first, let us briefly reconsider why conservativity is fundamental for a formal framework for object-orientation. Overall, an object-oriented model can be inconsistent in the sense that there is no state satisfying all invariants. However, no axiom generated during encoding of a model into our framework should introduce a logical inconsistency into the meta-logic HOL. A logical inconsistency of the meta-logic results in an unsound reasoning, i. e., one can prove falsity. A proof over the consistency of an object-oriented model in the above sense should be valid in any case, independent of any generated axioms. Thus, we require that method definitions in class diagrams satisfy the requirements of a family of finite constant definition.

As we do not introduce any axioms for operation calls, these are unproblematic, as long as recursion is well-founded (as discussed in Section 3.9). Nevertheless, the requirement enforced by Definition 3.27 on page 93 for operation invocations has several consequences, especially on the form of admissible recursive operation invocations in our framework: item 1 and item 2 rule out a general recursive invocation of the operation to be specified. Consider again the operation m specified in class `C`, which overrides the operation m already defined in class `A`. Assume that the postcondition ψ' of m defined in class `C` is given by

$$\psi'\ self\ a_1 \cdots a_n\ result \equiv \\ \big(result \triangleq (m\ self\ a_1 \cdots a_n\ ) + 1\big) \wedge \big(\partial(m\ self\ a_1 \cdots a_n\ )\big). \quad (3.114)$$

On the right-hand side in the definition of ψ' the (overridden) operation m itself occurs. Due to late-binding, it is not possible to decide statically which concrete operation specification (in our case either the one specified in class `A` or the one specified in class `C`) must be use to "unfold" this reference.

Following Section 3.2.2, the operation invocation is represented by:

$$\mathrm{Call}(\mathrm{invokeS}\ \mathrm{C}_{[\mathrm{A}]}\ \mathrm{OpTab}_{\mathrm{m}}\ self\ a_1\ \ldots\ a_n) \quad (3.115)$$

where the occurrence of $\mathrm{OpTab}_{\mathrm{m}}$ is an instance of item 2 of Definition 3.27 on page 93.

The example also shows why this kind of syntactic restriction is necessary: from the recursive equation

$$(m\ self\ a_1 \cdots a_n\ ) = (m\ self\ a_1 \cdots a_n\ ) + 1\,. \quad (3.116)$$

and the definedness of the result one can infer $1 = 0$ in the object-logic, and then one can prove false in HOL, and then simply everything from there.

In a proof-environment, recursive definitions are potentially dangerous. Furthermore, our framework is designed to live with the open-world assumption, i. e., with the potential extensibility of object universes, as a default; further restrictions such as finalizations of class diagrams or a self-restriction to Liskov's Principle [73] may be added on top, but the system in itself does not require them. This has the consequence that even in the following variant

$$\begin{aligned} \psi'\ self\ a_1 \cdots a_n\ result \equiv result \triangleq\ & \mathtt{if}\ p\ a_1 \\ & \mathtt{then}(m\ self\ a_1 \cdots a_n\ ) + 1 \\ & \mathtt{else}\ 0 \\ & \mathtt{endif} \end{aligned} \quad (3.117)$$

the termination for the invocation

$$m\ self\,(q\ a_1)\ a_2 \cdots a_n \tag{3.118}$$

is undecidable (even if the operations p and q are known and terminating): a potential overriding may destroy the termination of this recursive scheme.

In form of a pre-translation process, operation specifications with a limited form of recursive invocations can be converted into the format that satisfies the constraints of a finite constant definition family. In fact, these invocations are calls to a specific, statically resolvable, operation. These limited forms are assumed to occur in the postcondition ψ' and can be listed as follows:

1. calls to superclass operations, i. e.,

$$m\ (self_{[\mathrm{A}]})\ x_1 \cdots x_n\,. \tag{3.119}$$

   This invocation can be translated into the non-recursive call

$$\mathrm{Call}\ \mathrm{S_m}\ \ self\ x_1 \cdots x_n\,. \tag{3.120}$$

2. direct recursive well-founded invocations, i. e.,

$$m\ (self_{[\mathrm{C}]})\ x_1 \cdots x_n\,. \tag{3.121}$$

   This invocation can be translated into a directly recursive call which can be handled as described in Section 3.9.

In case of a finalized class hierarchy, the number of possible operation specifications an operation invocation can refer to is fixed, i. e., the operation table is not extensible. In that case, in our example, the invocation

$$\mathrm{invokeS}\ C\ \mathrm{OpTab_m}\ \ self\, a_1\ \ldots\ a_n \tag{3.122}$$

can by replaced by the case-switch:

$$\begin{array}{l} \text{if isType}_{\mathrm{A}}\ self\ \text{then S}_m \\ \text{else if isType}_{\mathrm{B}}\ self\ \text{then S}_m \\ \text{else if isType}_{\mathrm{C}}\ self\ \text{then S}'_m \\ \text{else } \bot \end{array} \tag{3.123}$$

which selects the suitable operation specification by the type of the context object represented by *self*.

Summing up, conservativity implies that only limited forms of recursive invocations are admissible in our framework. In an open-world (no class

finalization so far), only operation calls can be treated. In a (partially) closed-world (the class hierarchy has been finalized), an invocation can be expanded to a case-switch considering the dynamic type of *self* over calls. For supporting invocations, we do not require that the complete model is finalized. It is sufficient that the parts of the models containing invocations are finalized. For example, our framework allows models where

- parts that do not contain recursive operation specification are extensible,
- parts that only contain recursive invocations that can be statically resolved into calls are extensible, and
- parts that contain invocations are finalized. As invocations are only conservative, if the operation table is fixed, all classes that are related by inheritance must be finalized or sterilized.

Here the flexibility allowing partially closed-worlds allows for deciding on a case by case basis which strategy to follow.

3.12 SPECIFYING FRAME PROPERTIES

When using contracts, or pairs of preconditions and postconditions for state transition there arises the need to specify *exactly* which parts of the system are allowed to be modified and which have to stay unchanged, i. e., we have to specify the *frame property* of the system. Otherwise, arbitrary relations from pre-states to post-states are allowed. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i. e., to specify the frame properties of system transition. As an example, consider Figure 3.10 with an particular focus on the specification of the operation `deposit` of the class `Account`. This specification only describes which part of the system should change, i. e., the balance of the context object (which is an `Account` object) should be increased. But this is not specified, which parts of the system should remain unchanged, e. g., the `id` of the context object.

frame property

One solution to solve this frame problem would be an implicit invariability assumption on the meta-level which would somehow express "all things that are not changed explicitly remain unchanged." But this is neither formal nor precise and thus not usable within a formal framework for object-oriented specifications.

Another possibility is to enumerate, in the postcondition of the operation, all path expressions that should remain unchanged, e. g., in our example a first attempt to do so would be:

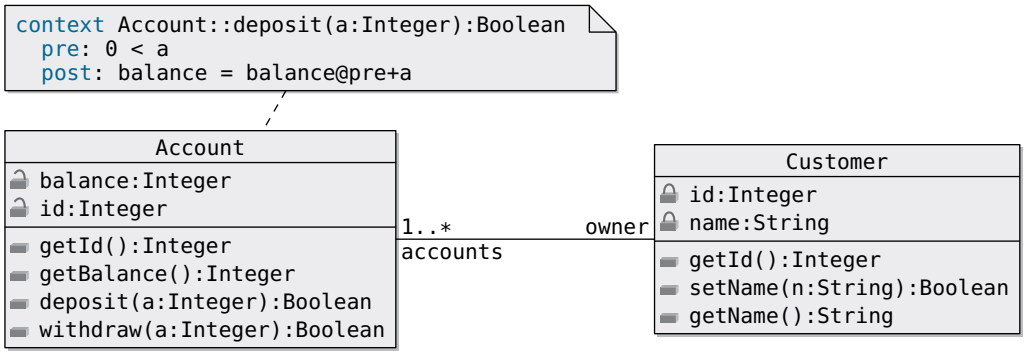


Figure 3.10: Consider a state transition constrained by the operation specification for the operation `deposit`. Obviously, only the attribute `balance` of one specific object should be changed, but how can this be specified?

```
context Account::deposit(a:Integer):Boolean
  post: balance = balance@pre+a
  post: id = id@pre
  post: owner = owner@pre
  post: owner.id = owner@pre.id@pre
  post: owner.name = owner@pre.name@pre
```

But this is also not sufficient, as it would still not describe if objects not related to our context object (of type `Account`) must remain unchanged or not. Enumerating all classes (and attributes) using static path expressions (e. g., `Customer::name = Customer::name@pre`) is tedious and moreover leads to contradictions if the `name` attribute of the owner of the context object should be changed.

Thus, we prefer a different approach for describing the frame property of a system transition. We propose an operation modifiedOnly _ that allows for explicitly enumerating all objects of a given state that are allowed to be modified. As a prerequisite, we define a predicate OidOf for accessing the object identifier of an object:

$$\text{OidOf}\ \tau\ X \equiv \{x.\ (\tau\ x) = \text{Some}(\text{mk}_{\text{Object}}\ X)\} \tag{3.124}$$

which allows a uniform definition of modifiedOnly _ for the referential universe and the non-referential universe:

$$\begin{aligned}&\text{modifiedOnly}\ X \equiv \lambda(\tau, \tau').\\ &\quad \llcorner \forall\, i \notin \left(\bigcup(\text{oidOf}\ \tau) \setminus \ulcorner \text{Rep}_{\text{Set}}(X(\tau, \tau'))\urcorner\right).\ \tau\ i = \tau'\ i \lrcorner .\end{aligned} \tag{3.125}$$

Thus requiring modifiedOnly $\varnothing$ in a postcondition of an operation allows for stating explicitly that an operation is a query. Further, the following

equivalence holds:

$$(\text{modifiedOnly}\, X) \wedge (\text{modifiedOnly}\, Y) = \text{modifiedOnly}(Y \cap X)\,. \quad (3.126)$$

The two main advantages of defining the set of modifiable objects using their semantic representations are:

alias

1. There may be *aliases*, i. e., two path expressions that point to the same object. Therefore, forbidding the assignment to one expression denoting an object does not imply that it is unchanged; it could be changed via another path expression (reference). Our approach solves the *alias problem* by referring to values and not to (not necessarily unique) names. This paves the way to a smooth integration of Hoare Calculi [23].

2. We allow recursive functions traversing object structures (such as associations), such that sets of modified objects may be collected and specified recursively. For example, we could specify, that an operation of the class `Customer` is only allowed to change those objects (accounts) reachable via `self.accounts` which have a balance less than a specified value.

Since frame properties are an important part of the system specification, and thus it is not astonishing, that most precondition/postcondition specification languages provide a means to specify frame properties, e. g., JML [70, 71] allows for explicitly enumerating a list of references [71] to be assignable or not. We extend this schema by allowing arbitrary predicates that construct the set of all objects (or references) that are allowed to be modified. Our construction, which is in principle an extension Δ-operator in Object-Z, is strictly more powerful than just enumerating references explicitly.

3.13 DISCUSSION

In this section, we will summarize the features and limitations of our framework for object-oriented specifications. But first, we explain the two main components of our framework more detailed: the object store and the object-oriented constraint language:

object store

An *object store* provides the core notion of object-oriented data structures, e. g., a formalization of classes and instances including for example concepts like inheritance and subtyping. In particular, the object store provides the formal semantics of path expressions, i. e., expressions navigating through a concrete object structure. By combining an object store with an object-oriented constraint language, one can restrict the set

of valid object-structures semantically, i. e., specific states are ruled out based on their semantics. Without an object-oriented constraint language, object-structures can only be ruled out by their structure, i. e., if they not well-typed.

object-oriented constraint language

Bertino et al. [16] introduce the concept of an object-oriented query language which is closely related to the concept of an *object-oriented constraint language* we used in this chapter. In our understanding, the main characteristics of an object-oriented constraint language are:

- an underlying logic supporting undefinedness; four different types of equality, based on identity and values, each in a strict and non-strict variant;
- a typed set theory and basic datatypes; path expressions for navigating in the graph representing the object-structures;
- different predicates for restricting instances in a specific system state or class hierarchy; and
- support for calling or invoking user defined operations.

Allowing the user for calling purely functional operation specifications, i. e., query operations, within the constraint language itself allows for a certain degree of logical extensibility. Overall, in contrast to a query language, a constraint language *constrains* the set of valid system states, i. e., the set of possible object structures.

In the following we summarize the key features of our framework. The most important feature of our formalization of an object store is its extensibility, i. e., its support for direct proof reuse after extending the class model. The limitations of this constructions are in particular:

- The reuse of proof objects is limited to the extension of class hierarchies. Merging different class hierarchies requires re-proving all properties, e. g., by re-playing the proof scripts.

  This limitation is a consequence of the fact that type sums are not commutative; thus, the order in which β-instances were created is relevant and leads to the generation of type-incompatible constructors and accessors in different class universes for the same class. On the other hand, since the process of merging class hierarchies, which also includes the detection of shared subclasses, is a highly nontrivial one, one cannot expect that this phenomenon can be coped smoothly in a type system not built for this purpose.

  The two most common cases of merging different class models are probably the parallel extension of class models and the reuse of libraries. For these two scenarios, the proof scripts can be replayed

automatically, i. e., without user interaction. Moreover, these two scenarios can also be resolved in advance by introducing abstract classes (or only an additional type variables) that can act as extension points.

- The typing of equality expressions in our formalization is more liberal than usual, e. g., in OCL. In particular, depending on the definition of the root of the class hierarchy (e. g., `OclAny` or `Object`), referential or (shallow) value equality result for the constraint language (see Section 3.6).

- Conservative support for late-binding is restricted to parts of the class hierarchy that are finalized or sterilized. It seems to be impossible to encode *conservatively* an extensible object structure supporting late-binding. Nevertheless, our framework supports partial closed-worlds and thus a high degree of flexibility (see Section 3.11).

- As most object-oriented programming languages, our framework does not support multiple inheritance. Extending our universe construction for supporting multiple inheritance is a highly nontrivial task.

  Nevertheless, multiple subtyping based on interfaces, as found in Java or the UML is supported by our framework. As an interface only specifies operation that must be implemented by classes realizing the interface support for subtyping is reduced to showing the usual proof obligations for subtyping: the precondition of the class must imply the precondition of all interfaces it realizes and the postcondition of the interface must imply the postcondition of the class. Of course, this does not need a special support within the object encoding and therefore, we can support multiple subtyping easily.

Since we aim for a framework that is object-oriented, it is reasonable to demand that our frameworks should support the concepts introduced in Section 2.1. From these concepts, only a few were not yet discussed in this chapter, in particular:

- Violating the *encapsulation* of a class, i. e., accessing path which is protected by access specifiers like `private`, `protected` or `public`, can be checked purely statically. Thus, on the one hand an import mechanism for specifications can already reject specifications that violate disobey the access specifiers. On the other hand, it is debatable, if one should pay attention to these concepts during formal analysis. One could even think about interpreting the forbidden

accesses (i. e., the access to a private attribute from the outside) as an undefined expression and as such encoding access specifiers into our framework. We decided against this for two reasons: first, it would result in a quite complex reasoning, and second, the static check is easy and efficient. Thus, we see the access specifiers as a kind of syntactic well-formedness rule the model has to be compliant with.

- At the moment, our framework has no direct support for *associations*. They can be represented by their association ends together with additional class invariants. A pre-compilation step can implement this conversion. We will discuss this problem in more detail in Section 4.3.1.

- *Packages* in general and *namespaces* in particular are a purely syntactic concept for avoiding name clashes. Thus it is not necessary to provide semantical support for them, albeit our implementation maps them to the namespace concept of Isabelle/HOL and thus generates names for the logical constants that are divided into hierarchic namespaces similar to the one used in the user model.

In the next chapter, we will show, how this framework can be used both for giving a formal, machine-checked semantics for UML/OCL and also for developing an interactive theorem prover for UML/OCL.

A FORMAL SEMANTICS FOR UML/OCL

4

In this chapter, we show how our framework presented in Chapter 3 can be used to provide both a standard-compliant, machine-checked, formal semantics and an interactive proof environment for the *Object Constraint Language* (OCL), called HOL-OCL.

4.1 CHALLENGES

Formalizing, in a machine-checkable way, a real-world standard for a specification language is always a technically challenging task. This is especially true if, as in the case of OCL such a standard was not originally developed with formalization in mind. The formalization of such a standard requires an in-depth analysis of the already existing description of the language, resulting in:

- A description of missing parts of the current standard together with proposals to fill the gaps.
- A description of inconsistencies of the current standard together with proposals to resolve them.
- Proposals for alternative definitions or extensions of the language.

Moreover, all decisions made, e. g., for resolving inconsistencies or fixing gaps, should capture somehow the informal intention of the authors of the original standards. This requires, if possible intensive discussion with them, or an intensive analysis of examples and actual usages of the language being formalized.

Our formalization of the OCL standard [88, Appendix A] we present in this chapter is based on the framework presented in Chapter 3. It provides the following benefits over a paper-and-pencil formalization:

A Consistency Guarantee. Since all definitions in our formal semantics are conservative and all rules are derived, the consistency of the complete framework is reduced to the consistency of HOL for the

entire language. In particularly, this holds also for HOL-OCL, the interactive theorem prover we developed for UML/OCL.

A Technical Basis for a Proof-Environment. Based on the derived rules, control programs (i. e., *tactics*) implement semi-automated reasoning over OCL formulae; together with a compiler for class diagrams, this results in a general proof environment called HOL-OCL. Its correctness is reduced to the correctness of a (well-known) HOL theorem proving system.

Proofs for Requirement Compliance. The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative part [88, Appendix A]. We provide formal proofs for the compliance of our OCL semantics with these requirements.

Formalization Experience. Since the semantics of the whole language is formalized and machine-checked, we extend or modify the semantics while preserving consistency. This conservative formalization allows for extending the language or examining semantic without the risk of introducing inconsistencies in the language.

In this chapter, we will present a machine-checked semantics for OCL in the context of UML class models which is based on the framework presented in Chapter 3. We will start, in Section 4.2, with a brief overview of the OCL standard. In Section 4.3 we present our formalization of the OCL semantics and show in Section 4.4 why our semantics complies to the standard. Moreover, we propose several extensions of the OCL in Section 4.5.

4.2 A NOTE ON OCL STANDARDS

In this section, we give a brief overview of the official OCL standard [88] In particular, report on the historic development and the current state of a formal semantics for OCL as described in the different versions of the OCL standard.

4.2.1 *A Historic Overview*

The UML and OCL standards are developed in an open process by the *Object Management Group* (OMG). Such an open process leads to variety of (intermediate) standardization documents [67]; especially for UML and OCL, which have a long history. For example, OCL was already introduced in 1997 as a supplement to the UML standard. The different versions of OCL 1.x are very close to each other, containing mainly an informal

motivation of the intended use and semantics of OCL together with a formal grammar of its concrete syntax. Reading these versions of the standard leaves more questions open than it answers. These shortcomings and open questions, like the handling of undefinedness, or recursion, were discussed [37, 53, 74] in academia and this discussions clearly fertilized the development towards OCL 2.0. Especially the work of Richters [100] served as formal underpinning of the OCL 2.0 development. It was a major break-through in the process of defining a formal semantics for OCL. Many problems, like the handling of undefinedness, were clarified during the OCL 2.0 standardization process, some questions however, like the handling of recursion, are still unsolved.

In the following, we present a formalization based on the following two documents:

1. *OMG Unified Modeling Language Specification* (excluding Chapter 6 which describes OCL 1.5) Version 1.5 [90].

2. *UML 2.0 OCL Specification* [88], denoted as "OMG Final Adopted Specification."

More recent versions, especially [89], are an ad-hoc attempt to align the UML 2.0 with the OCL 2.0. Among many other annoyances, new datatypes are introduced without giving them a consistent semantics. For example, besides `OclUndefined` (called `invalid`), also an exception element, called `null`, is introduced. On the one hand, the intention of the authors of the standard [89] is to give `OclInvalid` a strict and `null` a non-strict semantics with respect to collection type constructors: "Note that in contrast with `OclInvalid` `null` is a valid value and as such can be owned by collections." [89, pp. 36]. Nevertheless, `null` is still strict with respect to other operations: "Any property call applied on null results in OclInvalid, except for the operation oclIsUndefined()" [89, pp. 138]. On the other hand, both `invalid` and `null` conform to all classifiers, in particular `null` conforms to `invalid` and vice versa. Moreover, the conforms relationship is antisymmetric and therefore `invalid` and `null` are actually indistinguishable. Considering that many of these changes were made without giving much thought to their impact on the existing specification, the version [89] of the OCL standard represents a considerable step back with respect to consistency and potential for formal semantics. Moreover, the problems we report in this chapter, are still valid for current versions of OCL.

4.2.2 *The Role of Semantics in the Standard*

We claim to provide a semantic representation that is compliant with the semantics presented in the OCL standard. In this section, we make our

claim more precise, in particular we have to discuss to which parts of the standard we claim to be compliant. Standards issued by the OMG are usually divided into normative parts and informative parts. The *normative* parts define the standard. In contrast, the *informative* parts of the standard are not normative, i. e., they are thought of containing motivating and background material. Thus, any OCL compliant work must honor the normative parts of the OCL standard whereas it can ignore the informative ones.

normative
informative

The semantics of OCL appears in the following chapters of the OCL standard [88]:

Chapter 7 "OCL Language Description": This is a *informative* chapter motivating the use of OCL and introduces it in an informal way, mostly by showing examples. We used this chapter mainly for catching the intentions of the standard in cases where the other parts of the standard are unclear or contradictory.

Chapter 10 "Semantics Described using UML": This *normative* chapter describes the "semantics" of OCL using the UML itself. Merely an underspecified "evaluation" environment is presented. Nevertheless, some of the information presented in this chapter is helpful for formalizing the standard. Moreover, the chapter title reveals that meaning of "semantics" is not always that obvious, Harel and Rumpe [55] discusses this issue in more detail.

Chapter 11 "The OCL Standard Library": This *normative* chapter is, in our opinion, the best source of the normative part of the standard describing the intended semantics of OCL. It describes the semantics of the OCL expressions as requirements (in form of pairs of preconditions and postconditions) they must fulfill. Overall, we prove these requirements for our embedding and thus show that our embedding satisfies these requirements.

Appendix A "Semantics": This *informative* appendix defines the syntax and semantics of OCL formally in a textbook-style mathematical notion. It is mostly based on the work of Richters [100].

Overall, we criticize the semantic foundations of the standard for several reasons:

1. The normative part of the standard does not contain a formal semantics of the language.

2. The consistency and completeness of the formal semantics given in "Appendix A" is not checked formally.

3. There is no proof, neither formal nor informal that the formal semantics of informative "Appendix A" satisfies the requirements of the normative chapter 11.

Nevertheless, we think the OCL standard [88] is mature enough to serve as a basis for a machine-checked semantics and formal tools support.

4.3 A MACHINE-CHECKED OCL SEMANTICS

In this section, we present a formal, machine-checked semantics for OCL based on our framework. As we already presented our framework in Chapter 3, we only have to choose building blocks that match the OCL semantics as described in the standard. In Section 4.4 we will discuss the standard compliance of our semantics in detail.

For presentational reasons, we introduce an explicit semantic function into our shallow embedding. Of course, with respect to HOL, this is just the identity, i. e.,

$$\mathrm{Sem}[\![x]\!] \equiv x \qquad \text{with type } \alpha \Rightarrow \alpha\,. \tag{4.1}$$

4.3.1 *Encoding the Underlying Data Model*

OCL heavily relies on an underlying, user-defined class model, thus we start our formalization by fixing a semantics for class models. As OCL describes itself as a constraint language for the UML, the underlying data model should be compatible with the UML standard [90].

THE OBJECT UNIVERSE. Using our framework, we have to decide if OCL is based on a sharing semantics for object-structures (i. e., the non-referential universe) or a creation semantics (i. e., the referential universe). An interpretation of such a universe construction is given in the standard:

> Each object is uniquely determined by its identifier and vice versa. Therefore, the actual representation of an object is not important for our purposes. *(OCL Specification [88], page A-7)*

On the one hand, this formulation suggests that there is no difference between an object and its value, which would lead to a sharing semantics. On the other hand, the normative part always identifies objects by a reference to it, e. g.:

> If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self*. *(OCL Specification [88], page 15)*

As the referential universe is close to the usual programming language semantics of object constructors we use it as the basis for our formalization of OCL. Furthermore, in this setting, the reference to an object in the store can always be reconstructed which paves the way for reference types as for example in Java.

Therefore, we define the supertype of all types of the user-defined UML models, which is called `OclAny` in OCL, based on the referential variant (item 2) in Definition 3.11, i. e.,

$$\alpha\ \texttt{OclAny} := \left((\text{OclAny}_{\text{tag}} \times \text{oid}) \times \alpha_{\perp}\right). \tag{4.2}$$

Overall, we suggest to resolve this ambiguity of the standard in favor of the referential setting, which we also see as the default HOL-OCL configuration.

PATH EXPRESSIONS: ACCESSORS. The path expressions of OCL, i. e., attribute accessor can be directly defined by the constructors and accessors of our framework, see Section 3.4.3. In more detail, we define path expressions on the basis of the level 2, as introduced in Section 3.5.3. Using a level 2 interpretation of path expressions guarantees that a path expression is either undefined or represents an instance fulling the invariants of its type. For example, assume the access to a class attribute `id`; we interpret the path expression `self.id` as follows:

$$\text{Sem}[\![\texttt{self.id}]\!] \equiv \mathit{self}.\,\text{id}^{(2)}\ . \tag{4.3}$$

In Section 4.3.7 we discuss in more detail, why we propose the level 2 constructs as basis of the OCL semantics.

TYPES, CASTING AND TYPE TESTS. The type-casts and type tests from of OCL can be directly defined by the corresponding operations of our framework, i. e.,

$$\text{Sem}[\![\mathit{self}\ \texttt{.oclIsTypeOf}\ (\text{T})]\!] \equiv \text{isType}^{(2)}_{\text{T}}\ \mathit{self}\,, \tag{4.4a}$$

$$\text{Sem}[\![\mathit{self}\ \texttt{.oclIsKindOf}\ (\text{T})]\!] \equiv \text{isKind}^{(2)}_{\text{T}}\ \mathit{self}\,, \text{ and} \tag{4.4b}$$

$$\text{Sem}[\![\mathit{self}\ \texttt{.oclAsType}\ (\text{T})]\!] \equiv \text{self}^{(2)}_{[\text{T}]}\ . \tag{4.4c}$$

Where the operations $\text{isType}^{(2)}_{_}\ _$, $\text{isKind}^{(2)}_{_}\ _$, and $_^{(2)}_{[_]}$ are constructed by applying the schema presented in Equation 3.81 on page 78 (Section 3.5.3), based on their the corresponding level 1 operations: $\text{isType}^{(1)}_{_}\ _$, $\text{isKind}^{(1)}_{_}\ _$, and $_^{(1)}_{[_]}$. Recall, the level 2 operations are based on semantic type and kind sets and thus include invariant checking. In Section 4.3.7 we discuss in more detail, why we propose the level 2 constructs as basis of the OCL semantics.

We model the type `OclVoid` implicitly using the lifting combinators and the type class bot, see Section 3.2.1 for details. Moreover, the types `OclModelElementType` and `OclType` are modeled implicitly, respectively replaced by the characteristic type set (of a type).

LIMITATIONS OF OUR OBJECT MODEL. The main limitations, compared to the UML standard, of our framework are:

- We limit inheritance to single inheritance and do not support multiple inheritance. This decision is a price we have to pay for supporting extensibility. Since multiple inheritance is known to introduce many new problems into object-oriented methods and moreover most object-oriented programming languages also forbid multiple inheritance. Moreover, we support classes implementing multiple interfaces and thus, we support a specific form of multiple subtyping. As this setting is similar to the situation for typed object-oriented languages like Java, we think this is a reasonable choice. Nevertheless, supporting multiple inheritance is not possible with the presented encoding scheme, thus this is a real limitation of our framework.

- We represent associations by their association ends together with some OCL constraints. Gogolla and Richters [44] present such conversions from graphical UML notations to OCL in more detail. Moreover, we assume that association ends belong to classes participating in an association. This is compliant to UML 1.5 [90], but not to compliant to UML 2.0. As there is at the moment neither a formal semantics for UML 2.0 (Broy et al. [19] report on first work in that direction) and moreover OCL 2.0 and UML 2.0 are not yet finally aligned. Overall, direct support for associations, e. g., representing them as relations, is not only desirable to be standard compliant, but also from a formal reasoning point of view. Thus, we see the direct support for associations, including the support for association classes, as future work which extends our framework naturally.

Furthermore, the OCL standard ignores visibilities on the specification level:

> The OCL specifications puts no restriction on visibility. In OCL all model elements are considered visible.
>
> *(OCL Specification [88], page 63)*

This decision conforms to our advice (see Chapter 3) for ignoring visibilities on the specification level.

4.3.2 *Primitive Datatypes*

Undefinedness is omnipresent in OCL, this gets especially clear if we look how the standard [88, Appendix A] introduces the primitive types, or basic types as they are called in the OCL standard:

> Let $\mathcal{A}^\star$ be the set of finite sequences of characters from a finite alphabet $\mathcal{A}$. The semantics of a basic type $t \in T_B$ is a function I mapping each type to a set:
>
> - $I(\mathit{Integer}) = \mathbb{Z} \cup \{\bot\}$
> - $I(\mathit{Real}) = \mathbb{R} \cup \{\bot\}$
> - $I(\mathit{Boolean}) = \{\text{true}, \text{false}\} \cup \{\bot\}$
> - $I(\mathit{String}) = \mathcal{A}^\star \cup \{\bot\}$
>
> *(OCL Specification [88], page A-9, definition A.14)*

This definition corresponds directly to the lifted datatypes introduced in Section 3.3.2, i. e., we identify the primitive types as follows:

$$\text{Sem}[\![\texttt{Integer}]\!] \equiv \texttt{Integer}_\tau\,, \tag{4.5a}$$
$$\text{Sem}[\![\texttt{Real}]\!] \equiv \texttt{Real}_\tau\,, \tag{4.5b}$$
$$\text{Sem}[\![\texttt{Boolean}]\!] \equiv \texttt{Boolean}_\tau\,, \text{and} \tag{4.5c}$$
$$\text{Sem}[\![\texttt{String}]\!] \equiv \texttt{String}_\tau\,. \tag{4.5d}$$

4.3.3 *Encoding Built-in OCL Operations*

The standard contains "principles" for the semantics of the operations, consider for example:

> In general, an expression where one of the parts is undefined will itself be undefined. *(OCL Specification [88], page 15)*

In other words, one could rephrase this semantic principle as "all operations are strict," be it for standard or for user-defined operations. The OCL standard requires as default for all operations to be *strict*, both for the case of built-in like the $_ + _$ on $\texttt{Integer}_\tau$ or user defined operations declared in class diagrams. Other "principles" are hidden in the semantic definitions; for example the passing of the context. Nevertheless, these "principles" motivated our decision for a combinator-style semantics approach in our framework. Overall, for $\texttt{Integer}_\tau$, $\texttt{Real}_\tau$, and $\texttt{String}_\tau$ we can directly use the semantics introduced in Section 3.3.2.

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	$\bot$	false	$\bot$	$\bot$	true	true
true	$\bot$	$\bot$	true	$\bot$	$\bot$	false
$\bot$	false	false	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	true	$\bot$	true	$\bot$	true	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 4.1: The formal semantics of the Boolean Operations as given in the OCL standard [88, p. A-12].

Also, the defining the operations on collections is straight-forward with one notable exception: higher-order constructs like quantifiers and the iterators need a slightly more complicated lifting process, [21] presents the details of this construction.

The core logic, i. e., the operations over the type Boolean are non-strict. The standard defines the semantics of these operations by giving truth tables (see Table 4.1). These truth tables describe the semantics of a Strong Kleene Logic (Definition 3.7 on page 57) as defined in Section 3.3.1. Thus we can directly reuse the definitions for a Strong Kleene Logic of our framework:

$$\mathrm{Sem}[\![x_0 \texttt{ and } x_1]\!] \equiv x_0 \wedge x_1 \tag{4.6a}$$
$$\mathrm{Sem}[\![x_0 \texttt{ or } x_1]\!] \equiv x_0 \vee x_1 \tag{4.6b}$$
$$\mathrm{Sem}[\![x_0 \texttt{ xor } x_1]\!] \equiv (x_0 \vee x_1) \wedge \neg(x_0 \wedge x_1) \tag{4.6c}$$
$$\mathrm{Sem}[\![x_0 \texttt{ implies } x_1]\!] \equiv x_0 \longrightarrow x_1 \tag{4.6d}$$
$$\mathrm{Sem}[\![\texttt{not } x]\!] \equiv \neg\, x \tag{4.6e}$$

Moreover, other non-strict constructs such as

$$\mathrm{Sem}[\![self \texttt{ .oclIsUndefined()}]\!] \equiv \not\delta\, self \tag{4.7}$$

are defined as an exception of the "all operations are strict" rule and are defined as lifting from the definedness predicate def introduced in Section 3.2, without using the strictness combinator.

4.3.4 *Collection Types*

Besides the logical connectives, the constructors for collections are the exceptions to the "all operations are strict" rule of OCL:

> Note that constructors having element values as arguments are deliberately defined not to be strict. A collection value therefore may contain undefined values while still being well-defined.
>
> *(OCL Specification [88], page A-17)*

This behavior results in a non-strict (see Equation 3.17 on page 50) semantics for the collection types. Noteworthy, the normative part of the standard omits any description of constructors of collections and thus both smashed and non-smashed collection variants would be compliant to the normative part of the OCL standard.

We strongly opt for a smashed collection semantics, mainly for three reasons:

1. Smashed collection semantics coincides with the "all operation are strict" principle. Furthermore, a non-smashed collection semantics would lead to unexpected behavior, e. g., expression like `Set{}->union(Set{OclUndefined})` would be undefined, i. e., result in `OclUndefined` and *not* in `Set{OclUndefined}`. Thus, for a non-smashed collection semantics with strict operations even simple laws like $\varnothing \cup X = X$ do not hold.

2. OCL tends to define its constructs towards executability and proximity to object-oriented programming languages such as Java.

3. And most important for our purpose, OCL with non-smashed collection semantics leads to very complicated logical calculi. Just consider the rule

$$\frac{\partial\, self}{\forall\, e \in self\,.\, \partial\, e}$$

   which only holds for a smashed semantics. Without such rules, reasoning over navigations, i. e., collections, always requires a proof of the definedness of all elements of a navigation.

Therefore we use smashing semantics as the default for HOL-OCL. Nevertheless, to study the effects of a non-smashed collection semantics on formal reasoning, we provide a separate configuration of HOL-OCL featuring a non-smashed collection semantics.

Moreover, all collection types of the standard are defined to be finite. Whereas the framework presented in Section 3.3.3 easily allows for the required definitions, we suggest that the future OCL standard deviates from this definition in two points:

1. The constructors for collection should be defined to be strict, i. e., we opt for a smashed collection semantics (see Equation 3.18 on page 51).

2. The type `Set` should support infinite sets.

Our preference for infinite sets is mainly motivated by the fact that this construction allows for treatment of type sets *within* OCL, including the set of all Integers.

Summarizing, we define the semantics of the core operations for collections as follows:

$$\mathrm{Sem}[\![X\texttt{->includes}(x)]\!] \equiv x \in X\,, \tag{4.8a}$$

$$\mathrm{Sem}[\![X\texttt{->complement}()]\!] \equiv X^{-1}\,, \tag{4.8b}$$

$$\mathrm{Sem}[\![X\texttt{->union}(Y)]\!] \equiv X \cup Y\,, \text{and} \tag{4.8c}$$

$$\mathrm{Sem}[\![X\texttt{->intersection}(Y)]\!] \equiv X \cap Y\,. \tag{4.8d}$$

4.3.5 *Equality*

The OCL standard defines equality as the strict equality over values [88, p. A-12] and since objects are values in the standard (see also Section 4.3.1) we choose the strict reference quality $_ \doteq _$ as the default OCL equality, i. e.,

$$\mathrm{Sem}[\![a = b]\!] \equiv a \doteq b \tag{4.9}$$

Nevertheless, we strongly suggest to include the strong (reference) equality, i. e., $_ \triangleq _$, in future version of the standard. We suggest to use "==" as concrete syntax for the strong equality, thus we define

$$\mathrm{Sem}[\![a == b]\!] \equiv a \triangleq b \tag{4.10}$$

Already Cook et al. [37] proposed the inclusion of a strong equality into the OCL standard. In particular in postconditions using the `result` keyword the use of the strong equality, i. e., $result \triangleq \phi$ (where ϕ is an arbitrary OCL expression with the same type as the operation the postcondition belongs to), is useful to describe explicitly that the return value of an operation can be undefined. For example, consider the following operation specification:

```
context C::m(a:Integer):Integer
post: result = 5 div a
```

What is the semantics of this operation given that the precondition does not rule out `a=0`? If the standard strict equality is used this results in an inconsistent specification. If the strong equality is used this operation simple returns undefined when called with an argument of `0`. Depending on the circumstances, both may be reasonable. Thus we suggest to extend OCL with a strong equality operation.

4.3.6 *Encoding User-defined Operations*

The OCL standard, in its present form, forbids overriding of operations. Thus, OCL does only support operation calls, the invocation of operations would be an extension of the standard. We will discuss the operation invocations in more detail in Section 4.5.1.

For defining the semantics of calls to user-defined operation specification, as introduced in Section 3.9, we have first to define the semantics of operation specifications (see Section 3.8 on page 87). The OCL standard defines the semantics of operation specifications as

> The semantics of an operation specification is a set $R \subseteq \text{Env} \times \text{Env}$ defined as
>
> $$[\![\text{context} : C :: op(p_1 : T_1, \ldots, p_n : T_n)\text{pre} : P\text{post} : Q]\!] = R$$
>
> where R is the set of all pre- and post-environment pairs such that the pre-environment τ_{pre} satisfies the precondition P and the pair of both environments satisfies the postcondition Q:
>
> $$R = \left\{ (\tau_{\text{pre}}, \tau_{\text{post}}) \middle| \tau_{\text{pre}} \vDash P \wedge (\tau_{\text{pre}}, \tau_{\text{post}}) \vDash Q \right\}$$
>
> *(OCL Specification [88], page A-33, Definition A.33)*

Therefore, we choose in our framework the *operation semantics with conjunct semantics* (Equation 3.101 on page 88).

Finally, we have to explore, if recursive operations calls should be supported or not. The OCL standard states:

> We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable.
>
> *(OCL Specification [88], page A-27)*

which is sneaking around the underlying problem. Obviously, in a formal proof environment which should be consistent one cannot follow this strategy. Thus we suggest to either limit OCL only to non-recursive calls, or introduce recursion in a way that guarantees termination. We will discuss the latter in more detail in Section 4.5.2.

4.3.7 *Encoding Invariants*

The OCL standard describes the concepts of class invariants informally as:

> When the invariant is associated with a Classifier, the latter is referred to as a "type" in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time.
> *(OCL Specification [88], page 8)*

In our reading, this leads to an identification of the (syntactic) concept of *type* with the (semantic) concept of a *class invariant*. Therefore, we define the semantics of OCL invariants using the concepts of semantic types as introduced in Section 3.5.2.

4.3.8 *Context Declarations*

The OCL standard [88, pp. 157] introduces several classifications of OCL formulae based on the context the formulae is stated. Up to now we have already seen:

- Invariant for classes, denoted by `inv:`, which we encode as described in Section 3.5 and Section 4.3.7.
- Preconditions (`pre:`) and postconditions (`post:`) for operations on which we base our operation specifications upon, see Section 3.8 for details.

Moreover, the standard defines the context declarations, which can be easily converted into invariants, preconditions, and postconditions:

- Initialization (`init`) of attributes, e. g.,

```
context A::x:Integer
  init: 5
```

  which can be directly converted into an invariant for class `A`:

```
context A:
  inv: self.oclIsNew() implies self.x = 5
```

  This formula can be considered as non-standard with regard to the OCL standard, as `oclIsNew()` is syntactically only allowed in postconditions, but it is valid in HOL-OCL.

- The "`body:`" keyword is a shorthand for defining post-conditions that return the result of an evaluation of an OCL expression, e. g.,

```
context A::f():Integer
  body: 5
```

  can be directly converted into

```
context A::f():Integer
  post: result = 5
```

- For invariants restricting the value of an attribute to the result of an evaluation of an OCL expression, the shorthand "`derive`," similar to "`body`" for operations, is provided, e. g.,

```
context A::x:Integer
  derive: 5
```

  can be directly converted into

```
context A
  inv: self.x = 5
```

- For the textual definition of new attributes and operations the context declaration "`def:`" is provided. From our point of view, there is no difference between statements defined graphically in the case tool and those defined using this context declaration.

4.4 A NOTE ON STANDARD COMPLIANCE

When we claim to be compliant to the standard, we do not mean that we converted "literally" the "Semantics" chapter of the OCL standard [88, appendix A] into an Isabelle theory. The deviations from the standard can be grouped into the following six classes:

Making the standard more precise: The most important point here is that the standard uses naïve set theory as basis for the notions type, state, and model. For example, types were explained by some type interpretation function that maps types to a (never described) universe of values and objects. As we use a typed semantic domain, these problems do not occur, e. g., within our formalization only well-typed OCL formula are possible. This is because an ill-typed OCL formulae is also an ill-typed HOL formula in our framework and thus rejected already by the type-checking for HOL. Another example for making the standard more precise is the decision for a smashed collection semantics (see Section 3.2.1).

Presentational issues: This covers our decision to turn OCL into a shallow embedding, as well as our decision to use a combinator-style presentation for the bulk of semantic definitions both for conceptual and technical reasons. In Section 4.4.1, we show why these formulations are equivalent to the ones used in the standard.

Generalizations and Extensions: This covers for example our decision to use an infinite collection type $\texttt{Set}_\tau$, since logical connections between, e. g., `.oclIsTypeOf` and *class invariants* can therefore be satisfactorily treated inside OCL.

Repairing glitches: The standard contains, as can be expected for a large semi-formal document, several errors in local definitions which were revealed during our formalization (see Section 4.4.2 and [21, 28]).

Proofs for Compliance Requirements: The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative "Appendix A" of the standard [88]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see Section 4.4.2).

Providing alternative mathematical syntax: Being the first who did substantial proof work in OCL, we early noticed the need for a compact, mathematical notation for OCL specifications as alternative to the programming-language like notation used in the OCL 2.0 standard. Especially while interactive proving properties, we are favoring the mathematical notion, as it allows for a much more concise presentation. However, we support both syntactical variants as input and output of your system. A table comparing the concrete syntax of the standard and our proposal for a mathematical syntax is given in Appendix A.

In Section 4.4.1, we formally show that our combinator style semantics is equivalent to (a formalized version of) the textbook-style semantics of [88, Appendix A]. Moreover, in Section 4.4.2 we show, that our semantics fulfills the requirements of [88, Chapter 11]. Thus, our formalization also provides in some sense the link between [88, Chapter 11] and [88, Appendix A] missing in the standard: using the detour of our formalization we show (or refute) that the (informal) textbook-style semantics of the standard fulfills the normative requirements of the standard.

4.4.1 *Comparing Textbook-style and Combinator-style Semantics*

In Chapter 3, we use a combinator-style presentation rather than a textbook-style presentation as it is used, for example, in the OCL standard. We use a combinator-style presentation for reasons of conciseness as well as better tool-support. In this section, we show that we can prove formally the equivalence between our semantics and (a formalized version of) the textbook-style semantics of the standard.

Let us consider the definition of *strict* operations over primitive types. For these operations, the standard defines the semantics for them by just

one example. Namely, the semantics of the addition over `Integer` (_ + _). It is defined in the standard [88, page A-11] as follows:

$$I(+)(i_1, i_2) = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \tag{4.11}$$

This definition uses once again the semantic function I. This semantic function I for primitive types and basic operations is integrated in the more general semantic interpretation function for OCL expressions:

Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \mathrm{Expr}_t$ is a function $I[\![e]\!] : \mathrm{Env} \to I(t)$ that is defined as follows.

(i) $I[\![v]\!](\tau) = \beta(v)$.

(ii) $I[\![\mathrm{let}\, v = e_1 \,\mathrm{in}\, e_2]\!](\tau) = I[\![e_2]\!](\sigma, \beta\{v/I[\![e_1]\!](\tau)\})$.

(iii) $I[\![\mathrm{undefined}]\!](\tau) = \bot$ and $I[\![\omega]\!](\tau) = I[\![\omega]\!]$.

(iv) $I[\![w(e_1, \ldots e_n]\!]\tau = I(w)(\tau)\big(I[\![e_1]\!](\tau), \ldots, I[\![e_n]\!](\tau)\big)$.

(OCL Specification [88], page A-26, definition A.30)

Here, τ refers to the *environment* (in the sense of the standard), i. e., a pair consisting of a map assigning variable symbols to values and a pair of system states.

There are two more semantic interpretation functions; one concerned with path expressions (i. e., *attribute and navigation expressions* [88, Definitions A.21], and one concerning the interpretation of preconditions and postconditions $\tau \vDash P$ which is used in two different variants.

To show the equivalence of the two formalization styles, we use our already introduced semantic function. Recall its definition as identity in our shallow embedding:

$$\mathrm{Sem}[\![x]\!] \equiv x \qquad \text{with type } \alpha \Rightarrow \alpha\,. \tag{4.12}$$

$\mathrm{Sem}[\![E]\!]\tau$ can be thought of as the *fusion* of the two semantic functions $I(o)$ and $I[\![E]\!]$ used in the OCL standard.

Now we show for our first strict operation in OCL, the not operator, that it is in fact an instance of the standards definition scheme:

$$\mathrm{Sem}[\![\neg\ X]\!]\gamma = \begin{cases} \lfloor \neg \lceil \mathrm{Sem}[\![X]\!]\gamma \rceil \rfloor & \text{if } \mathrm{Sem}[\![X]\!]\gamma \neq \bot, \\ \bot & \text{otherwise}\,. \end{cases} \tag{4.13}$$

This is formally proven within HOL-OCL. Table 4.2 shows the trivial and canonical proof: it consists of the unfolding of all combinator definitions

```
lemma "(Sem⟦¬ x⟧γ) = (if Sem⟦x⟧γ ≠⊥ then ⌞¬⌜Sem⟦x⟧γ⌝⌟ else ⊥)"
  apply(simp add: OclNot_def DEF_def lifto_def lift1_def lift2_def
                  semfun_def )
done
```

Table 4.2: Proving that ¬ _ is faithful with respect to the standard is trivial and canonical: Unfolding the definitions of all combinator definitions and the semantic function reduces the proof to an application of the simplifier of Isabelle.

```
lemma "(Sem⟦x + y⟧γ) = (if (Sem⟦x⟧γ ≠ ⊥) ∧ (Sem⟦y⟧γ ≠ ⊥)
                           then ⌞⌜Sem⟦x⟧γ⌝ + ⌜Sem⟦x⟧γ⌝⌟
                           else ⊥)"
  apply(simp add: OclNot_def DEF_def lifto_def lift1_def lift2_def
                  semfun_def )
done
```

Table 4.3: Proving that _ + _ is faithful with respect to the standard is trivial and canonical: Unfolding the definitions of all combinator definitions and the semantic function reduces the proof to an application of the simplifier of Isabelle.

(they are just abbreviations of re-occurring patterns in the textbook-style definitions!) and the semantic function Sem which is merely a syntactic marker in our context.

For the binary example of the integer addition, one proceeds analogously and receives as result:

$$\mathrm{Sem}[\![X + Y]\!]\gamma = \begin{cases} \lfloor \ulcorner \mathrm{Sem}[\![X]\!]\gamma \urcorner + \ulcorner \mathrm{Sem}[\![Y]\!]\gamma \urcorner \rfloor & \text{if } \mathrm{Sem}[\![X]\!]\gamma \neq \bot \text{ and} \\ & \mathrm{Sem}[\![Y]\!]\gamma \neq \bot, \\ \bot & \text{otherwise}. \end{cases} \tag{4.14}$$

Table 4.3 shows the quite simple, formal proof of this. Brucker and Wolff [21] present the details for the remaining operations.

In the following, we summarize the differences between the textbook definitions of the OCL standard and our combinator-style approach:

1. The standard [88, chapter A] assumes an "untyped set of values and objects" as semantic universe of discourse. Since we reuse the types from the HOL-library to give Boolean, Integers and Reals a semantics, meta-expressions like $\{\mathrm{true}, \mathrm{false}\} \cup \{\bot\}$ used in the standard are simply illegal in our interpretation. This makes the injections $\lfloor_\rfloor$ and projections $\ulcorner_\urcorner$ necessary.

2. The semantic functions in the standard are split into $I(x)$, $I[\![e]\!]\tau$, $I_{\text{ATT}}[\![e]\!]\tau$ and $\tau \vDash P$. Since we aim at a shallow embedding (which ultimately suppresses the semantic interpretation function), we prefer to fuse all these semantic functions into one.

3. The *environment* τ in the sense of the standard is a pair of a variable map and a state pair. The variable map is superfluous in a shallow embedding (binding is treated by HOL itself), our contexts τ just comprises the pair of pre-state and post-state, thus an implementation of our notion of context.

Overall, we can show, by simple proofs that our semantics is equivalent to the textbook-style semantics of the standard. Nonetheless, we deviate in some points from the standard semantics for our system, HOL-OCL. Most remarkably, we allow infinite sets and require the constructors of the collection types to be strict. In both cases, we could adhere the standard, but this would result in much more complicated proof calculi and would make reasoning over OCL specification much more difficult as it already is.

4.4.2 *Compliance to the Requirements of the OCL Standard*

As already described, the semantics of OCL is spread over several chapters in the OCL standard and in particular, there is no *normative* formal semantics. Many core concepts of OCL are more or less stated implicitly, e. g., while explaining some example. For example, in the following explanation of the role of undefinedness

> In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:
>
> - True OR-ed with anything is True
> - False AND-ed with anything is False
> - False IMPLIES anything is True
> - anything IMPLIES True is True
>
> The rules for OR and AND are valid irrespective of the order of arguments and they are valid whether the value of the other subexpression is known or not. *(OCL Specification [88], page 15)*

we learn two important details of OCL: OCL is based on a Strong Kleene Logic. Thus, most operators of the logical type like _ `and` _ (written _ ∧ _)

are explicitly stated exceptions from the "all operations are strict" principle also stated in this explanation.

Moreover, in the normative part [88, chapter 11], requirements (given as OCL specifications) were formally stated on the standard operations of OCL, e. g., for the implication it is stated:

```
context Boolean::implies(b:Boolean)
  post: (not self) or (self and b)
```

The question, if these requirements are met by the informative semantics description[appendix A] [88], where the semantics of the implication is given by a truth table, is neither investigated nor even mentioned in the standard. Table 4.1 on page 111 shows the semantics of the Boolean operations as given in the formal semantics part of the OCL standard. Summarizing, there are three descriptions of the semantics for the Boolean operations. If we look closer on the implication, we find:

- The statement "anything implies true is true" in the informal description [88, p. 15].
- The postcondition `(not self) or (self and b)` in the normative part [88, p. 139].
- The truth table given in the formal semantics of the standard [88, p. A-21].

If we analyzing this situation, we get:

- The postcondition `(not self) or (self and b)` does not fulfill the informal stated requirement "anything implies true is true" as the postcondition would evaluate to undefined in case self is undefined.
- The formal semantics given as truth table obviously fulfills the informal requirement, and thus does not comply to the requirement given as postcondition.

One can easily deduce that one way of fixing this inconsistency would be to change the requirement to

```
context Boolean::implies(b:Boolean)
  post: (not self) or b
```

which would be semantically equivalent to the truth table given in the formal semantics. One could argue that this is not really an inconsistency in the standard, as the semantics of a postcondition is not clear, if the context object (*self*) is undefined. Following this argumentation, one could argue that the formal semantics is only a refinement of the requirement.

It is a contribution of our work that we can in fact formally prove the requirements are met by our semantics. In the case of the logical connectives, compliance to the standard is proven by deriving lemmas representing the complete truth table as required in the standard. Further, we also prove the normative requirements, and thus connect the informative formal semantics with the normative requirements of the standard.

For example, the requirements of the standard [88, chapter 11] for `isEmpty` over collections is given as follows:

```
context Collection::isEmpty():Boolean
  post: result = (self->size() = 0)
```

which resembles the informal meaning that a collection is empty, if and only if its size is equal to zero. As this requirement is formulated for an abstract class, i. e., a class without implementation, we have to prove this requirement of each subclass. Namely, we have to prove it for the classes `Set`, `Sequence`, `Bag`, and `OrderedSet`. In particular, we will now consider this requirement for the classes `Bag` and `Set`. For bags we formalize this requirements as follows

$$\frac{}{self\,\texttt{->isEmpty()} = \big(self :: (\alpha\ \texttt{Bag}_\tau)\big)\texttt{->size()} \doteq 0}\ . \tag{4.15}$$

Instead of using operation specifications, we prefer their reformulation as algebraic properties that are directly usable in proofs.

Using our formalization based on a smashing semantics for bags, we can easily show that this property holds for our semantics. Table 4.4 shows the corresponding formalization in HOL-OCL, together with its proof. The proof uses a case split (either *self* is defined or not) followed by a simplification where the OCL definitions are unfolded.

For sets, the formalization of the requirement has to take into account that we prefer the use of infinite sets. Sadly, the requirement only holds for finite sets, as for infinite sets the size is undefined. Thus we formalize the requirement for sets as follows:

$$\frac{\vDash \partial\big(self\,\texttt{->size()}\ \big)}{self\,\texttt{->isEmpty()} = \big(self :: (\alpha\ \texttt{Set}_\tau)\big)\texttt{->size()} \doteq 0}\ . \tag{4.16}$$

The constraint $\vDash \partial\big(self\,\texttt{->size()}\ \big)$ (the size of the set must be defined) is a tribute to our extension of the standard to infinite sets; it has the effect to constrain this specification to finite sets, i. e., to the domain the requirement is intended to hold. Once again, we can easily show that this property holds for our semantics. Table 4.5 shows the corresponding formalization in HOL-OCL, together with its proof.

```
lemma "(self ->isEmpty()) = ((self :: (α Bag_τ)) ->size() ≐ 0)"
  apply(rule Bag_sem_cases_ext, simp_all)
  apply(simp_all add: OCL_Bag.OclSize_def OclMtBag_def OclStrictEq_def
                      Zero_ocl_int_def ss_lifting)
  done
```

Table 4.4: Proving that the operation `_->isEmpty()` over bags fulfills the requirements of the OCL standard, i. e., the postcondition `result=(self->size()=0)` holds. We start the proof by applying the rule Bag_sem_cases_ext, followed by a simplification step. Overall, this results in two subgoals (either *self* is defined or not). We can prove both goals by applying the simplifier and unfolding all OCL definitions. The name ss_lifting refers to a simplifier set that unfolds all lifting related definitions, e. g., lift_0.

```
lemma "∂(self ->isEmpty())
       ⟹
       (self ->isEmpty()) = ((self :: (α Set_τ)) ->size() ≐ 0)"
  apply(rule ext)
  apply (drule_tac τ = x in valid_elim)
  apply (frule defSize_implies_finite_Sets)
  apply (auto simp: OclIsEmpty_def OclSize_def OclStrictEq_def
                    Zero_ocl_int_def ss_lifting)
  done
done
```

Table 4.5: Proving that the operation `_->isEmpty()` over sets fulfills the requirements of the OCL standard, i. e., the postcondition `result=(self->size()=0)` holds. We start the proof by exploiting extensionality and the rule valid_elim for converting the global validity in a local one. As the size of the set is defined we also know that the set is defined. Therefore, no case split is needed and we can prove the property directly by applying the automatic tactic `auto` with a simplifier configuration for unfolding the OCL and lifting definitions. The name ss_lifting refers to a simplifier set that unfolds all lifting related definitions, e. g., lift_0.

[21] presents these proofs in more detail, we use them to show that our formalization captures the intention of the OCL standard.

4.4.3 *Faithful Representing UML Object Structures*

For backing our claim that the presented encoding of object structures models faithfully encodes object-oriented data structures, e. g., in the sense of programming languages like Java or C# or the UML standard [90]), we prove a variety of properties. As the UML standard [90] does not present a formal semantics for UML, we cannot give a formal proof that we formalized class models as described in the standard. Nevertheless, we prove for each class properties, like specific type-cast relations that are usually considered as object-oriented. These properties cannot be proven once and for all and thus have to be proven for each user-defined model, e. g., during the encoding of a specific UML model. This is similar to other datatype packages in interactive theorem provers like Isabelle/HOL.

Among many other properties, our datatype package proves that for each pair of classes A and B, related by a generalization (inheritance), where B is a subclass of A it holds that every class is of the kind of its superclass:

$$\frac{\tau \vDash self\ .\texttt{oclIsTypeOf}\ (B)}{\tau \vDash self\ .\texttt{oclIsKindOf}\ (A)}\ . \tag{4.17}$$

Moreover, in that case one can cast the class as well as the more complicated property:

$$\frac{\tau \vDash \partial\, self \qquad \tau \vDash self\ .\texttt{oclIsKindOf}\ (B)}{\tau \vDash self = self\ .\texttt{oclAsType}\ (A)\ .\texttt{oclAsType}\ (B)\ .\texttt{oclAsType}\ (A)}\ . \tag{4.18}$$

As all type-casts are strict operations, one can from Equation 4.18 directly infer the following rule:

$$\frac{\tau \vDash \partial\, self \qquad \tau \vDash self\ .\texttt{oclIsKindOf}\ (B)}{\tau \vDash self\ .\texttt{oclAsType}\ (A)\ .\texttt{oclAsType}\ (B)\ .\texttt{oclIsTypeOf}\ (B)}\ , \tag{4.19}$$

and also

$$\frac{\tau \vDash \partial\, self \qquad \tau \vDash self\ .\texttt{oclIsKindOf}\ (B)}{\tau \vDash \partial(self\ .\texttt{oclAsType}\ (A)\ .\texttt{oclAsType}\ (B)\ .\texttt{oclIsTypeOf}\ (B))}\ . \tag{4.20}$$

Proving these properties is not only needed for showing that our encoding captures the spirit of object-orientation, they are also a prerequisite for a successful reasoning over object structures, i. e., as simplification rules. Moreover, if these properties can be proven the user-defined model already ensures some basic notion of consistency, for example consider a

model where the user defined for class *B* (which is a subclass of *A*) the following invariant:

```
context B:
  inv: not self.oclIsKindOf(A)
```

which is syntactically correct, but make no sense as the invariant contradicts the generalization (inheritance) between class *A* and *B*. In this example, our datatype package will not be able to prove the required properties and will reject the model as being *inconsistent*.

4.5 EXTENDING OCL

In this section, we will propose several extensions to the OCL which increase, in our opinion, the overall usability of OCL. Some of these extensions will introduce new, expressive, constructs into the language. Examples for such extensions are a well-defined semantics for recursive operation calls or the support for specifying frame properties. Others will not change the language itself but allow for more concise specification. An example for such an extension is the introduction of strict Boolean operations.

4.5.1 *Operation Invocation*

The, from object-oriented programming languages, well-known concept of overriding is not yet fully supported by OCL. We believe, this is more or less due to some accidental circumstances:

1. The UML standard [90, chapter 4.4.1] requires that operation names are unique within the same namespace. Albeit, the UML standard allows one to (explicitly) override methods, i. e., implementation of operations.

2. The OCL standard [88, chapter 7.3.41] restricts the use of the precondition and postcondition declarations to operations or other behavioral features. Sadly, all OCL tools we know of do not support the specification of preconditions and postconditions for methods.

3. Whereas the OCL standard speaks on several places of operation calls, it does not give hints how operation overriding should be solved, neither does it explain in detail concepts like operation (method) calls or operation (method) invocations.

Bringing these together, one has to conclude that operation overriding is underspecified, or even not supported in OCL. Nevertheless, we think that overriding inherited operations or methods is a very important feature

of object-orientation and thus should be supported by the OCL. Thus we already provide the theoretical foundations for supporting late-binding (and thus overriding of operations) within HOL-OCL (see Section 3.10 on page 91 and Section 3.11 on page 94 for details), nevertheless a concrete syntax for specifying this has to be worked out. For example, as simple workarounds, one can ignore for operations the well-formedness constraint of UML that requires operation names to be unique within one namespace, or one could introduce new context declarations allowing one to specify preconditions and postconditions for methods.

4.5.2 *Recursive Operations*

The OCL standard requires that recursions should always be terminating to rule out the problems already discussed in Section 3.11 on page 94:

> The right-hand-side of this definition may refer to operations being defined (i. e., the definition may be recursive) as long as the recursion is not infinite. *(OCL Specification [88], page 16)*

and also in the formal semantics chapter the same statement is made:

> For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences [...]. However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable.
>
> *(OCL Specification [88], page A-31)*

We propose to restrict recursive operation calls to well-founded-recursion, based on the semantics presented in Section 3.9 on page 89. Moreover, we propose to extend the concrete syntax of OCL for allowing the direct specification of a measure, which is needed for well-founded recursion. For example, this would allow the following definition of the well-known factorial function (in the context of a class `A`):

```
context A::fac(x:Integer):Integer
pre:  true
post: if x < 0 then 1 else x * f(x-1) endif
measure: m(x, y) = if x < 0 or y < 0
                      then false else x < y endif
```

Overall, this would resolve the obscurities in the standard with respect to recursive calls and moreover this paves the way for supporting formal

reasoning over recursive specifications.

4.5.3 *Explicit Representation of Type Sets and Kind Sets*

As already described in Section 3.5.2 on page 74 we represent types in our framework, and thus also in HOL-OCL, via their characteristic set. As types sets allow for specifying global properties of types in an easy way, we propose to extend the OCL standard in two ways:

- The characteristic set, i. e., the set of all instances, can be infinite (e. g., for the type Integer). Therefore we use an infinite set theory for HOL-OCL. In Section 5.5.3, we discuss the advantages of this setup in more detail.
- For supporting characteristic sets in concrete syntax of OCL we suggest two new operations, which can be described in the style of [88, Chapter 11] as follows:

```
-- Returns all possible instances of self, this may be
-- an infinite set. The Type T is equal to self.
OclType::typeSetOf()::Set(T)
```

and

```
-- Returns all possible instances of self and its
-- subtypes, this may be an infinite set. The Type T is
-- equal to self.
OclType::kindSetOf()::Set(T)
```

In contrast to the operation `allInstances()`, the result of both `typeSetOf()` and `kindSetOf()` does not depend on the system state. Moreover, the kind sets and type sets may be infinite, even for object types.

As we identify types by their type set, in an implementation, such as HOL-OCL the expressions `type::typeSetOf()` and `type::kindSetOf()` can be directly mapped to the corresponding type or kind set during loading of a specification.

For example, these operations allow one for specifying that the addition on Integers is commutative

```
Integer.typeSetOf()->forall(x, y | x + y = y + x)
```

which are, for example, useful for making the requirements of the `sum()` operation more precise. This operation computes the sum of all elements by applying successively the applying an _ + _ operation, which must be (syntactically) defined over the element type of the collection. The

OCL standard [88, page 141] requires informally for this _ + _ operation that is must be associative and commutative. Using our above proposed operations one could express these two properties formally (within OCL) as precondition of the `sum()` operation.

4.5.4 *Strict Boolean Operators*

In addition to the non-strict Boolean connectives which provide a Strong Kleene Logic (SKL), we suggest to provide additionally a strict variant (i. e., following Definition 3.5 on page 55). We believe that both for formal reasoning and also for runtime checking, strict Boolean connectives are very useful in certain situations as they can lead to a concise specification. Thus we suggest to make both variants, i. e., strict and non-strict Boolean connective, available within the same specification, e. g., using the concrete syntax proposed in Table A.1 on page 190:

$$\mathrm{Sem}[\![x_0 \texttt{ sand } x_1]\!] \equiv x_0 \mathbin{\dot{\wedge}} x_1 \tag{4.21a}$$

$$\mathrm{Sem}[\![x_0 \texttt{ sor } x_1]\!] \equiv x_0 \mathbin{\dot{\vee}} x_1 \tag{4.21b}$$

$$\mathrm{Sem}[\![x_0 \texttt{ sxor } x_1]\!] \equiv (x_0 \mathbin{\dot{\vee}} x_1) \mathbin{\dot{\wedge}} \neg(x_0 \mathbin{\dot{\wedge}} x_1) \tag{4.21c}$$

$$\mathrm{Sem}[\![x_0 \texttt{ simplies } x_1]\!] \equiv x_0 \mathbin{\dot{\longrightarrow}} x_1 \tag{4.21d}$$

As the negation is already a strict operation, we do not need to define it again. Moreover, as any strict operation, they simplify the undefinedness reasoning, see Chapter 5 for details.

4.5.5 *Accessing All Instances of the Previous State*

We believe that this is only a minor point, but nevertheless we mention it here: The OCL standard [88, subsection 7.5] restricts the use of `@pre` to properties (attribute, operations, …) of instances of classes. Reading this part of the standard literally, the standard rules out expressions like `allInstances@pre()`, as `allInstances()` is defined as a feature of classes (and not of an instances of a class). Overall, we suggest to explicitly allow `allInstances@pre()`. The semantics for this expression is as obvious as for `allInstances()` itself, i. e.,

$$\mathrm{Sem}[\![t \texttt{::allInstances()}]\!] \equiv \mathrm{allInstances}\ t \tag{4.22a}$$

and

$$\mathrm{Sem}[\![t \texttt{::allInstances@pre()}]\!] \equiv \mathrm{allInstances@pre}\ t\,. \tag{4.22b}$$

For details, see also Section 3.7 on page 86.

4.5.6 *Frame Properties*

We already motivated in Section 3.12 on page 97 that a possibility for specifying the frame property, i. e., all "things" that do not change during a system transition is very important. Thus we propose to extend OCL with means for specifying the frame property of an operation, in particular we propose to introduce an operation `->modifiedOnly()` whose semantic is defined by the operator modifiedOnly (see Equation 3.125 on page 98):

$$\text{Sem}[\![\, \mathit{self}\ \texttt{->modifiedOnly()}\,]\!] \equiv \text{modifiedOnly}\ \mathit{self}\ . \tag{4.23}$$

4.6 DISCUSSION

In this chapter, we presented a formal, machine-checked semantics for OCL in the context of UML class models. The semantics we presented is based on the formal framework we presented in Chapter 3. Further, we can guarantee that our semantics is consistent, under the assumption that Isabelle/HOL itself is consistent.

In contrast to the formal semantics of the standard, i. e., [88, Appendix A], we formally define the semantics for all operations and not only define the semantics "by example." [21] contains the Isabelle theory files of our complete formalization.

Additionally, we showed that our semantics is equivalent to a formalization of [88, Appendix A] and also that our semantics fulfills the requirement of [88, Chapter 11]. Thus, we indirectly showed that [88, Appendix A] also fulfills most of the requirements of [88, Chapter 11].

Our semantics deviates from the semantics OCL standard in two points: First we propose a smashed semantics for collections and second, we prefer infinite sets over finite ones. The decision for a smashed semantics is only a deviation from the informal semantics given in [88, Appendix A]; as the normative part of the OCL standard omits a specification of the constructors for collections, a smashed semantics is still compliant to the normative part of the OCL standard [88]. Moreover, this decision is also backed up by more recent version of the standard [89]. Extending the OCL standard with support for infinite sets requires minor modifications on several places in the standard, especially [88, Chapter 11]; for example, the precondition of operations converting sets into sequences need to be extended by a constraint requiring the finiteness of the set, see [21] for details.

Moreover, our semantics is the basis for a formal tool we develop, called HOL-OCL. As HOL-OCL is built on top of Isabelle/HOL in general and in particular on top of the semantics we presented in this chapter, we can ensure that HOL-OCL implements exactly our formalized semantics. Overall, this is, together with the consistency guarantee, one great advantage of

building formal tools on top of well-known and reliable generic theorem provers, e. g., Isabelle/HOL.

In the next chapter, we will present several proof calculi for our formalization of OCL. These calculi built the basis for the (automatic) proof support HOL-OCL provides.

CALCULI FOR OBJECT-ORIENTED SPECIFICATIONS

5

In this chapter, we develop several deduction systems for object-oriented constraint languages in general and OCL, as defined in Chapter 4, in particular. We define two equational calculi (UEC and LEC) usable for interactive proofs or proofs by hand, and a tableaux calculus (LTC) geared towards automatic reasoning. All rules we present are *derived* within Isabelle/HOL from the semantic definitions presented in Chapter 3. Therefore we can guarantee the logical soundness, with respect to the core logic HOL, of all these rules.

5.1 CHALLENGES

Having a conservative embedding for an object-oriented constraint language $\mathcal{L}$, e. g., OCL, as a shallow embedding into HOL, one might ask why the development of specific calculi is necessary. Of course, one can always unfold the definitions and thus converting, an $\mathcal{L}$ formula into a HOL expression and try to prove the latter. However, we dismiss this idea, mainly, due to the following two reasons:

- The semantics of HOL and the concepts of object-orientation are not closely related, thus the encoding is quite complex. Furthermore, object-oriented constraint languages are usually very rich languages, i. e., also providing a theory of its own datatypes. Therefore, unfolding of all definitions to the HOL level leads to a tremendous blow-up in the size of the formulae. The resulting HOL formulae are not efficient to analyze, neither for automatic proof tactics nor interactively: they are just too big.
- Unfolding all definitions of our framework and thus doing the reasoning over a pure HOL specification broadens the gap between the original specification (e. g., given as UML models annotated with OCL constraints) and the representation (pure HOL formulae) over which the verification is done. Our experience with larger case-studies [25], carried out in using a similar proof-environment

> for Z [26], shows that around half of the flaws found during formal verification are fixed by correcting the specification and half of them are caused by misstating the verification goals. Thus, for every flaw found one has to decide if it is caused by a failure in the specification or not. This decision is much easier if the formal analysis is carried out in the same language the specification is written in.

Therefore, we are strongly in favor for supporting proofs on the level of the object-oriented specification. This strategy does not only avoid the blow-up in the representation of the specification, it also provides effective means to communicate with the designers and domain experts of the original specifications.

Developing proof calculi that allow for a good integration into the generic proof tactics of Isabelle is a challenge in itself. Moreover developing them for a *object-oriented constraint language* $\mathcal{L}$ over object structures such as OCL (over UML object models), Spec# (over C# models), or JML (over Java models) provides several additional difficulties:

- The logic itself is three-valued as it has to support means of exception handling. Moreover a strict evaluation model (similar to programming languages) is usually the default. In contrast, the meta-logic and the proof support provided by Isabelle are optimized for two-valued reasoning.
- We do not only have the logic but also a rich datatype theory with collection types which influence the overall flavor of the language. As consequence, a good combination of specialized subcalculi for all these types is a prerequisite for supporting automated reasoning in practically relevant reasoning work.

Proof calculi for formal languages with this flavor are rarely explored; especially for OCL no proof calculi supporting a Strong Kleene Logic are published. Moreover, the OCL standard does not provide any calculi and furthermore also neither a proof method nor a formal methodology. We therefore develop a formal methodology by ourselves, based on previous work [51, 63] for three valued logic and based on own experiences with the development of an interactive proof environment [26]. We extend this work by providing support object-oriented data models and developing a tool-supported methodology. The focus of our discussion of calculi in this section is mostly on the subset of our framework described in Chapter 4; this will result in an interactive theorem prover environment for UML/OCL called HOL-OCL. An overview of the architecture of this system is given in Section 6.1.1, the technical aspects of this system are described elsewhere [21].

In Section 5.2, we discuss the validity of formulae and derive basic properties of validity statements. Based on this discussion, we will in Section 5.3 introduce several equality and congruence relations over formulae which will serve as the basis for the calculi we develop. In Section 5.4 we present two subcalculi optimized for reasoning over two distinct properties of $\mathcal{L}$ formulae: undefinedness and context passingness (see Section 5.4.1). Moreover, we will present a setup for arithmetic computations and calculi for converting a fragment of object-oriented specifications into HOL. On this basis, we will describe the logic of our framework in Section 5.5 and develop proof calculi in Section 5.6. We conclude this chapter with some remarks on the development of automated proof support for the presented calculi and a general discussion how the presented calculi are related to OCL (Section 5.7).

5.2 VALIDITY AND JUDGMENTS

In this section, we will introduce a notion of validity of $\mathcal{L}$ formulae. On this basis, we will also discuss a first characterization of valid formulae.

5.2.1 *Validity of Formulae*

Recall that formulae of our constraint language $\mathcal{L}$ depend on a context (see Section 3.2.1 for details), thus we can define the validity of a formula with respect to a concrete context τ, or with respect to all contexts. Therefore, we introduce the notion of *local validity* and, moreover, we generalize validity to judgments of the form:

local validity

$$\big(\tau \vDash_{\mathtt{t}} P\big) \equiv \big(P(\tau) = {}_{\llcorner}\mathtt{true}_{\lrcorner}\big), \tag{5.1a}$$

$$\big(\tau \vDash_{\mathtt{f}} P\big) \equiv \big(P(\tau) = {}_{\llcorner}\mathtt{false}_{\lrcorner}\big), \text{ and} \tag{5.1b}$$

$$\big(\tau \vDash_{\bot} P\big) \equiv \big(P(\tau) = {}_{\llcorner}\bot_{\lrcorner}\big). \tag{5.1c}$$

As a shorthand for all three variants, we will write $\tau \vDash_x P$ for $x \in \{\bot, \mathtt{f}, \mathtt{t}\}$. We will write $\tau \vDash P$ for $\tau \vDash_{\mathtt{t}} P$ and use this as our default notion of validity.

Further, we generalize local validity judgments to a notion of *global validity* judgments, sometimes also called universal validity judgments:

global validity

$$\big(\vDash_{\mathtt{t}} P\big) \equiv \big(\forall\, \tau.\ \tau \vDash_{\mathtt{t}} P\big), \tag{5.2a}$$

$$\big(\vDash_{\mathtt{f}} P\big) \equiv \big(\forall\, \tau.\ \tau \vDash_{\mathtt{f}} P\big), \text{ and} \tag{5.2b}$$

$$\big(\vDash_{\bot} P\big) \equiv \big(\forall\, \tau.\ \tau \vDash_{\bot} P\big). \tag{5.2c}$$

In analogy to local validity, we will write $\vDash_x P$ for $x \in \{\bot, \mathtt{f}, \mathtt{t}\}$ as a shorthand for all three variants and we will write $\vDash P$ for $\vDash_{\mathtt{t}} P$. Overall, global validity captures the fact, that a formula is valid for all contexts.

Naturally, we can build both a local congruence, and a global congruence based on our validity notion. Moreover, the global and local congruences are related via the following three theorems:

$$\frac{\bigwedge \tau.\ (\tau \vDash X) = (\tau \vDash Y) \quad \bigwedge \tau.\ (\tau \vDash_{\mathsf{f}} X) = (\tau \vDash_{\mathsf{f}} Y)}{X = Y}\,, \tag{5.3a}$$

$$\frac{\bigwedge \tau.\ (\tau \vDash X) = (\tau \vDash Y) \quad \bigwedge \tau.\ (\tau \vDash_{\bot} X) = (\tau \vDash_{\bot} Y)}{X = Y}\,, \text{ and} \tag{5.3b}$$

$$\frac{\bigwedge \tau.\ (\tau \vDash_{\mathsf{f}} X) = (\tau \vDash_{\mathsf{f}} Y) \quad \bigwedge \tau.\ (\tau \vDash_{\bot} X) = (\tau \vDash_{\bot} Y)}{X = Y}\,. \tag{5.3c}$$

Since a validity statement like $\tau \vDash X$ has the type bool in HOL, all equalities in the premises of these rules can be seen as logical equivalences

$$\tau \vDash X \Leftrightarrow \tau \vDash Y\,. \tag{5.4}$$

As such, they can be decomposed into implications from left to right and vice versa.

In principle, reasoning over formulae of our constraint language can either be based on a decomposition strategy of judgments or on exploiting equivalences between formulae or judgments over them. In the latter case, the transport of knowledge of contexts is a major technical issue in reasoning over such formulae which turns out to be even more important (i. e., more fundamental) than reasoning over definedness of subterms. We will treat both reasoning over contexts and definedness in separate subcalculi (see Section 5.4.1 and Section 5.4.2). Overall, there is a notable similarity to labeled deduction systems [43, 112].

5.2.2 *A Theory of Basic Judgment*

Following the definitions introduced in the previous section, we can easily check the following link between judgments and equalities:

$$\big(\tau \vDash A\big) = \big(A\ \tau = {\llcorner}\mathsf{true}{\lrcorner}\big) = \big(A\ \tau = \mathsf{t}\ \tau\big)\,. \tag{5.5}$$

Moreover, the following analogous equations reveal that only one kind of judgments is needed. As canonical form we take the validity judgment, i. e., for the global validity:

$$(X = \bot) = (\vDash \not\partial\, X)\,, \tag{5.6a}$$

$$(X = \mathsf{f}) = (\vDash \neg\, X)\,, \text{ and} \tag{5.6b}$$

$$(X = \mathsf{t}) = (\vDash X)\,; \tag{5.6c}$$

and for the local validity:

$$(X\ \tau = \bot\ \tau) = (\tau \vDash \not\emptyset\, X)\,, \tag{5.6d}$$
$$(X\ \tau = \mathtt{f}\ \tau) = (\tau \vDash \neg X)\,, \text{and} \tag{5.6e}$$
$$(X\ \tau = \mathtt{t}\ \tau) = (\tau \vDash X)\,. \tag{5.6f}$$

Applied from right to left, these theorems reveal also the character of judgments as rewrite-rules that can be used by automatic rewriting procedures. From these equalities, the base cases for judgments follow directly, i. e., for the global validity:

$$\neg(\vDash \bot)\,, \tag{5.7a}$$
$$\neg(\vDash \mathtt{f})\,, \tag{5.7b}$$
$$\vDash \mathtt{t}\,; \tag{5.7c}$$

and for the local validity:

$$\neg(\tau \vDash \bot)\,, \tag{5.7d}$$
$$\neg(\tau \vDash \mathtt{f})\,, \tag{5.7e}$$
$$\tau \vDash \mathtt{t}\,. \tag{5.7f}$$

An important property of judgments is related to the three-valuedness of $\mathcal{L}$, i. e., *quadrium non datur*:

quadrium non datur

$$\frac{}{(\tau \vDash A) \vee (\tau \vDash \neg A) \vee (\tau \vDash \not\emptyset\, A)}\,. \tag{5.8}$$

With this rule, a defined formula can be converted into formulae which are true or which are false; this gives rise for six corresponding case-split lemmas.

5.3 EQUIVALENCES AND CONGRUENCES

In this section, we refine the notion of congruence and equivalence; in particular we will introduce several different equivalence relations for formulae of our constraint language. These equivalences will be one building block of the calculi we will present in Section 5.6.

5.3.1 *Basic Equivalences and Congruences*

We distinguish four equivalences over formulae of our constraint language:

UC: *Universal (Formula) Congruence (UC).* This equivalence is a congruence. It requires that two formulae A and B, both of type $\mathtt{Boolean}_\tau$, agree in all contexts τ and in all three truth values of type $\mathtt{Boolean}_\tau$,

Universal (Formula) Congruence (UC)

i. e., they are equal with respect to the meta (HOL) equality. They have the form

$$\frac{}{A = B} \quad \text{or} \quad \frac{A_1 = B_1 \quad \cdots \quad A_n = B_n}{A_{n+1} = B_{n+1}} \,. \tag{5.9}$$

Local (Formula) Equivalence (LE)

LE: *Local (Formula) Equivalence (LE).* This equivalence requires that two formulae agree on all three truth values of $\texttt{Boolean}_\tau$ in a specific context τ. They have the form

$$\frac{}{A\ \tau = B\ \tau} \quad \text{or} \quad \frac{H_1 \quad \cdots \quad H_n}{A_{n+1}\ \tau = B_{n+1}\ \tau} \,. \tag{5.10}$$

The premises H_i can have the form $A\ \tau = B\ \tau$ or instances of this scheme such as $\tau \vDash_x A$.

Universal Judgement Equivalence (UJE)

UJE: *Universal Judgement Equivalence (UJE).* This equivalence requires that two formulae for all contexts τ agree on one value x from $\texttt{Boolean}_\tau$. They have the form $\vDash_x A = \vDash_x B$ or Horn-clauses over them.

Local Judgement Equivalence (LJE)

LJE: *Local Judgement Equivalence (LJE).* This equivalence requires that two formulae agree on a specific truth value x of type $\texttt{Boolean}_\tau$ in a specific context τ, i. e., $\tau \vDash_x A = \tau \vDash_x B$ or horn-clauses over them.

All three possible kinds of judgments, namely $\vDash A$ (*universal validity*), $\vDash_{\mathsf{f}} A$ (*universal invalidity*), and $\vDash_{\bot} A$ (*universal undefinedness*), can be converted into each other. Thus, we can choose just one of them as representative; the same holds for the local counterparts (e. g., the local validity $\tau \vDash A$)). In both cases, we choose the validity as representative judgment.

Moreover, the UJE-format is only of notational interest: it is not possible to build a complete calculus using only UJE-rules. For example, consider the valid rule

$$\frac{\vDash \partial A \qquad \vDash \partial B}{(\vDash A \wedge B) = \big((\vDash A) \wedge (\vDash B)\big)} \,. \tag{5.11}$$

This rule holds due to distribution of universal quantification over $_ \wedge _$. An analogue version for the disjunction of our object-oriented constraint language $\mathcal{L}$ does not hold, i. e., only the following variant holds:

$$\frac{\vDash \partial A \qquad \vDash \partial B}{(\vDash A \vee B) = \big(\forall \tau.\ (\tau \vDash A) \vee (\tau \vDash B)\big)} \,. \tag{5.12}$$

The LE-format, however, is flexible enough to build complete calculi. For example, consider

$$\frac{\tau \vDash \partial A \quad \tau \vDash \partial B}{(\tau \vDash A \wedge B) = \big((\tau \vDash A) \wedge (\tau \vDash B)\big)} \tag{5.13}$$

as a propositional equivalence or

$$\begin{aligned}(\tau \vDash \forall\, x \in S.\ A \wedge B) = \big(&\forall\, x.\ (\tau \vDash x \in S) \longrightarrow (\tau \vDash A)\big) \\ &\wedge \big(\forall\, x.\ (\tau \vDash x \in S) \longrightarrow (\tau \vDash B)\big)\end{aligned} \tag{5.14}$$

as an equivalence on predicates.

Judgments are propositional and formulae in the form of LJE can be decomposed into implications from left-to-right and from right-to-left. Thus, there is another line to automated reasoning over $\mathcal{L}$-formulae: they can be turned into a tableau calculus (LTC, see Section 5.6.3).

Unfortunately, there is a trade-off between completeness of the various calculi based on these equivalences and deductive efficiency. UC is the only congruence that can be directly processed by Isabelle's simplifier; normalizations in UC can be computed relatively efficiently. While UC comprises several thousands of rules (among them, the strictness and computational rules of operators) it does not form a complete calculus for several reasons: Some properties in $\mathcal{L}$ are inherently context dependent, in particular when referring to paths. Others are difficult to formalize as a universal congruence. On the other end of the spectrum, since local judgments, e. g., $\tau \vDash A$, are simply propositions, they are extremely flexible. When extending LJE rules to equivalences over propositional (predicative) formulae, it is not difficult at all to convert them into a fairly abstract but still provably complete calculus with safe rules. Safe rules are rules whose application yield an equivalent transformation of the proof state. Their use is "safe" in the sense that no logical content is lost.

With respect to LTC, it is well-known that the complexity for tableaux-based reasoning in Strong Kleene Logic is higher than tableaux-based reasoning for two-valued logics [49]. However, the logic is only one little fragment of the overall problem of building decision procedures for fragments of our constraint language: Most operations are strict, and from the data-invariants, definedness of many literals can be inferred, such that large fragments of the language are in fact two-valued. Furthermore, we do not only obtain the logic but also a rich datatype theory with collection types which give the overall language a flavor in its own. As consequence, a good combination of all these types of calculi is a prerequisite for developing procedures for automated reasoning that are applicable in realistic case-studies.

5.3.2 *On the Relationship Between the Different Forms of Equivalence*

In this section, we will explore briefly how the different forms of equivalence (i. e., UC, LE, LJE, and LTC) relate to each other. In particular, the following characterizations between statements in these forms of equivalence hold:

- Universal (Formula) Congruence can be converted into Local (Formula) Equivalence as follows:

$$(A = B) = (\forall\, \tau.\ A\ \tau = B\ \tau)\,. \tag{5.15}$$

- Universal (Formula) Congruence can be expressed in Local Judgement Equivalence by the following two equations:

$$\begin{aligned}(A = B) = \Big(&\big(\forall\, \tau.\ (\tau \vDash A) = (\tau \vDash B)\big)\\ &\wedge \big(\forall\, \tau.\ (\tau \vDash \not\phi\, A) = (\tau \vDash \not\phi\, B)\big)\Big)\end{aligned} \tag{5.16a}$$

and

$$\begin{aligned}(A = B) = \Big(&\big(\forall\, \tau.\ (\tau \vDash A) = (\tau \vDash B)\big)\\ &\wedge \big(\forall\, \tau.\ (\tau \vDash \neg A) = (\tau \vDash \neg B)\big)\Big)\,.\end{aligned} \tag{5.16b}$$

These two equations implicitly exploit the "quadrium non datur" (Equation 5.8): If two formulae agree in two truth-values, they have also to agree on the third.

- Local (Formula) Equivalence can be expressed in Local Judgement Equivalence by the following two equations:

$$\begin{aligned}(A\ \tau = B\ \tau) = \Big(&\big((\tau \vDash A) = (\tau \vDash B)\big)\\ &\wedge \big((\tau \vDash \not\phi\, A) = (\tau \vDash \not\phi\, B)\big)\Big)\end{aligned} \tag{5.17a}$$

and

$$\begin{aligned}(A\ \tau = B\ \tau) = \Big(&\big((\tau \vDash A) = (\tau \vDash B)\big)\\ &\wedge \big((\tau \vDash \neg A) = (\tau \vDash \neg B)\big)\Big)\,.\end{aligned} \tag{5.17b}$$

They also also implicitly exploit the "quadrium non datur" (Equation 5.8).

- The relation of Local Judgement Equivalence to equivalence is described by:

$$\begin{aligned}\big((\tau \vDash A) = (\tau \vDash B)\big) = \Big(&(\tau \vDash A \longrightarrow \tau \vDash B)\\ &\wedge (\tau \vDash B \longrightarrow \tau \vDash A)\Big)\,.\end{aligned} \tag{5.18}$$

This characterization justifies an own tableaux calculus on the basis of local judgments. This rule is the starting point for the development of the Local Tableaux Calculus (LTC) (see Section 5.6.3).

- The connection of strong equality to Local (Formula) Equivalence is described by:

$$\big(\tau \vDash (a \triangleq b)\big) = (a\ \tau = b\ \tau)\,. \tag{5.19}$$

5.4 SUBCALCULI

In this section, we introduce our first calculi for our framework. These calculi are specialized to particular tasks, for example handling undefinedness or reasoning over context-passingness.

5.4.1 *Reasoning about Context-Passingness*

In the following, we discuss the first subcalculus, the reasoning over *context-passingness*. Revising the definition (see Equation 3.15 on page 50) *context-passingness*

$$\mathrm{cp}(P) \equiv \big(\exists f.\ \forall X\tau.\ P\ X\ \tau = f(X\ \tau)\tau\big) \tag{5.20}$$

that appears in Section 3.2.1, one might wonder why this definition is so important for reasoning in $\mathcal{L}$. An answer can be drawn from the following rule:

$$\frac{A\ \tau = B\ \tau \qquad \mathrm{cp}\ P}{P\ A\ \tau = P\ B\ \tau}\,. \tag{5.21}$$

In other words, any Local (Formula) Equivalence $A\ \tau = B\ \tau$ is in fact a congruence for all terms $P\ X$ that are context passing. As a consequence, being context passing ($\mathrm{cp}\ _$) is a pre-requisite for replacing a term by another in some (context-passing) term $P\ X$. Thus P can also be interpreted as the "surrounding term" marked by the "position" X. Since global equivalence is semantically closely connected to strong equality, this means that all sorts of term-rewriting in our constraint language, e. g., OCL, will be constrained to "adequate surrounding terms," i. e., terms that are context passing. Context-passingness is a tribute to the fact that the typing of our logic depends on the context; it can be seen as an invariant of semantic functions representing the operations.

The inference rules for establishing context-passingness are contained in Table 5.1 and follow an inductive scheme over the structure of the expressions of the constraint language: The base-cases (Equation 5.22a and Equation 5.22b) are straight-forward, i. e., constant expressions or identities are context passing. The lemmas presented in Equation 5.22c, Equation 5.22d, and Equation 5.22e contain the step-cases and work for all operators that had been defined via the context lifting combinators.

$$\frac{}{\mathrm{cp}(\lambda\, X.\ c)} \tag{5.22a}$$

$$\frac{}{\mathrm{cp}(\lambda\, X.\ X)} \tag{5.22b}$$

$$\frac{}{\mathrm{cp}(\mathrm{lift}_0\ c)} \tag{5.22c}$$

$$\frac{\mathrm{cp}\ P}{\mathrm{cp}(\lambda\, X.\ \mathrm{lift}_1\ f(P X))} \tag{5.22d}$$

$$\frac{\mathrm{cp}\ P \qquad \mathrm{cp}\ P'}{\mathrm{cp}\big(\lambda\, X.\ \mathrm{lift}_2\ f(P\ X)(P'\ X)\big)} \tag{5.22e}$$

Table 5.1: The core rules of the subcalculus for context passingness.

We presented all operations of our framework in combinator-style, therefore, with these combinators, these generic step-cases pave the way for the automatic generation of one cp-rule with a uniquely defined pattern for each operator. Thus, for all expressions built entirely from operators (which is for example the case for all formulae of the original system specification) of $\mathcal{L}$, the derivation of $\mathrm{cp}\ P$ formulae are done fully automatic by backward chaining (using both the Isabelle simplifier as well as the Isabelle classical reasoner).

5.4.2 *Reasoning about Undefinedness and Definedness*

Definedness and undefinedness are indeed opposite concepts, i. e., they satisfy the rule of the excluded middle for all X in all types:

$$\tau \vDash \not\partial\, X \lor \tau \vDash \partial\, X\,. \tag{5.23}$$

This gives, of course, rise to case-split techniques that can be applied automatically in calculi based on Universal Judgement Equivalence or Local Judgement Equivalence.

However, we focus on strict operations as, for example, it is the case for OCL. For strict operations, the use of undefinedness in deduction is easier than its counterpart.

Undefinedness can be propagated throughout a proof state via forward reasoning and exploited via rewriting. The forward reasoning part is

$$\partial\,\mathtt{f} = \mathtt{t} \qquad \partial\,\mathtt{t} = \mathtt{t} \qquad \partial\,\bot = \mathtt{f} \tag{5.24}$$

$$\partial\,\partial\,X = \mathtt{t} \qquad \partial(\neg\,X) = \partial\,X \tag{5.25}$$

$$\partial(X \wedge \partial\,X) = \mathtt{t} \qquad \partial(\neg\,X \wedge \partial\,X) = \mathtt{t} \tag{5.26}$$

$$\partial(\partial\,X \wedge X) = \mathtt{t} \qquad \partial(\partial\,X \wedge \neg\,X) = \mathtt{t} \tag{5.27}$$

$$\partial(X \triangleq Y) = \mathtt{t} \qquad \partial(X \doteq Y) = \partial\,X \wedge \partial\,Y \tag{5.28}$$

$$\partial\,f\,X = \partial\,X \qquad \partial\,f\,X\,Y = \partial\,X \wedge \partial\,Y \tag{5.29}$$

(a) The core definedness rules hold for all strict operations f.

$$\partial(\mathtt{if}\,X\,\mathtt{then}\,Y\,\mathtt{else}\,Z\,\mathtt{endif}) = \partial\,X \wedge (X \wedge \partial\,Y \vee \neg\,X \wedge \partial\,Z) \tag{5.30}$$

$$\partial(X \wedge Y) = (\partial\,X \wedge \partial\,Y) \vee \neg\,X \vee \neg\,Y \tag{5.31}$$

$$\partial(X \vee Y) = (\partial\,X \wedge \partial\,Y) \vee X \vee Y \tag{5.32}$$

$$\partial(X \longrightarrow Y) = (\partial\,X \wedge \partial\,Y) \vee \neg\,X \vee Y \tag{5.33}$$

(b) Strong definedness rules.

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y \quad \tau \vDash \partial\,Z}{\tau \vDash \partial(\mathtt{if}\,X\,\mathtt{then}\,Y\,\mathtt{else}\,Z\,\mathtt{endif})} \tag{5.34}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \longrightarrow Y)} \tag{5.35}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \vee Y)} \tag{5.36}$$

$$\frac{\tau \vDash \partial\,X \quad \tau \vDash \partial\,Y}{\tau \vDash \partial(X \wedge Y)} \tag{5.37}$$

$$\frac{\mathrm{cp}\,P}{\begin{aligned}(\tau \vDash \partial(\forall\,x \in S.\,Px)) = (\tau \vDash \partial\,S \wedge ((\exists\,x.\,\tau \vDash x \in S \wedge \neg\tau \vDash Px)\\ \vee\,(\forall\,x.\,\tau \vDash \partial\,x \wedge \tau \vDash x \in S \longrightarrow \tau \vDash \partial\,Px)))\end{aligned}} \tag{5.38}$$

(c) Weak definedness rules.

Table 5.2: The definedness calculi.

covered by rules like

$$\frac{\tau \vDash \not\partial\, X \quad \mathrm{cp}\, P}{P\, X\, \tau = P\, \bot\, \tau} \tag{5.39}$$

and several variants used for technical purposes. After replacing a term X by $\bot$, strictness rules like $f\ \bot\ Y$ or $f\ Y\ \bot$ can reduce the size of subgoals drastically.

We now focus on the far more involved treatment of definedness. The core of reasoning over definedness is in fact representable in a Universal (Formula) Congruence calculus. It is summarized in Table 5.2a. This rule set contains also a class of rules for "strict standard operations f." With this set of operations, we refer to operations that had been defined by constant definitions of the form:

$$f \equiv \mathrm{lift}_2(\mathrm{strictify}(\lambda\, x.\ \mathrm{strictify}(\lambda\, y.\ {}_\llcorner g\ \ulcorner x \urcorner\ \ulcorner y \urcorner_\lrcorner))). \tag{5.40}$$

As in the case of the generation of cp-inference rules, we exploit the combinator-style definitions of the standard operators here and generate this type of rules in pre-computation steps once and for all.

This strong definedness calculus can be extended to quantifiers as follows:

$$\frac{\mathrm{cp}\, P}{\partial(\forall\, x \in S.\ Px) = \partial\, S \wedge \big((\exists\, x \in S.\ (\neg\, Px)) \vee (\forall\, x \in S.\ \partial\, Px)\big)} \tag{5.41a}$$

and

$$\frac{\mathrm{cp}\, P}{\partial(\exists\, x \in S.\ Px) = \partial\, S \wedge \big((\forall\, x \in S.\ (\neg\, Px)) \vee (\exists\, x \in S.\ \partial\, Px)\big)}\ . \tag{5.41b}$$

The power, but also a major drawback of this type of calculus based on Universal (Formula) Congruence stems from the rules listed in Table 5.2b. They result in the generation of numerous case splits, which are often unnecessary: if we know that all variables in a subgoal are defined (and this is an important special case that we can achieve by initial case-splits done once and for all), simple conditional rules leading to direct backward-chaining are sufficient.

To overcome this drawback, we derived the following alternative rule-set listed in Table 5.2c. It reduces the burden of applicability to the question, if the definedness of a term can be derived. By the way, the $(\tau \vDash \partial\, x)$-part in the last rule is strictly speaking redundant (as we will see when we discuss the Set-theory in more detail), but facilitates the establishment of $\tau \vDash \partial\, Px$ since this additional assumption will be used if x occurs in (the instance of) $P\, x$.

5.4.3 *Arithmetic Computational Rules*

An important source of deduction is computation. Computation is needed when $\tau \models 3 + 5 \doteq 4$ is refuted. So far, we have only used declarative concepts to introduce numbers; the question arises how this can be used for computation and deduction.

This problem is by no means new and deeply intertwined with the existing solution in Isabelle/HOL. In the HOL library, a type bin for binary two's complement representation has been introduced by classical, conservative means. The Isabelle parser is configured to parse a literal like 3 to bitstring representation using two's complement, i. e., $(101)_2$. Further, an axiomatic class num is defined providing a function declaration numberOf :: bin ⇒ α :: num that can be overloaded for each type declared to be an instance of class num. Thus, for new datatypes, just a new function is defined that converts a bitstring representation to this new type. In the library, such a conversion has been provided, for example, for int. Based on these definitions, suitable rules have been derived that perform the integer operations like addition on the two's complement representation directly; these rules can be directly processed by the simplifier.

With respect to the types $\texttt{Integer}_\tau$, $\texttt{Real}_\tau$ and $\texttt{String}_\tau$ we can proceed analogously. For example, after declaring $\texttt{Integer}_\tau$ to be an instance of num, we provide the following definition numberOf for the representation conversion:

$$(\mathrm{numberOf} :: \mathrm{bin} \Rightarrow \texttt{Integer}_\tau)\ b \equiv \mathrm{lift}_0(\llcorner(\mathrm{numberOf} :: \mathrm{bin} \Rightarrow \mathrm{int})\, b \lrcorner)\,. \tag{5.42}$$

Thus, the "new" numberOf with type bin ⇒ $\texttt{Integer}_\tau$ is the context-lifted, $\bot$-lifted version of the "old" numberOf on integers (HOL). From this definition, among others, the following rules can be derived::

$$\partial(\mathrm{numberOf}\, a) = \texttt{t}\,, \tag{5.43a}$$

$$(\mathrm{numberOf}\, a) + (\mathrm{numberOf}\, b) = \mathrm{numberOf}(a +_2 b)\,, \tag{5.43b}$$

$$(\mathrm{numberOf}\, a) \cdot (\mathrm{numberOf}\, b) = \mathrm{numberOf}(a \cdot_2 b)\,, \text{and} \tag{5.43c}$$

$$\frac{\neg\, \mathrm{iszero}(\mathrm{numberOf}(a -_2 b))}{((\mathrm{numberOf}\, a) \triangleq (\mathrm{numberOf}\, b)) = \texttt{f}}\,. \tag{5.43d}$$

Thus, besides definedness-related computations ("all values are defined"), computations in $\mathcal{L}$ were mapped directly to computations in the underlying meta-logic HOL. This setup enables the standard simplifier of Isabelle to handle arithmetic on the level of $\mathcal{L}$ automatically. For example, the simplifier refutes judgments like

$$\tau \models 3 + 2 \doteq 7 \tag{5.44}$$

fully automatically, i. e., without user interaction. In the following, we give a short sample proof:

```
lemma "⟦τ ⊨ 3 ≐ C; τ ⊨ C + 2 ≐ (7 :: Integer_τ)⟧
       ⟹
       τ ⊨ A ∧ B"
  apply(ocl_hypsubst, simp)
done
```

The first proof method `ocl_hypsubst` canonizes the proof state, i. e., replaces the variables with their values. The intermediate proof state looks as follows:

$$⟦τ ⊨ 3 + 2 ≐ (7 :: \texttt{Integer}_τ); ⟧ ⟹ τ ⊨ A ∧ B \tag{5.45}$$

The standard simplifier (configured to use the derived rules above automatically) computes the addition, refutes the judgment and thus proves the goal. Moreover, we can prove

$$\frac{τ ⊨ \not\delta\, C; τ ⊨ C + 2 ≐ (7 :: \texttt{Integer}_τ) ∨ (D ≐ 2 + 3)}{τ ⊨ D ≐ 5} \tag{5.46}$$

by using a similar proof script:

```
lemma "⟦τ ⊨ ¡δ C; τ ⊨ C + 2 ≐ (7 :: Integer_τ) ∨ (D ≐ 2 + 3)⟧
       ⟹
       τ ⊨ D ≐ 5"
  apply(ocl_hypsubst, simp)
done
```

This script proves the goal by combining reasoning over undefinedness and arithmetic, i. e., from the undefinedness of C, we can directly infer $D ≐ 2 + 3$ must hold in the assumption; thus D is defined and by using the presented arithmetic rules, i. e., computing $2 + 3$, we can solve the proof goal.

5.4.4 *Conversion to HOL*

For a fragment of our constraint language, that is built for expressions that are always defined, a "conversion" into standard HOL formulae over Local Judgement Equivalences are possible. The propositional part of the translation is described in Table 5.3a, the predicative part in Table 5.3b. The rules for the other collection types are accordingly.

5.5 THE LOGIC

In this section, we introduce the core rules of the underlying logic, namely rules for reasoning over equalities, the logical connectives and also the integration of a typed set theory.

$$\frac{\tau \vDash \partial\, A}{(\tau \vDash \neg A) = (\neg\tau \vDash A)} \tag{5.47}$$

$$\frac{\tau \vDash \partial\, A \quad \tau \vDash \partial\, B}{(\tau \vDash A \wedge B) = (\tau \vDash A \wedge \tau \vDash B)} \tag{5.48}$$

$$\frac{\tau \vDash \partial\, A \quad \tau \vDash \partial\, B}{(\tau \vDash A \vee B) = (\tau \vDash A \vee \tau \vDash B)} \tag{5.49}$$

$$\frac{\tau \vDash \partial\, A \quad \tau \vDash \partial\, B}{(\tau \vDash A \longrightarrow B) = (\tau \vDash A \longrightarrow \tau \vDash B)} \tag{5.50}$$

(a) Propositional Conversion

$$\frac{\tau \vDash \partial(S :: (\beta :: \mathrm{bot})\ \mathtt{Set}_\tau) \quad \mathrm{cp}\, P}{(\tau \vDash \forall\, x \in S.\ P(x :: \tau \Rightarrow \beta)) = (\forall x.\ \tau \vDash x \in S \longrightarrow \tau \vDash P\, x)} \tag{5.51}$$

$$\frac{\tau \vDash \partial(S :: (\beta :: \mathrm{bot})\ \mathtt{Set}_\tau) \quad \mathrm{cp}\, P}{(\tau \vDash (\exists\, x \in S.\ P(x :: \tau \Rightarrow \beta))) = (\exists x.\ \tau \vDash x \in S \wedge \tau \vDash (P\, x))} \tag{5.52}$$

(b) Predicative Conversion

Table 5.3: Conversion to HOL

5.5.1 *Reasoning over Equality*

The strong equality satisfies the usual properties *except* the Leibniz rule (substitutivity):

$$\frac{}{\tau \vDash a \triangleq a}\,, \tag{5.53a}$$

$$\frac{\tau \vDash a \triangleq b}{\tau \vDash b \triangleq a}\,\text{, and} \tag{5.53b}$$

$$\frac{\tau \vDash a \triangleq b \quad \tau \vDash b \triangleq c}{\tau \vDash a \triangleq c}\,. \tag{5.53c}$$

Instead of substitutivity, the following, slightly weaker, form of substitutivity (for context passing P) holds:

$$\frac{\tau \vDash a \triangleq b \quad \tau \vDash P\ a \quad \mathrm{cp}\ P}{\tau \vDash P\ b}\,. \tag{5.53d}$$

This side-constraint is not surprising, since by Equation 5.19 shown in Section 5.3.2 we know that strong equality and global equivalence are semantically equivalent.

The following two lemmas

$$\frac{\tau \vDash \partial\, a \quad \tau \vDash \partial\, b}{(a \doteq b)\tau = (a \triangleq b)\tau} \tag{5.54a}$$

and

$$\frac{\tau \vDash a \doteq b}{\tau \vDash a \triangleq b} \tag{5.54b}$$

show that strict equality is indeed convertible into strong equality.

5.5.2 *Core-Logic (Boolean)*

core-logic propositional fragment

With *core-logic* we refer to the sub-language consisting of the logical connectives $\neg_$, $_\wedge_$, $_\vee_$, $_\longrightarrow_$, etc., which we also call the *propositional fragment* of our logic (in the sense of a propositional multi-valued logics). For each operator, we derive computational rules representing the truth tables of the logical connectives (see Definition 3.7 on page 57), e. g., for $_ \wedge _$ the following rules hold:

$$\mathtt{f} \wedge \mathtt{f} = \mathtt{f}, \qquad \mathtt{f} \wedge \mathtt{t} = \mathtt{f}, \qquad \mathtt{f} \wedge \bot = \mathtt{f}, \tag{5.55a}$$

$$\mathtt{t} \wedge \mathtt{f} = \mathtt{f}, \qquad \mathtt{t} \wedge \mathtt{t} = \mathtt{t}, \qquad \mathtt{t} \wedge \bot = \bot, \tag{5.55b}$$

$$\bot \wedge \mathtt{f} = \mathtt{f}, \qquad \bot \wedge \mathtt{t} = \bot\text{, and} \qquad \bot \wedge \bot = \bot. \tag{5.55c}$$

Beside such computational rules, the core-logic enjoys a lattice-like structure with the rules shown in Table 5.4a.

Table 5.4b shows the rules that deal with logical reasoning related to implication. However, the rules for Universal (Formula) Congruence do not form a complete calculus. The problem is hidden in the only conditional rule, which has to be rephrased as rule for Local (Formula) Equivalence to achieve completeness. Unfortunately, this conditional rule corresponds to applying the "assumption" and is therefore particularly vital in deduction. The Boolean case-split rule in Table 5.4c is interesting for automated reasoning. Applying case-splits consequently over all Boolean variables yields proof procedures of sufficient power, i. e., many facts over the library are proven using such a proof procedure.

5.5.3 *Set Theory and Logic*

membership
comprehensions

Set theory is the theory of *membership*, i. e., $x \in S$, on the one hand and set constructions like *comprehensions*, i. e., $(\!|x \in S | P\ x|\!)$, on the other.

In our framework, we have a typed form of a set theory which rules out, for example, Russel's Paradox. With respect to typedness, the set theory of our framework is more related to set theory of HOL, but more distant to the one of ZF. Undefinedness, on the other hand, is a distinguishing feature. As we already explained in Section 3.3.3, the inclusion of elements in a set may result in an undefined set (smashed semantics) or in a set, that just contains undefined elements (non smashed semantics). In this section, we will limit ourselves to the case of a smashed semantics for the selection types, which is also our proposal for the OCL semantics (see Section 4.3.4. On the deduction level, smashed semantics boils down to the following rule:

$$\frac{\tau \vDash x \in S}{(\tau \vDash \not\emptyset\, x) \wedge (\tau \vDash \not\emptyset\, S)}\ . \tag{5.59}$$

This has the consequence, that whenever we eliminate a universal or existential quantification, we know that the variable over which a quantifier ranges is defined. In itself, this is also useful to deduce that the body of a quantifier is defined.

In the following, we discuss the core of the collection theories at the example of the set theory. For brevity, we omit the type constraints from this presentation. Quantifiers and set constructors in our object-oriented framework have an operational character with respect to undefinedness; in the standard, quantifiers were defined via iterators, hence fold-like constructs which also reflect the behavior in case of undefinedness. This represents a particular challenge for deduction; on the other hand, many constraint languages over data-structures that support undefinedness, like OCL, Spec#, or VDM have the same characterizations.

$$\mathtt{f} \wedge X = \mathtt{f} \qquad \mathtt{t} \vee X = \mathtt{t} \tag{5.56a}$$

$$\mathtt{t} \wedge X = X \qquad \mathtt{f} \vee X = X \tag{5.56b}$$

$$X \wedge X = X \qquad X \vee X = X \tag{5.56c}$$

$$X \wedge Y = Y \wedge X \qquad X \vee Y = Y \vee X \tag{5.56d}$$

$$X \wedge (Y \wedge Z) = (X \wedge Y) \wedge Z \qquad X \vee (Y \vee Z) = (X \vee Y) \vee Z \tag{5.56e}$$

$$\neg(\neg X) = X \qquad (X \wedge Y) = \neg(\neg X \vee \neg Y) \tag{5.56f}$$

$$(X \vee Y) \wedge Z = (X \wedge Z) \vee (Y \wedge Z) \qquad (X \wedge Y) \vee Z = (X \vee Z) \wedge (Y \vee Z) \tag{5.56g}$$

$$Z \wedge (X \vee Y) = (Z \wedge X) \vee (Z \wedge Y) \qquad Z \vee (X \wedge Y) = (Z \vee X) \wedge (Z \vee Y) \tag{5.56h}$$

$$\neg(X \wedge Y) = \neg X \vee \neg Y \qquad \neg(X \vee Y) = \neg X \wedge \neg Y \tag{5.56i}$$

(a) Lattice

$$X \longrightarrow \mathtt{f} = \neg X \qquad X \longrightarrow \mathtt{t} = \mathtt{t} \tag{5.57a}$$

$$\mathtt{f} \longrightarrow X = \mathtt{t} \qquad \mathtt{t} \longrightarrow X = X \tag{5.57b}$$

$$\frac{\partial X = \mathtt{t}}{(X \longrightarrow X) = \mathtt{t}} \tag{5.57c}$$

$$X \longrightarrow Y = \neg X \vee Y \tag{5.57d}$$

$$X \longrightarrow (Y \wedge Z) = (X \longrightarrow Y) \wedge (X \longrightarrow Z) \tag{5.57e}$$

$$X \longrightarrow (Y \vee Z) = (X \longrightarrow Y) \vee (X \longrightarrow Z) \tag{5.57f}$$

$$(X \wedge Y) \longrightarrow Z = X \longrightarrow (Y \longrightarrow Z) \tag{5.57g}$$

$$(X \vee Y) \longrightarrow Z = (X \longrightarrow Z) \wedge (Y \longrightarrow Z) \tag{5.57h}$$

$$X \longrightarrow (Y \longrightarrow Z) = Y \longrightarrow (X \longrightarrow Z) \tag{5.57i}$$

(b) Logic

$$\frac{P\ \bot = P'\ \bot \quad P\ \mathtt{t} = P'\ \mathtt{t} \quad P\ \mathtt{f} = P'\ \mathtt{f} \quad \mathrm{cp}(P)\ \ \mathrm{cp}(P')}{P\ X = P'\ X} \tag{5.58}$$

(c) Boolean Case-Split: Trichotomy

Table 5.4: The Core of the UC Calculus ("Propositional Calculus")

For the quantifiers of our framework the following general Universal (Formula) Congruence-rules hold:

$$\frac{}{\forall\, x \in \bot.\ P\, x = \bot}\ , \tag{5.60}$$

$$\frac{}{\exists\, x \in \bot.\ P\, x = \bot}\ , \tag{5.61}$$

$$\frac{}{\forall\, x \in \varnothing.\ P\, x = \mathtt{t}}\ \text{, and} \tag{5.62}$$

$$\frac{}{\exists\, x \in \varnothing.\ P\, x = \mathtt{f}}\ . \tag{5.63}$$

The following two rules

$$\frac{\tau \vDash \partial\, X \quad \tau \vDash \partial\, a \quad \mathrm{cp}\, P}{(\forall\, x \in X \,\mathrm{insert}\, X\, a.\ P\, x)\ \tau = \big((P\, a) \wedge (\forall\, x \in X.\ P\, x)\big)\ \tau} \tag{5.64}$$

and

$$\frac{\tau \vDash \partial\, X \quad \tau \vDash \partial\, Y}{\big(\forall\, x \in (X \cup Y).\ P\, x\big)\ \tau = \big((\forall\, x \in X.\ P\, x) \wedge (\forall\, x \in Y.\ P\, x)\big)\ \tau} \tag{5.65}$$

allow for the elimination of quantifications over known finite sets via computation. Besides, there is a tableaux calculus for quantifier elimination that can be directly derived from the rules, extending the calculus shown in Table 5.3b.

The core of the set theory of our framework is the relation between the element-hood, i. e., membership ($x \in S$), and the set comprehension ($(\!|x \in S | P\, x|\!)$) and the relation to equality. In particular, they provide a form of set extensionality:

$$\frac{\tau \vDash a \in S \quad \tau \vDash \not\partial(P\, a) \quad \mathrm{cp}\, P}{\big((\!|x \in S | P\, x|\!)\big)\ \tau = \bot\tau}\ , \tag{5.66}$$

$$\frac{\begin{array}{c}\tau \vDash x \in S \\ \vdots \\ \tau \vDash \partial\, S \quad \tau \vDash \partial\, a \quad \bigwedge x.\ \ \tau \vDash \partial(P\, x) \quad \mathrm{cp}\, P\end{array}}{\big(\tau \vDash a \in ((\!|x \in S | P\, x|\!))\big) = \big(\tau \vDash P\, a \wedge \tau \vDash a \in S\big)}\ , \tag{5.67}$$

$$\frac{\begin{array}{c}\tau \vDash x \in S \\ \vdots \\ \tau \vDash \partial\, S \quad \tau \vDash \partial\, a \quad \tau \vDash a \in S \quad \bigwedge x.\ \ \tau \vDash \partial(P\, x) \quad \mathrm{cp}\, P\end{array}}{\big(\tau \vDash a \in ((\!|x \in S | P\, x|\!))\big) = \big(\tau \vDash P\, a\big)}\ \text{, and} \tag{5.68}$$

$$\frac{\begin{array}{c}\tau \vDash \partial\, x \\ \vdots \\ \bigwedge x.\ \ (x \in S)\ \tau = (x \in T)\ \tau\end{array}}{S\tau = T\tau}\ . \tag{5.69}$$

5.6 CALCULI

In this section, we present several calculi for our framework. In particular, we develop a tableaux calculus that is especially well-suited for automated deduction.

5.6.1 *A Universal Equational Calculus*

Universal Equational Calculus (UEC)

The basis of a *Universal Equational Calculus (UEC)* for our framework are Horn-clauses over universal congruences; due to the rich algebraic structure of Strong Kleene Logic, UEC allows for logical reasoning in formulae and local validity judgments. A proof of a formula in UEC is a derivation of a formula to $\mathtt{t}$.

reduction rules (R-rules)

Based on the elementary *reduction rules (R-rules)* for the logical operators, it is not difficult to derive the laws of the surprisingly rich algebraic structure of Strong Kleene Logic: both $_ \wedge _$ and $_ \vee _$ enjoy associativity, commutativity and idempotency. The logical operators also satisfy both distributivity and the de Morgan laws. It is essentially this richness and algebraic simplicity that we will exploit in the applications.

The logical implication is also representable in this equational reasoning style, which is quite intuitive and therefore greatly facilitates "by-hand-proofs," see Table 5.4b. These rules form the core of the logical calculus. However, the crucial assumption rule

$$\frac{\partial(X) = \mathtt{t}}{(X \longrightarrow X) = \mathtt{t}} \tag{5.70}$$

that allows one to deduce that a fact follows from a list of assumptions leads to a complication:

$$\begin{aligned} & A_1 \wedge \cdots \wedge A_k \wedge B \wedge A_{k+1} \wedge \cdots \wedge A_n \longrightarrow B \\ = {} & A_1 \wedge \cdots \wedge A_n \wedge B \longrightarrow B \\ = {} & A_1 \wedge \cdots \wedge A_n \longrightarrow (B \longrightarrow B) \end{aligned}$$

only under the additional assumption $\partial(B) = \mathtt{t}$, we can conclude

$$= A_1 \wedge \cdots \wedge A_n \longrightarrow \mathtt{t}$$

and thus, resolve our proof goal to:

$$= \mathtt{t}$$

This means that a subcalculus for the definedness predicate is needed. Moreover, this means that each application of the assumption rule leads to a sub-proof over the definedness of the assumption. Therefore, we consider two alternative definitions of the implication, namely $_ \overset{1}{\longrightarrow} _$ (defined in Equation 3.39b) [52] and $_ \overset{2}{\longrightarrow} _$ (defined in Equation 3.39c) [56].

Recall, that a comparison of all three implication variants is given in Table 3.4 on page 58.

For the first variant, i. e., $_ \xrightarrow{1} _$, the counterparts to rules appearing in Table 5.4b on page 148 hold, except for two details:

1. We have $(X \xrightarrow{1} X) = \mathtt{t}$ such that the subproof for $\partial B = \mathtt{t}$ is not necessary. The handling of this implication in proofs is therefore more intuitive. In principle, this could be a motivation to prefer this variant of implication over the default one ($_ \xrightarrow{1} _$). In particular, if a difference in proof complexity could be shown.
2. However, the problem is only shifted to the "reductio ad absurdum"-rule $(X \longrightarrow \mathtt{f}) = \neg X$, whose counterpart requires now the proviso for definedness.

For the second variant, i. e., $_ \xrightarrow{2} _$, the situation is even worse: besides the obvious fact, that the crucial rule $\bot \xrightarrow{2} \mathtt{t}$ does *not* hold, also the following rules from Table 5.4b do not hold:

$$((X \vee Y) \xrightarrow{2} Z) = (X \xrightarrow{2} Z) \wedge (Y \xrightarrow{2} Z)\,, \tag{5.71}$$

$$((X \wedge Y) \xrightarrow{2} Z) = (X \xrightarrow{2} (Y \xrightarrow{2} Z))\,, \text{ and} \tag{5.72}$$

$$X \xrightarrow{2} (Y \xrightarrow{2} Z) = Y \xrightarrow{2} (X \xrightarrow{2} Z)\,. \tag{5.73}$$

As both non-standard variants introduce, from the deduction point of view, unnecessary complications (especially the second variant with its dramatic algebraic deficiencies), we will in the following only consider the standard implication, i. e., $_ \longrightarrow _$.

A FURTHER PROOF PRINCIPLE OF UEC: TRICHOTOMY. An interesting technique for proving $P\ X = P'\ X$ is based on a case split over $\bot$, $\mathtt{t}$ or $\mathtt{f}$. The enabling rule Table 5.4c is called *trichotomy*; it requires a particular constraint over the treatment of the implicit context τ inside P and P'. In principle, it would suffice to require that τ is changed "on its way through P and P'" in the same way. However, since all constructs of our framework, including the logical connectives are lifted over the contexts, we apply a slightly stronger restriction, namely that τ is unchanged, i. e., P or P' are *context passing* with respect to τ. It turns out that this concept is necessary for other calculi, too. As already discussed in Section 5.4.1, the property of being context passing can easily inferred in a backward proof whose size is equal to the size of the term. These inferences use inherently higher-order concepts.

trichotomy

$$\frac{A\ \tau = A'\ \tau}{(\neg A)\ \tau = (\neg A')\ \tau} \qquad \frac{A\ \tau = A'\ \tau \quad B\ \tau = B'\ \tau}{(A \vee B)\ \tau = (A' \vee B')\ \tau} \qquad \frac{A\ \tau = A'\ \tau \quad B\ \tau = B'\ \tau}{(A \oplus B)\ \tau = (A' \oplus B')\ \tau} \tag{5.74}$$

(a) Congruence Rules for the Operations $\neg_$, $_ \vee _$, and $_ \oplus _$.

$$\frac{\tau \vDash \partial(A) \qquad \begin{array}{c}[\tau \vDash A]\\ \vdots \\ B\ \tau = B'\ \tau\end{array}}{(A \wedge B)\ \tau = (A \wedge B')\ \tau} \qquad \frac{\tau \vDash \partial(A) \qquad \begin{array}{c}[\tau \vDash A]\\ \vdots \\ B\ \tau = B'\ \tau\end{array}}{(A \longrightarrow B)\ \tau = (A \longrightarrow B')\ \tau} \tag{5.75}$$

$$\frac{\begin{array}{c}[\tau \vDash A]\\ \vdots \\ B\ \tau = B'\ \tau\end{array} \qquad \begin{array}{c}[\tau \vDash_{\mathsf{f}} A]\\ \vdots \\ C\ \tau = C'\ \tau\end{array}}{(\mathtt{if}\ A\ \mathtt{then}\ B\ \mathtt{else}\ C\ \mathtt{endif})\ \tau = (\mathtt{if}\ A\ \mathtt{then}\ B'\ \mathtt{else}\ C'\ \mathtt{endif})\ \tau} \tag{5.76}$$

(b) Context Rules for the operations $_ \wedge _$, $_ \longrightarrow _$, and $\mathtt{if}\ _\ \mathtt{then}\ _\ \mathtt{else}\ _\ \mathtt{endif}$.

$$\frac{\tau \vDash A \qquad P\ \mathsf{t}\ \tau = P'\ \mathsf{t}\ \tau \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ \tau = P'\ A\ \tau} \tag{5.77}$$

$$\frac{\tau \vDash_{\mathsf{f}} A \qquad P\ \mathsf{f}\ \tau = P'\ \mathsf{f}\ \tau \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ \tau = P'\ A\ \tau} \tag{5.78}$$

$$\frac{\tau \vDash_{\bot} A \qquad P\ \bot\ \tau = P'\ \bot\ \tau \qquad \mathrm{cp}(P) \qquad \mathrm{cp}(P')}{P\ A\ \tau = P'\ A\ \tau} \tag{5.79}$$

(c) Propagation of the Local Validity.

Table 5.5: The Local Equational Calculus (LEC).

5.6.2 *A Local Equational Calculus*

Analogously to universal equality, a local validity calculus can be developed: *Local Equational Calculus* (LEC). Table 5.5a shows the general scheme of Local Equational Calculus (LEC) congruence rules. For example, this general schema is applicable for the negation ($\neg_$), disjunction ($_ \vee _$) or the exclusive or ($_ \oplus _$). For several operators, stronger logical rules can be derived, that accumulate semantic knowledge for sub-derivations from the context in which they are applied in; these rules are presented in Table 5.5b. This information can be used by the third group of rules in Table 5.5c, which allows for generalizing sub-terms in larger contexts (which must be context-passing) according to assumptions.

The calculus LEC is particularly suited for backward-proofs; when applied bottom-up, formulae were decomposed deterministically via the congruence and the context rules. During this process, semantic context knowledge is accumulated in the assumption list, which can be exploited via the propagation rules who replace sub-terms by $\mathtt{t}$, $\mathtt{f}$, or $\bot$ which leads in combination with UEC in practice to drastic simplifications of the current proof goal. A proof in LEC is a derivation that leads to $A\tau = \mathtt{t}\tau$, which is notationally equivalent to $\tau \vDash A$.

5.6.3 *The Judgment Tableaux Calculus LTC*

The conversion technique discussed in Section 5.4.4 requires reasoning on side-conditions such as cp P or definedness ∂X. The question arises, if this can be avoided when performing a logical decomposition of the $\mathcal{L}$ formulae directly.

The tableaux methodology is one of the most popular approaches to design and implement proof-procedures. While originally developed for first-order theorem proving, renewed research activity is being devoted to investigating tableaux systems for intuitionistic, modal, temporal and many-valued logics, as well as for new families of logics, such as non-monotonic and sub-structural logics. Many of these recent approaches are based on a special labeling technique on the level of judgments, called labeled deduction [7, 43, 112]. Of course, labeling can also be embedded into a higher-order, classical meta-logic. Being a special case of a many-valued logic, tableaux calculi for Strong Kleene Logic based on labeled deduction have been extensively studied [49, 51, 64]. In this section, we present a tableaux calculus for the predicative fragment of $\mathcal{L}$, i. e., for Strong Kleene Logic roughly following [64]. It is designed to be processed by the generic proof procedures of Isabelle and thus leads directly to an implementation in HOL-OCL.

Tableau proofs may be viewed as trees where the nodes are sets of formulae. Tableau rules extend the leaves of a tree by a new subtree, i. e., by adding leaves below, where the latter case is called "branching" and is used for case splits. Classical tableau rules capture the full logical content of the expanded connective. Backtracking from a rule application is never necessary. The goal of the process is to construct trees in a deterministic manner, where the leaves can eventually all be detected as "closed," i. e., a logical contradiction is detected. This last step, however, may be combined with the non-deterministic search for a substitution making this contradiction possible. Table 5.6 presents the core of LTC, which we will discuss in the sequel in more detail.

A NATURAL DEDUCTION TABLEAUX CALCULUS FOR HOL. The particular format of a rule as a Horn-clause is the reason for the well-

$$\frac{\tau \vDash \partial(A) \qquad \begin{matrix}[\tau \vDash A]\\ \vdots\\ R\end{matrix} \qquad \begin{matrix}[\tau \vDash \neg A]\\ \vdots\\ R\end{matrix}}{R} \qquad \frac{\begin{matrix}[\neg(\tau \vDash \neg A)]\\ \vdots\\ \tau \vDash A\end{matrix}}{\tau \vDash \partial(A)} \tag{5.80}$$

$$\frac{\begin{matrix}[\neg(\tau \vDash \neg A)]\\ \vdots\\ \tau \vDash A\end{matrix} \qquad \tau \vDash \not\partial A}{\tau \vDash \neg\partial(A)} \qquad \frac{\begin{matrix}[\neg(\tau \vDash \neg A)]\\ \vdots\\ \tau \vDash A\end{matrix} \qquad \tau \vDash \neg\partial(A)}{\tau \vDash \not\partial A} \tag{5.81}$$

(a) Definedness Introduction and Elimination

$$\frac{\tau \vDash \neg(\neg A)}{\tau \vDash A} \qquad \frac{\tau \vDash A}{\tau \vDash \neg(\neg A)} \qquad \frac{\tau \vDash \not\partial(\neg A)}{\tau \vDash \not\partial A} \qquad \frac{\tau \vDash \not\partial A}{\tau \vDash \not\partial(\neg A)} \tag{5.82}$$

(b) Negation

$$\frac{\tau \vDash A \wedge B \qquad \begin{matrix}[\tau \vDash A, \tau \vDash B]\\ \vdots\\ R\end{matrix}}{R} \qquad \frac{\tau \vDash \neg(A \wedge B) \qquad \begin{matrix}[\tau \vDash \neg A]\\ \vdots\\ R\end{matrix} \qquad \begin{matrix}[\tau \vDash \neg B]\\ \vdots\\ R\end{matrix}}{R} \tag{5.83}$$

$$\frac{\begin{matrix}[\tau \vDash A, \tau \vDash B]\\ \vdots\\ R\end{matrix} \qquad \tau \vDash A \;\; \tau \vDash B}{\tau \vDash (A \wedge B)} \qquad \frac{\begin{matrix}[\neg(\tau \vDash \neg B)]\\ \vdots\\ \tau \vDash \neg A\end{matrix}}{\tau \vDash \neg(A \wedge B)} \tag{5.84}$$

$$\frac{\begin{matrix}[\tau \vDash \partial(B))]\\ \vdots\\ \tau \vDash \neg\partial(A)\end{matrix} \qquad \begin{matrix}[\tau \vDash \partial(A)]\\ \vdots\\ \tau \vDash A\end{matrix} \qquad \begin{matrix}[\tau \vDash \partial(B)]\\ \vdots\\ \tau \vDash B\end{matrix}}{\tau \vDash \not\partial(A \wedge B)} \tag{5.85}$$

$$\frac{\tau \vDash \not\partial(A \wedge B) \qquad \begin{matrix}[\tau \vDash \not\partial A, \tau \vDash \not\partial B]\\ \vdots\\ R\end{matrix} \qquad \begin{matrix}[\tau \vDash \not\partial A, \tau \vDash B]\\ \vdots\\ R\end{matrix} \qquad \begin{matrix}[\tau \vDash A, \tau \vDash \not\partial B]\\ \vdots\\ R\end{matrix}}{R} \tag{5.86}$$

(c) Conjunction Introduction and Elimination

$$\frac{\tau \vDash A \;\; \tau \vDash \neg A}{R} \qquad \frac{\tau \vDash A \;\; \tau \vDash \not\partial A}{R} \qquad \frac{\tau \vDash \neg A \;\; \tau \vDash \not\partial A}{R} \tag{5.87}$$

$$\frac{\tau \vDash \not\partial\partial(A)}{R} \qquad \frac{\tau \vDash A}{\tau \vDash \partial(A)} \qquad \frac{\tau \vDash \neg A}{\tau \vDash \partial(A)} \tag{5.88}$$

(d) Contradictions

Table 5.6: The core of LTC.

known symmetry of rule-sets (similar to sequent calculi): for each logical connective, two corresponding rules, called *introduction* and *elimination* rules, have to be introduced; the former act on conclusions of a goal, the latter on one of the assumptions.

As already mentioned, the Tableaux Method requires decomposition rules that capture the full logical content of the expanded connective, i. e., no information is lost. It is instructive to consider the example of the disjunction introduction rule (Equation 5.89) and the disjunction elimination rule (Equation 5.90) for HOL, which are usually presented in the textbooks as:

$$\frac{\begin{array}{c}[\neg B]\\ \vdots\\ A\end{array}}{A \vee B} \text{ (Disjunction Introduction)} \tag{5.89}$$

$$\frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots\\ R\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ R\end{array}}{R} \text{ (Disjunction Elimination)} \tag{5.90}$$

Using the introduction rule (Equation 5.89), a proof state (which has again the format of a Horn-clause)

$$\frac{X}{Y \vee Z} \tag{5.91}$$

can be transformed into

$$\frac{X \qquad \neg Z}{Y}\,. \tag{5.92}$$

This proof state transformation does not lose the information that the goal is satisfiable if Z holds (this leads to a contradiction in the assumption list). Moreover, a proof state of the form

$$\frac{X \vee Y \qquad A}{Z} \tag{5.93}$$

can be transformed via the elimination rule (Equation 5.90) into the two subgoals

$$\frac{X \qquad A}{Z} \quad \text{and} \quad \frac{Y \qquad A}{Z}\,. \tag{5.94}$$

Overall, this proof state transformation performs a case distinction.

Keeping these remarks in mind, the presentation of LTC in Table 5.6 is pretty much straight-forward: first, we present groups of tableaux rules

for definedness ($\partial_$), negation ($\neg_$) and conjunction ($_ \wedge _$), and we conclude with a group of rules for closing clauses. Moreover, negative judgments can be replaced by HOL-disjunctions as a consequence of the following fundamental property for judgments, namely that judgment is either valid or invalid or undefined (see Section 5.2.2).

Four (see Table 5.6a) of the six cases for definedness rules are straightforward, while the latter two constitute contradictions and are presented as closure rules later (see Table 5.6d).

Now, we consider the case for the negation ($\neg_$) in Table 5.6b. These rules are a consequence of $\neg(\neg(X)) = X$ and eliminate these situations at the root of a formula. These elimination rules have a particularly simple form and can therefore be used directly as *destruction rules*, i. e., a rule that destroy a premise, that are used to weaken assumptions in a Horn-clause directly. The last two rules of the group are notational equivalences resulting from our notational conventions for validity ($\models_$); they are thus not explicitly inserted into the rule set.

destruction rules

Finally, we describe the rules for closing a goal. The underlying HOL system already includes three rules that resolve satisfiable Horn-clauses, namely the (HOL) not-elimination, classical contradiction, not-introduction rule and the assumption rule (from left to right):

$$\frac{\neg P \qquad P}{R} \qquad \frac{\begin{array}{c}[\neg P]\\ \vdots\\ \text{false}\end{array}}{P} \qquad \frac{\begin{array}{c}[P]\\ \vdots\\ \text{false}\end{array}}{\neg P} \qquad \frac{P}{P} \tag{5.95}$$

Besides these HOL-logical rules for closing a goal, there are also $\mathcal{L}$-logical rules motivated by the satisfiability of a Horn-clause (Table 5.6d).

This gives rise to a useful format of a proof state in LTC; it is a Horn-clause of one of the two forms:

$$\frac{H_1 \dots H_i, \neg(H_{i+1}) \dots \neg(H_m)}{H_{m+1}} \qquad \frac{H_1 \dots H_i, \neg(H_{i+1}) \dots \neg(H_m)}{\neg(H_{m+1})} \tag{5.96}$$

where H_i has the form $\tau \vDash_{C_i} A_i$. Standard proof states in UEC can be converted automatically in proof states of this form via one of the bridge-theorems (Equation 5.3a, Equation 5.3b, and Equation 5.3c on page 134). The elimination and introduction rules shown above reduce or split proof steps of this form in logically equivalent steps to new ones. Eventually, the process results in a sequence of Horn-clauses with labeled literals.

HANDLING QUANTIFIERS. In the following, we discuss an extension of the propositional $\mathcal{L}$ fragment to a language with bounded quantifiers introduced for collections. For brevity, we will concentrate on the quantifiers on $\texttt{Set}_\tau$.

$$\frac{\bigwedge x.\quad \tau \vDash P(?x)}{\tau \vDash \forall\, x \in S.\ P(x)} \qquad \frac{\tau \vDash \partial\, S}{\tau \vDash \not\partial\, P(?x)} \tag{5.97}$$

$$\frac{\tau \vDash \forall\, x \in S.\ P(x) \qquad [\tau \vDash ?x \in S] \quad [\tau \vDash \neg ?x \in S]}{R} \tag{5.98}$$

$$\frac{\tau \vDash \not\partial\, \forall\, x \in S.\ P(x) \qquad [\tau \vDash \not\partial\, S] \quad [\tau \vDash ?x \in S, \tau \vDash \not\partial\, P(?x)]}{R} \tag{5.99}$$

$$\frac{\tau \vDash ?x \in S \quad \tau \vDash \neg\, P(?x)}{\tau \vDash \neg\, \forall\, x \in S.\ P(x)} \tag{5.100}$$

$$\frac{\tau \vDash \neg\, \forall\, x \in S.\ P(x) \quad \bigwedge x. \qquad [\tau \vDash x \in S, \tau \vDash P(x)]}{R} \tag{5.101}$$

(a) Skolem

$$\partial(X) = \mathtt{t} \Rightarrow \forall\, x \in X.\ \mathtt{t} = \mathtt{t} \tag{5.102}$$

$$\partial(X) = \mathtt{t} \Rightarrow \forall\, x \in X.\ \mathtt{f} = \mathtt{f} \tag{5.103}$$

$$\forall\, x \in X.\ P(x) \wedge Q(x) = \forall\, x \in X.\ P(x) \wedge \forall\, x \in X.\ Q(x) \tag{5.104}$$

(b) Distributivity

$$\frac{\mathrm{cp}(P) \quad \bigwedge x.\ \mathrm{cp}(P'x)}{\mathrm{cp}(\lambda\, X.\ \forall\ (P\, X)\ (\lambda\, x.\ (P'\, x\, X)))} \tag{5.105}$$

(c) Context Passing for Quantifiers

Table 5.7: Extensions of LTC: quantifiers.

First, we present some universal equalities of the universal quantifiers, which also satisfy the usual context passing rules (Table 5.7c). With respect to strictness rules, the universal quantifier (and its dual the existential quantifier) follows the usual scheme:

$$\forall\, x \in \bot.\ P\, x = \bot\,, \tag{5.106}$$

$$\exists\, x \in \bot.\ P\, x = \bot\,, \tag{5.107}$$

$$\forall\, \bot \in X.\ P = \bot\,, \text{ and} \tag{5.108}$$

$$\exists\, \bot \in X.\ P = \bot\,. \tag{5.109}$$

The distributivities of Strong Kleene Logic can be extended to the quantifiers, see Table 5.7b. If $x \in S$ is valid, we know that x must be defined. This is a characteristic property of smashed sets that yields the following property:

$$\frac{\tau \vDash x \in S}{\tau \vDash \partial\, x} \tag{5.110}$$

We present the rules for the bounded universal quantifier. These rules are simply variations of standard quantifier rules in HOL, but subsume also definedness-reasoning. We start with the usual introduction and elimination rules for valid judgments. We use meta-quantifier and meta-variables to represent Skolem terms and terms for witnesses; respectively (constructed during the proof at need via unification and resolution), see Table 5.7a.

The following introduction and elimination rules capture the essence for undefined quantifiers: if the set S is defined (implying that each element in it is defined), then there must be an instance of the quantifier body P that is undefined. On the other hand, from an undefined quantifier we have a case split for undefined S or for witnesses of undefined $P(?x)$: The rules for the existential quantifiers follow easily from the definition and rules above and are omitted here.

5.7 DISCUSSION

In this chapter, we presented derived calculi, i. e., all presented rules are proven, for the object-oriented constraint language $\mathcal{L}$. Deriving the calculi guarantees the logical soundness, with respect to the core logic HOL, of all presented calculi. As we focused on the specific configuration of our framework described in Chapter 4 we will discuss briefly, how $\mathcal{L}$ and OCL relate to each other. A second point we discuss briefly in this section is the aspect of automated reasoning, based on the presented calculi, within HOL-OCL.

5.7.1 *Calculi for OCL*

The OCL standard does in general not provide any information how reasoning over OCL specifications should be carried out, and in particular, it does not present any calculi. But already the semantics presented in the standard [88, Appendix A] makes heavy use of the notion of validity. In particular, it already introduces the notion of *local validity*. For example, the validity for postconditions is explained as follows:

$$(\sigma_{\text{pre}}, \sigma_{\text{post}}) \vDash Q \quad \text{iff} \quad I[\![Q]\!](\sigma_{\text{pre}}, \sigma_{\text{post}}) = \text{true}$$

(OCL Specification [88], page A.33)

To avoid confusion with our notion of context τ, we changed the definition of the standard to use σ for referring to the pre-state and post-state, which the standard denotes in this section by τ_{pre} and τ_{post}. Moreover, in this definition, the variable assignment β, as occurring in the OCL standard is hidden syntactically. Denoting β explicitly, results in the following definition of local validity:

$$(\sigma_{\text{pre}}, \sigma_{\text{post}}) \vDash Q \quad \text{if and only if} \quad I[\![Q]\!]((\sigma_{\text{pre}}, \beta), (\sigma_{\text{post}}, \beta)) = \text{true} \tag{5.111}$$

for all variable assignments β. As we use a shallow embedding technique into a typed domain, the variable assignment is superfluous. Moreover, we abbreviate the state pair $(\sigma_{\text{pre}}, \sigma_{\text{post}})$ with the context τ. Therefore, our notion of local validity, introduced in Section 5.2.1 is a generalization of the notion of the standard. Thus, the calculi we presented in this chapter are directly applicable to UML/OCL specifications.

5.7.2 *Towards Automated Deduction*

At the moment, HOL-OCL provides only proof procedures for substitution that are specific to $\mathcal{L}$, respectively OCL. In fact, these proof procedures are the counterpart of substitution tactics Isabelle [86] provides for HOL. These substitution tactics for $\mathcal{L}$ serve as an abstract interface to the various rules that express local congruences, strict and strong equalities and congruences hidden in local judgments as pointed out in Section 5.2.2.

For the moment, this tactic setup allows for a step-by-step reasoning using the rules of the logic and UC-rules (including computational rules) in the Isabelle simplifier. Provided that sufficient information on the definedness of free variables is available in a proof state, this enables a conversion to HOL formulae with the rules discussed in Section 5.4.4 possible. A converted formula can be treated by the standard Isabelle proof procedures. This covers a certain extent of logical reasoning automatically.

However, the situation is clearly not yet satisfactory for larger, application oriented projects involving formal proofs. Here is a list of the most painful shortcomings when comparing it with proofs in "pure" Isabelle/HOL:

- The library on datatypes and on datatype-oriented rules is far from being sufficiently developed.
- An arithmetic decision procedure is missing.
- All rules of the LE-format are not usable by the simplifier. Since the complete core calculus and many datatype-oriented rules are in this format, the proof engineer is limited to elementary proof techniques excluding the simplifier whenever these rules are involved in a proof.
- Rules of the LE-format are also excluded from the classical reasoner.
- Our systems lacks a combined automatic tactic integrating all these procedures. Such an integrated tactic would be applicable as automatic procedures in many situations.

Partly, the situation is comparable to HOL ten years ago—and a fair comparison to similar logical languages has to take into account that the development of HOL libraries and proof procedures needed this time. For OCL, the development of proof procedures and, more critically, the technical support of formal methods based on OCL is still at the beginning. The latter will have to cope with path-expressions, modifies-only-clauses, and refinement-like situations.

In the following, we will summarize our ideas about potential future tactics to reason over OCL automatically.

With respect to arithmetic, besides a step-by-step reasoning, only the following paths to use automated procedures seem to be viable: defined arithmetic terms have to be converted (by unfolding semantic combinators and blowing away the cascades of definedness-conditions) into pure HOL arithmetic formulae and reuse the existing procedure (the adaption approach). Alternatively, the arithmetic tactic of Isabelle must be rewritten to cope with definedness issues (the re-engineering approach).

With respect to rewriting, we see (besides the not very attractive re-engineering approach) the following techniques to adapt to existing Isabelle technology:

Proof object transformation. Since one can instantiate the simplifier with new equalities obeying the Leibniz rule, one can run it in an unsafe-mode without checking the side conditions for context passingness. Proof objects generated in an unsafe mode could be extended to

full proof objects where the missing parts are reconstructed. It remains to be explored how costly this approach would be (in development time as well as runtime; previous experience [96] suggests that at least the runtime costs are insignificant).

Making context-passing explicit. One can transform the proof states and rules in a format where context-passingness is encoded directly at all positions in a term. As a consequence, the simplifier can process the transformed rules directly. Additionally to the conversion tactics that perform this term-transformation in forward and backward proof, the major changes for this technique boil down to the management of transformed and re-transformed rule-sets used by the simplifier.

In the next chapter, we will present a brief overview of the HOL-OCL, an instance of our framework providing an interactive theorem prover for UML/OCL, system architecture. Further, we report on case studies we carried out using HOL-OCL.

6 APPLICATIONS

In this chapter, we show the application of the framework and its calculi we presented in the previous chapters. In Section 6.1 we give an overview of the system architecture of the HOL-OCL system and in Section 6.2 we report on case studies that we carried out using HOL-OCL.

6.1 THE HOL-OCL SYSTEM

HOL-OCL

In this section, we present the HOL-OCL system. *HOL-OCL* is an interactive theorem prover for UML/OCL specifications that we developed on the basis of our framework, i. e., it is developed as a conservative, shallow embedding into Isabelle/HOL. This construction ensures the consistency of our logical framework and also the correctness with respect to the semantics presented in the previous chapters.

6.1.1 *An Architectural Overview*

HOL-OCL is integrated into a framework [27] supporting a formal, model-driven software development process. Technically, HOL-OCL is based on a repository for UML/OCL models, called su4sml, and on Isabelle/HOL; both are written in Standard Meta Language (SML). In particular, HOL-OCL is based on the SML interface of Isabelle/HOL and the UML/OCL model repository. As front-end, HOL-OCL provides a special instance of Proof General [5] and a documentation generation. Figure 6.1 on the next page gives an overview over the main system components of HOL-OCL. In this section, we briefly describe the main components of the HOL-OCL system, namely:

- The underlying data repository, called *su4sml*, which provides the XMI import facilities. *su4sml*
- The *datatype package*, or *encoder*, which encodes UML/OCL models into HOL-OCL, i. e., from a user's perspective it provides the XMI import facilities. *datatype package*

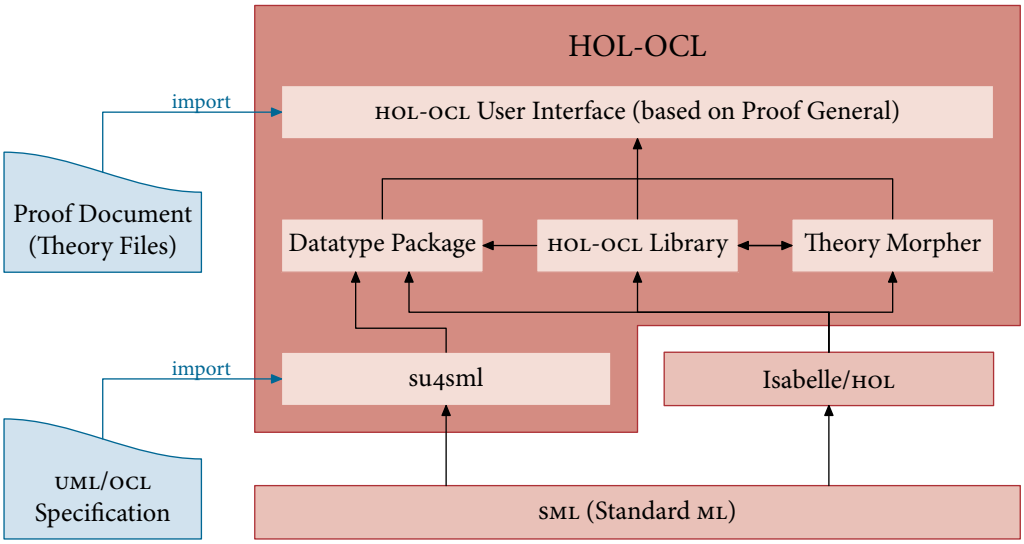


Figure 6.1: Overview of the HOL-OCL architecture. Specifications written in UML/OCL are imported into the model repository su4sml, written in SML. The theory morpher for lifting proven lemmas from the HOL to the OCL level is developed on top of Isabelle/HOL and is in itself the base tool for developing the HOL-OCL library. The datatype package encodes the UML/OCL models and proves already several properties over the specification. The formal analysis of the specification is carried out using a user interface based on Proof General.

- The *theory morpher* which derives many of the core OCL theorems by "lifting" them based on the corresponding theorems already proven for HOL. *theory morpher*
- The *HOL-OCL library* which provides the core theorems needed for verification and also a formal semantics for OCL. *HOL-OCL library*

6.1.2 *The Model Repository: su4sml*

The model repository *su4sml* [27] provides an interface to models expressed using the UML core (mainly class diagrams and statemachines) and OCL expressions. For these models, su4sml provides an import mechanism based on the XML Metadata Interchange (XMI) [92]. XMI is a standardized, Extensible Markup Language (XML)-based file format for exchanging UML models. Most Computer Aided Software Engineering (CASE) tools for UML can export models in XMI, which then can be imported into HOL-OCL.

For class models, su4sml resembles the tree structure given by the containment hierarchy. For example, a class contains attributes, operations, or statemachines. OCL expressions naturally translate into an abstract SML datatype in SML. This abstract datatype is modeled closely following the standard OCL 2.0 metamodel. In addition to these datatype definitions, the repository structure defines a couple of normalization functions, for example for converting association ends into attributes with corresponding type, together with an invariant expressing the cardinality constraint.

6.1.3 *The Encoder: An Object-oriented Datatype Package*

Encoding object-oriented data structures in HOL, as described in Section 3.4, is a tedious and error-prone activity if done manually. Thus it should be supported by an automated datatype package. In the theorem prover community, a *datatype package* [77] is a module that allows one to introduce new datatypes and automatically derive certain properties over them.

During the encoding, our datatype packages extends the given theory by a HOL-OCL-representation of the given UML/OCL model. This is done in an extensible way, i. e., classes can be added later on to an existing theory preserving all proven properties. In fact, any import represents such an extension, even the very first one which extends the HOL-OCL library. The obvious tasks of the datatype package are:

1. declare HOL types for the classifiers of the model,
2. encode the core data model and the operations defined on it into HOL, and

3. encode the OCL specification (including invariants and operation specifications) and combine it with the core data model.

Overall, the datatype package encodes conservatively the user supplied model following the schema presented in Section 3.5. Among others, this includes the definition of type and kind sets or the operation specifications. Moreover, the datatype package automatically proves several properties over the encoded model. In fact, the most important task is probably not that obvious: the package has to generate formal proofs that the generated encoding of object-structures is a faithful representation of object-orientation. This strategy, i. e., stating entirely conservative definitions and formally proving the datatype properties for them, ensures two very important properties:

1. our encoding fulfills the required properties (see Section 4.4.3), otherwise the proofs would fail, and

2. doing all definitions conservatively together with proving all properties ensures the consistency of our model (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

6.1.4 *The Theory Morpher*

The *theory morpher* provides automatic support for lifting theorems from the HOL level to the HOL-OCL level. This is based on our organization of the library, i. e., as a layered theory morphism. The theory morpher, or lifter, is a tactic-based program that lifts meta-level theorems to their object-level counterparts and meta-level prover configurations to object-level ones.

Such an automatic theory morpher is possible because we define our shallow embedding along a global semantic transformation from one language level to another. We generalized the underlying conceptual notions into a generic framework that shows that the overall technique is applicable in a wide range of embeddings in type systems; embedding-specific dependencies arise only from the specifications of semantic combinator (the layers), and technology specific dependencies from the used tactic language.

6.1.5 *The Library*

An important part of HOL-OCL is a collection of Isabelle theory files describing the formalization of the framework presented in Chapter 3 and Chapter 4. These theories, providing over 10 000 UML/OCL specific definitions and theorems, cover the core of UML/OCL; the properties of basic types such as `Integer`, `Real`, and `String` as well as collection types such

as `Bag`, `Sequence` and `Set`, and also the common superclass `OclAny`. Besides the model-specific part covered by the datatype package described in Section 6.1.3, the library with its body of derived rules represents the generic part of data-structure related reasoning in OCL. Moreover, these theories also contain new proof tactics written in SML.

6.2 CASE STUDIES

In this section, we report briefly on case studies we carried out using HOL-OCL. In particular, we report on our experiences in using a conservative import mechanism based on a datatype package and show how basic properties of a user supplied specification can be proven.

6.2.1 *Encoding User Specifications*

Among others, we used the following specifications as case studies for HOL-OCL:

Invoice: This is a model of a simple warehouse. It is a well-known case study for comparing specification formalisms, e. g., Frappier and Habrias [42] give an overview of several formalizations of this case study using different formalisms. The complete specification of this model is presented in Appendix B.

eBank: This model is an extension of the model presented in Section 2.1.2, e. g., customers can also trade currencies and own a checkbook.

Company: This is a simple company model which is used in the OCL standard [88, Chapter 7] for introducing OCL informally.

Royals and Loyals: This is a model of a bonus card system for customers. This model is used by Warmer and Kleppe [116] for introducing OCL.

All these models are imported into HOL-OCL using our conservative datatype-package (see Section 6.1.3). This requires proofs of several properties which can fail for inconsistent models. Already for the smallest model, Invoice, over 600 theorems are proven fully automatically by our datatype package during import of that model. Among others, the following properties are proven during import:

- For each class invariant, the co-recursive construction described in Section 3.5.2 is performed. This also includes proofs that the invariant representation used in this construction is monotone. If an invariant is supplied by the user that is not monotone, the proof will fail and the model will therefore be rejected during import.

	Invoice	eBank	Company	Royals and Loyals
number of classes	3	8	7	13
number of attributes	5	15	10	27
number of associations	3	5	6	12
number of operations	7	8	2	17
number of generalizations	0	3	0	2
size of OCL specification (lines)	149	114	210	520
generated theorems	647	1444	1312	2516
time needed for import (in seconds)	12	42	49	136

Table 6.1: Importing different UML/OCL specifications into HOL-OCL. The number of generated theorems depends on the size of the input model, i. e., of the number of classes, attributes, associations, operations, generalizations, and OCL constraints; the time for encoding the models depends on the number of theorems generated and also on their complexity.

- For each class, it is proven that the objects can be cast along the generalization (subtyping) hierarchy, see Section 4.4.3 for details.
- For each 2nd level constant (see Section 3.2) defined during import, the package tries to prove that the constant is strict and context-passing. The model can still be imported if these proofs fail.

All these proofs use the calculi presented in Section 5.6 in combination with both specialized tactics we developed ourselves and standard Isabelle tactics. Moreover, the conservative definitions for overloaded operations are generated. This includes user-defined operations as well as OCL operations like type-casts such as `self->asType(OclAny)`, which also require several proofs done automatically by the HOL datatype package of Isabelle.

Table 6.1 describes the size of each of the above mentioned models together with the number of generated theorems and the time needed for importing them into HOL-OCL. The number of generated theorems depends linearly on the number of classes, attributes, associations, operations and OCL constraints. For generalizations, a quadratic number (with respect to the number of classes in the model) of casting definitions have to be generated and also a quadratic number of theorems have to be proven. The time for encoding the models depends on the number of theorems generated and also on the complexity on their complexity.

Notably, even the Royals and Loyals model can be imported in ca. 2 minutes, even though more than 2500 theorems are proven during import. We owe these quite reasonable times for model import mainly to our extensible universe construction, as described in Section 3.4. Recall that importing a user-supplied model already represents such an extension of the initial model, i. e., the HOL-OCL library. Without an extensible universe construction we would have to replay the proof scripts for large

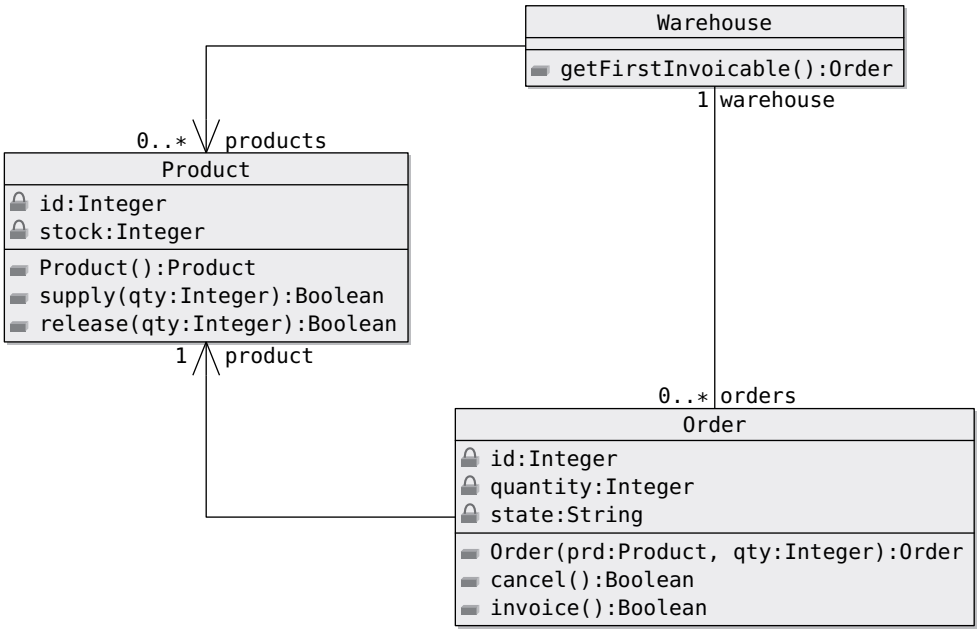


Figure 6.2: The Invoice case study models a simple system for invoicing orders; thus we need to model at least products, orders, and a warehouse managing the orders and products.

parts of the library. This is in our opinion of extra-ordinary value for practical work.

6.2.2 *Proving Properties of UML/OCL Models*

In the following, we use the Invoice model for showing some examples of how HOL-OCL can be used for formally proving properties of a UML/OCL model. These examples show that HOL-OCL can be used for analyzing models in general and in particular that it is a good starting point for the development for further machine-supported methodologies, like object-oriented refinement notions.

The main purpose of the Invoice system is to invoice orders, i. e., for a minimal system we need to model products, orders, and a warehouse. Figure 6.2 presents the UML data model of the Invoice system and Table 6.2 presents an excerpt of the OCL specification. For the complete informal and formal specification, see Appendix B.

As an example, we require that our specification fulfills at least the following requirements:

1. The postcondition of a constructor should imply the class invariant, i. e., the constructor creates a valid class fulfilling its invariant. Thus

```
context Product
  inv isNat: self.stock >= 0

context Product::Product():Product
  pre : true
  post: self.stock = 0 and self.oclIsNew()

context Product::supply(qty:Integer):Boolean
  pre: qty > 0
  post: self.stock = self@pre.stock + qty

context Product::release(qty:Integer):Boolean
  pre:  self.stock >= qty and qty > 0
  post: self.stock = self@pre.stock - qty
```

Table 6.2: OCL specification of the class `Product`. We mainly require, that the stock is non-negative and describe the basic behavior of operations for supplying and releasing products.

we prove for each constructor c (with postcondition post_c) of class C (with invariant inv_C) the following rule:

$$\frac{\tau \vDash \text{post}_C \ \mathit{self}}{\tau \vDash \text{inv}_C \ \mathit{self}} \ . \tag{6.1}$$

2. The class invariant and the precondition of an operation should be satisfiable in the same state, i. e., there exists at least one system state in which the operation can be called. This can be formalized for a class C (with invariant inv_C) and an operation m (with precondition pre_m) as follows:

$$\frac{}{\exists \ a_1 \cdots a_n \ \mathit{self} \ \tau. \ (\tau \vDash \text{inv}_C \ \mathit{self}) \wedge (\tau \vDash \text{pre}_m \ \mathit{self} \ a_1 \cdots a_n)} \ . \tag{6.2}$$

After loading the model into HOL-OCL, the first requirement for the class `Product` can be formulated as follows:

```
lemma "τ ⊨ Product_Boolean.post self result
       ⟹
       Product.inv self"
```

Where Product_Boolean.post is a logical constant describing the postcondition of the constructor `Product()` and Product.inv is a logical constant representing the invariant of the class `Product`. As a first step of our proof, we unfold these constants using the Isabelle simplifier:

```
apply(simp add: Product_post Product_inv)
```

resulting in a proof state representing the following proof obligation (using the OCL notation):

```
τ ⊨ self . stock ≐ 0 and self .oclIsNew()
⟹
τ ⊨ 0 <= self . stock
```

Applying the `safe` tactic,

```
apply(safe)
```

which is configured to use LTC (see Section 5.6.3) resolves the goal nearly. Note, except _ `and` _, all operations are strict. Moreover, _ `and` _ is only valid, if and only if, both arguments are `true`. Therefore, the assumption of this proof state already ensures the definedness of both *self* and *self* . stock. The remaining proof state looks as follows:

```
⟦τ ⊨ self . stock ≐ 0; τ ⊨ self .oclIsNew()⟧
⟹
τ ⊨ 0 <= 0
```

This obligation is easily proven by our OCL simplifier:

```
apply(ocl_simp)
done
```

Summarizing, we have formally proven that the postcondition of the constructor `Product::Product():Product` guarantees the invariant of the class `Product`.

As a second example, we prove that the conjunction of the precondition of the operation `Product::release(qty:Integer):Boolean` and the invariant of class `Product` is satisfiable, i. e., there is a system state in which the operation `release(qty:Integer):Boolean` can be called. We formalize this requirement as follows:

```
lemma
"∃ qty self τ.  (τ ⊨ Product. inv self)
              ∧(τ ⊨ release_Integer_Boolean. pre self qty)"
```

After unfolding the definitions using the Isabelle simplifier

```
apply(simp add: supply_pre Product_inv)
```

we get the following proof state:

```
∃qty self τ τ'. (τ, τ') ⊨ 0 <= self . stock
    ∧(τ, τ') ⊨ qty ≤ self . stock and 0 < qty
```

Using the existential introduction rule (exI) we construct witnesses for a satisfying state. In particular, we set the quantity *qty* to 1 and construct an instance of the class `Product` with no extension and the attributes `id` and `stock` store the value 1.

```
apply(rule_tac x="1" in exI)
apply(rule_tac x="λ(a, b). (⌞((OclAny_key. OclAny,
          oid :: oid), ⌞noext(Product_key. Product, ⌞1⌟, ⌞1⌟)⌟)⌟)"
  in exI)
  apply(rule exI)+
```

We can prove the resulting proof obligation using the following script:

```
apply(rule safe)
apply(simp_all add: Product.stock_def Product.stock_def
                    ss_lifting localValid2sem
                    Zero_ocl_int_def One_ocl_int_def OclStrongEq_def
                    OclLess_def OclLe_def )
done
```

Summarizing, we have formally proven that there exists at least one state that allows for the execution of the operation `release(qty:Integer)` of class `Product`.

6.3 DISCUSSION

In this chapter, we presented the system architecture of HOL-OCL and reported on some case studies. On the technical side, the most distinguishing feature of HOL-OCL is its use of Isabelle/HOL as a generic framework for tool development, instead of using it as a back-end tool only. On the theoretical side, our tool is based on a conservative shallow embedding. Using a generic interactive theorem prover as a framework for building new tools has several advantages: if done conservatively, the consistency of the underlying semantics can be guaranteed and the correctness with respect to this semantics is guaranteed by construction. Moreover, besides the obvious benefits like the reuse of rewriting and simplification algorithms we also get additional benefits like the reuse of user interfaces or the generation of proof documents. Overall, this shows the usability of our approach for building tools based on a machine-checked semantics.

Moreover, our experience shows that an extensible object store guarantees reasonable times for importing models. Already the first import of a model extends the existing base library which in our approach can be stored as a library of pre-compiled proof objects. There is no need for the time consuming task of replaying proof scripts for the base library while importing a user-defined model. Of course, this also allows for

the incremental loading of large models, which helps in analyzing large specifications.

However, the formal analysis HOL-OCL requires a fair amount of expertise in interactive theorem proving in general and Isabelle/HOL in particular. Thus, future extensions of the system should not only improve the degree of automation of the system but also provide support specialized support for analysis methodologies. For example, the presented consistency analysis could be supported by the fully automated generation of proof obligations and specialized tactics. Moreover, instead of describing a witnesses, i. e., a satisfying system state or scenario in a textual form, the corresponding object-diagram could also be part of the UML model. Extending our datatype-package with support for object diagrams would then allow for proving simple consistency proof obligations without user intervention. We will discuss this and other possible extension in more detail in Section 8.3.

7 RELATED WORK

In the previous chapters, we introduced a formal semantics for object-oriented data models and object-oriented constraint languages over these data models. We developed calculi for this for our framework and showed how this framework can be used to give a formal, machine-checked semantics for OCL. In this chapter, we will discuss related work which is as manifold as the list of topics discussed in this thesis: in Section 7.1 we discuss formal semantics for object-oriented systems in general. Further, we discuss the formal tool support for OCL (Section 7.5), formal semantics for OCL (Section 7.4), proof support for three-valued logics, and the embedding of object-oriented languages into theorem provers.

7.1 EMBEDDINGS OF OBJECT-ORIENTED LANGUAGES

Embedding languages into theorem provers has a long history [3, 18]. The technique was originally developed in the context of embedding hardware description languages (e. g., Boulton et al. [18] compare the embedding of three different hardware description languages). Nowadays, logical embeddings into theorem provers are a widely used technique for both reasoning about the embedded language itself and reasoning about specifications written in the object-language.

7.1.1 *Deep Embeddings of Object-oriented Languages*

There is various work based on a deep embedding of a Java-like language. Among these works are paper-and-pencil formalizations like [39, 41], but also many machine-checked semantics like [84, 85, 110, 113, 115]. In a deep embedding of a language semantics, syntax and types are represented by free datatypes. As a consequence, derived calculi inherit a heavy syntactic bias in form of side-conditions over binding and typing issues. This is unavoidable if one is interested in meta-theoretic properties such as type-safety, which all of the above mentioned works are aiming at. However, when reasoning about applications and not about language properties, this advantage turns into a major obstacle for efficient deduction. Thus,

while various proofs for type-safety [39, 41, 85, 113], soundness of Hoare calculi [114], and even soundness of verification condition generators [83, 105] have been provided, none of the mentioned deep embeddings has been used for substantial proof work in applications.

7.1.2 *Shallow Embeddings of Object-oriented Languages*

Using a shallow embedding for encoding an object-oriented language is still challenging. There are several encodings of classes as records, e. g., Aredo [4] presents such an encoding for PVS [93]. But this simple interpretation of classes as records does not provide support of object-oriented concepts like subtyping or inheritance.

Although there are several works on object-oriented semantics based on deep embeddings, there are only a few for shallow embeddings: Smith et al. [107] (a direct follow-up of Santen [104]) and Yatake et al. [118]. Moreover, there are shallow embeddings of a Java-like memory model by Jacobs and Poll [61] and Meyer and Poetzsch-Heffter [79].

The approach of Smith et al. [107], however, puts emphasis on a universal type for the method table of a *class*. This results in local universes for input and output types of methods and the need for reasoning about class isomorphisms; as the authors admit, this "creates considerable formal overhead." For example, subtyping on *objects* must be expressed implicitly via refinement.

Yatake et al. [118] developed a conservative, shallow embedding of class models into the HOL system [46]. They also provide a tool that generates applications specific, i. e., depending on the class model, theories using a non-extensible encoding schema. Similar to our construction, the store model presented by Yatake et al. [118] provides cast operations directly on the object store. Also similar properties about the encoding of class models are proven during the import of a class model.

The underlying encoding used by the LOOP tool [61] and Jive [79] shares same basic ideas with respect to the object model. However, the overall construction is based on a closed-world assumption and thus not extensible. Although several papers, e. g., [61], model class invariants as co-inductive definitions, none of these ideas has been implemented in a tool.

With respect to extensibility of data-structures, the idea of using parametric polymorphism is well-known in HOL research communities; for example, extensible records and their application for some form of subtyping has been described in HOOL [81]. Since only one extension possibility is provided by the presented encoding, this results in a restricted form of inheritance; namely, type conversions are only possible if a class has at most one direct subclass. In our notion, the work of Naraschewski and Wenzel [81] provides only α-extensions whereas our encoding generalizes

this by providing α-extensions and β-extensions.

Thus, while the basic concepts in our approach of representing subtyping by the subsumption relation on polymorphic types are not new [81, 104], we extended these works by including concepts such as undefinedness, mutual recursion between object instances, dynamic types, recursive method invocation and extensible class hierarchies. In particular, the extensionality of our constructions allows for an efficient implementation of an object-oriented datatype package.

7.2 DATATYPE PACKAGES

All approaches presented in Section 7.1.2 have two details in common: they are not extensible and only supported by tools generating theory files (using the concrete syntax of the underlying theorem prover) which are imported into the theorem prover.

In contrast, our extensible encoding of object-oriented data structures allows for implementing a datatype package. Datatype packages have been considered mostly in the context of HOL or functional programming languages. Going back to ideas of Milner, systems like [15, 77] build over a S-expression like term universe (co)-inductive sets which are abstracted to (freely generated) datatypes. The inductive package presented by Paulson [94] also uses subsets of the ZF set universe i. Overall, we extend this work into a generic datatype package for object-oriented data-structures. The underlying extensible encoding allows even the incremental import of object-oriented models.

Huffman et al. [58] suggested a universe construction based on Scott's reflexive domains. This work does not present a datatype package. It is merely a library construction geared towards functional programming languages like Haskell and not towards object-oriented programming languages.

7.3 PROOF SUPPORT FOR THREE-VALUED LOGICS

The construction of specialized decision and semi-decision procedures for many-valued logics such as Strong Kleene Logic has been investigated before. Most of this work is based on semantic tableaux methods. Examples for such works are Kerber and Kohlhase [63] and Beckert et al. [13]. The development of the tableaux-based proof-procedure (for two-valued logics) in Isabelle has been deeply influenced by the leanT^AP [11] algorithm. Interestingly, leanT^AP itself is just the "bare bones" version of its ancestor ${}_3T^AP$ [13] which was developed especially for Strong Kleene Logic. Thus, in some sense our work re-introduces three-valued reasoning into an implementation of leanT^AP. However, one of our design goals

is to provide suitably abstract calculi for Strong Kleene Logic that can be processed in a *generic* prover engine, even one that is optimized for two-valued reasoning.

7.4 FORMAL OCL SEMANTICS

The widely and successful use of UML in industry attracted many researchers to look at UML in general and OCL in particular. The quite informal nature of earlier versions of the OCL standard [87] stimulated a large variety of research on OCL. For example, there are various works [14, 31–33, 37, 53, 54, 68, 76, 100, 101] that either propose a formal semantics of OCL, or discuss semantic problems thereof. Most of them are based on the informal description given in the OCL 1.x standard. Most notable of these earlier works is the work of Richters [100] which also builds the basis for the formal semantics given in recent version of the OCL standard, i. e., [88, Appendix A].

Several of these works simplify the semantics of OCL drastically: even though the very first version OCL standard [87] already introduces a Strong Kleene Logic, several approaches, e. g., [14, 68] base their semantics on a classical two-valued one. There are mainly two motivations of using a two-valued logic: first, there are a variety of tools available for classical logics that can be reused and second, Hähnle [50] argues that at least for specifying structured software systems approaches based on a two-valued logic based on the underspecification of total functions are superior. The latter argumentation motivated the use of a two-valued logic for the KeY tool. The underlying translation of the OCL syntax into a first-order dynamic logic is described by Beckert et al. [14]. Motivated by the available tools for two-valued logics, Kyas [68] provides a direct translation of the OCL syntax into HOL as defined by PVS system. Of course, a direct mapping can neither be sound nor complete with respect to a semantics based on a Strong Kleene Logic.

Previous semantic definitions of OCL version 2.0, i. e., [88, Appendix A], are based on mathematical notation in the style of naïve set theory, which is in our view inadequate to cover subtle subjects of object-orientation. In particular, we criticize the use of naïve set theory for introducing the notions of type, state, and model. For example, types were explained by a type interpretation function $I(t)$ [88, introduced in Definition A.14] mapping to a (never described) universe of values and objects. The expression interpretation function $I[\![E]\!]$ assumes that variables and key operator symbols have been annotated with type expressions like in $_ =_t _$. Therefore typing is a prerequisite of the semantic construction of OCL. $I[\![E]\!]$ uses these type annotations to project and inject into subsets of the universe described by $I(t)$ without proof or argument that these definitions actually respect the typing. Also, the standard does not specify what correct typ-

ings are (the semantic function is defined for arbitrary typings), whether they are unique, and how to derive them. Using for our embedding a typed semantic domain resolves this problems automatically.

Recently, there seems to be a trend to define the semantics of OCL using OCL itself; either directly by using QVT and OCL by Marković and Baar [76] or indirectly by using sequence diagrams by Chiaradía and Pons [33]. Whereas the authors of [76] are aware of the circularity they are introducing and argue that the use a specific semantics given by a proprietary tool solves this problem, the authors of [33] are not aware of the problem. From our point of view, both approaches are not adequate for defining a semantics.

7.5 TOOL SUPPORT

Since OCL was introduced, many tools supporting OCL in one way or the other were developed. Toval et al. [111] present a comparison of tools supporting OCL. The tools they present in this comparison can be classified into three categories: type-checker, runtime constraint checkers, and execution environments that allow for the simulation and validation of models. Examples for the latter category, which is most closely related to the work presented in this thesis, are the USE tool [102] and OCLE [34]:

- The UML Specification Environment (USE) [102] allows the animation of UML/OCL specifications. A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in OCL are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation.

- The Object Constraint Language Environment (OCLE) [34] is a CASE tool with first-class support for OCL. In particular, OCLE allows for the interactive evaluation of OCL expressions and thus checking well-formedness rules of a UML specification.

Another, in our view important, category of OCL tools are proof environments for OCL; besides the work presented in this thesis, there are only two other proof environments supporting OCL, namely the KeY [1] tool and OCLVP [68]. Moreover, there are several proof environments for JML (e. g., Jive and LOOP) and Spec# (e. g., Boogie), which we include in the following discussion:

- The KeY tool [1, 10] is an integrated formal specification and verification environment for specifications consisting of Hoare-style annotations of Java programs. In contrast to HOL-OCL, the semantics

	HOL-OCL	KeY	OCLVP	Boogie	Jive	LOOP
object model	UML	Java	UML	C#	Java	Java
subtyping	multiple	multiple	multiple	multiple	multiple	multiple
inheritance	single	single	single	single	single	single
extensible	yes	no	no	yes	no	no
conservative	yes	no	no	no	yes	no
embedding	shallow	shallow	deep	pre-compilation	shallow	pre-compilation
constraint language	OCL 2.0	dynamic logic[1]	HOL[2]	Spec#	JML	JML
conservative	yes	no	yes	no	yes	no
invariants	semantic/structural	structural	structural	structural	structral	manual
embedding	shallow	pre-compilation	shallow	pre-compilation	shallow	shallow
datatype package	yes	no	no	no	no	no
meta-logic	HOL	dynamic Logic	HOL	specialized	HOL	HOL

[1] The KeY tool provides a frontend for using OCL 1.x as concrete input syntax.
[2] The OCLVP tool provides a frontend for using OCL 2.x as concrete input syntax.

Table 7.1: Comparing proof environments for object-oriented constraint languages.

used in the KeY tool is not compliant to OCL standard (nevertheless, it supports OCL 1.x as concrete input syntax). In particular, the KeY tool is based on a direct mapping of OCL syntax to a first-order dynamic logic [14]. The dynamic-logic used is two-valued, i. e., it does not support undefinedness within the logic. Moreover, it does not attempt to build up the theory of its constraint language by definitional axioms and thus has not formally investigated the issue of consistency.

- The Object Constraint Language Verification Platform (OCLVP) [68, 69] provides a formalization of UML class diagrams, state charts and OCL using HOL as provided by PVS [93]. The OCLVP tool directly translates OCL formulae into HOL formulae using a direct mapping, i. e., deliberately ignoring the Strong Kleene Logic of OCL. Neither is the underlying embedding of class diagrams extensible, nor does the tool provide support for invariants, preconditions and postconditions. OCL formulae are directly mapped to HOL formulae and operations to HOL functions.

- Boogie [6, 72] is a compile-time assertion checker for Spec#. We classify Boogie as a pre-compilation tool, i. e., a Spec# program is compiled into a standard imperative program which is used as input of a verification condition generator. The generated verification conditions are handled in an automatic theorem prover. While this architecture provides powerful tools that can handle large inputs, the theoretical foundation is problematic. For example, the compilation itself is not verified, and it is not clear if the generated conditions are sound. Moreover, the overall approach depends on the degree of automation that can be achieved by the underlying (automatic) theorem prover.

- The Jive [79] tool provides an environment for doing Hoare-style verification of Java programs. The underlying encoding of object structures is based on a shallow embedding of the Java memory model into IsabelleHOL. The overall construction is based on a closed-world assumption and several properties, e. g., some aspects of subtyping, are handled on the level of the Hoare-logic instead of the object store. Moreover, due to the closed-world model, Jive cannot provide an extensible data package, thus, for every analysis, large portion of the core systems must be re-proven. The same criticism is also valid for the LOOP tool [61].

Table 7.1 shows a summary of this comparison. Notably, Boogie and HOL-OCL are the only tools that are based on an extensible construction of the underlying data-store. In our experience a key feature needed for

providing an efficient and scalable implementation. HOL-OCL is the only environment providing semantical invariants, including their support in an object-oriented datatype package. Also, it is the only proof environment based on an OCL semantics conforming to the standard [88], e. g., supporting a three-valued logic.

Most of these tools provide a wealth of additional features that are not covered here. We deliberately restricted the comparison to the main topics of this thesis: the theorem prover component and the underlying embeddings of class models and constraint languages. For example, we exclude code-generation, code-verification, or proof-animation techniques that are also provided by several of these environments from this comparison.

CONCLUSION AND FUTURE WORK

8

In this chapter, we draw conclusions, summarize the contributions of this thesis and give an outlook on future work.

8.1 CONCLUSIONS

This thesis shows that a shallow embedding of an object-oriented specification language into HOL is possible and can serve several purposes: First, it provides a consistent framework for examining language features. Second, it can provide a machine-checked semantics for an industrial defined specification language. Third, it provides the basis for formal tools that, if implemented conservatively on top of a theorem prover environment, are guaranteed to be correct with respect to their formalization. And fourth, it enables the formal analysis of object-oriented specifications.

A flexible formal framework for object-orientation can be used to examine language features and discuss extensions of an existing language. This is not only useful during the development and standardization of a specification language but also for tool implementers providing tools for a subset or a semantic variation of a language. If a formal framework for a specification language is based on a conservative embedding, the consistency of all these variants can be guaranteed without the need of additional proofs. Inconsistencies introduced by extending the language are recognized immediately. In our view, this is an important prerequisite for providing different semantical extensions in a consistent way. For example, this also provides a formal understanding of the concept of "semantic deviation points" as introduced in the UML standard [91].

A standard containing a machine-checked semantics cannot only guarantee interoperability between different tools on the syntactical level but also on the semantical level. Especially the latter allows for the exchange of specifications between different tools without changing the interpretation of a specification. Thereby it does not matter if these tools are formal, like theorem provers or model-checkers or semi-formal like code-generators; there will always be a strong semantical link between the results of ap-

plying different tools on the same input. Overall, this brings MDE to a semantical level.

Embedding a specification language in a shallow and conservative way into a theorem prover results directly in a proof environment for that language. Thus, such an embedding is both a machine-checked semantics and the source code of an implementation. Overall, this method for defining semantics and building tools guarantees the consistency of the semantics and the correctness of the implementation.

Today formal methods are mainly used together with a software development process using a procedural programming language like C or functional programming languages like Haskell. An interactive theorem prover for object-oriented specifications enables the use of a formal analysis together with an object-oriented development process and object-oriented programming languages. For example, this also allows for the use of object-oriented specification for certifying systems with respect to EAL7 ("Formally Verified Design and Tested") of the Common Criteria international standard [36].

8.2 SUMMARY OF CONTRIBUTIONS

In this section, we summarize the most important contributions of this thesis.

In Chapter 3, we introduced a framework for object-oriented specifications. This framework is presented as a conservative, shallow embedding into Isabelle/HOL. It comprises an object store, i. e., a formalization of an object-oriented data structures and an object-oriented constraint language.

As a novel feature, our encoding of object-oriented data structures is extensible. This extensibility is a basis for the implementation of efficient tools based on this encoding. Moreover, it allows for the incremental formal analysis of specifications, i. e., the underlying data model can be extended without the time-consuming task of replaying proof scripts. As already the import of a user-supplied specification represents an extension of the base library containing several thousands of theorems, the extensionality is a cornerstone for building formal tools for object-oriented systems.

Our object-oriented constraint language can cope with undefinedness, e. g., introduced by path expressions that are not valid within a concrete system state. Moreover, we discuss several "semantic deviation points" for object-oriented constraint languages. Thus we provide a framework and a tool for a large variety of object-oriented constraint languages. The semantics of a concrete constraint language can be defined by selecting a specific subset of these semantical building blocks.

Both embeddings are structured using conservative theory morphisms using semantic combinators. This technique allows for the automatic derivation of theorems based on already proven theorems over the meta-logic possible. Thus, structuring an embedding using semantic combinators makes a conservative embedding technique for a real-world language feasible.

The modular way our semantics is organized allows for extending our framework in various ways. For example, by introducing temporal aspects or general recursion. The former may be used to give a formal semantics also to the other UML diagrams such as state diagrams or sequence diagrams, while the latter may provide the basis for to the development of powerful and executable libraries within our framework.

In Chapter 4, we presented a formal, machine-checked semantics of OCL based on the framework we presented in Chapter 3. Our formal OCL semantics is compliant to the OCL 2.0 standard [88]. In particular, we provide, for the first time, formal proofs showing that a formal semantics conforms to the normative requirements of the standard. Moreover, we also show, that our semantics is a machine-checked formalization of the informative semantics given in the OCL standard [88, Appendix A]. Thus, we provide the missing link between the formal semantics in the informative part [88, Appendix A] (based on the work of Richters [100]) of the standard and the normative part of the standard.

In Chapter 5, we derived several calculi and proof techniques for our framework, i. e., for an object-oriented constraint language. Since deriving means that we proved all rules with an interactive theorem prover, we can guarantee both the consistency of the semantics as well as the soundness of the calculi. Moreover, we developed automatic proof support for the derived calculi UEC, LEC, and LTC. In particular, the calculi led to rewriting and tableau-based decision procedures for certain fragments of our object-oriented constraint language. The development of HOL-OCL itself and also the case studies we carried out indicates that for predominantly strict language, there is sufficiently high potential for a efficient deduction in general and in particular that tools for effective reasoning over such language can be built on top of generic theorem prover environments such as Isabelle. Thus, we provided the basis for deduction-based OCL tools.

In Chapter 6, we presented an architecture for building formal tools that are based on a theorem prover like Isabelle/HOL. Based on our framework, we used this architecture for building HOL-OCL, an interactive theorem prover for UML/OCL. Moreover, we show the potential of using such tools by carrying out some case studies. Thus, we have provided a solid basis for turning object-oriented modeling in UML/OCL into a formal method. Moreover, our case studies showed that an extensible universe construction provides a reasonable fast import of user-defined models.

8.3 FUTURE WORK

In this section, we discuss some directions of further work. In particular, we discuss theoretical and technical extension of our framework to improve usability and to open new areas of research and application.

8.3.1 *Extending our Formal Framework*

IMPROVING PROOF SUPPORT. While our existing proof procedures for OCL are quite satisfactory, more work has to be done to increase efficiency and to cover larger fragments of the language (e. g., automated procedures for arithmetic). Moreover, the integration of special techniques for multi-valued logics, e. g., based on Hähnle [48] and the integration of external tools, i. e., model checkers, should improve the efficiency of our semi-automated techniques for three-valued logics.

SUPPORT FOR BEHAVIORAL SPECIFICATIONS. The UML offers several diagram types for specifying the behavior of a system, e. g., by using state diagrams (a variant of state machines). An embedding of state diagrams into our framework, e. g., supporting OCL formulae as guards, would allow for the combination of behavioral and data-oriented specifications within one consistent formal framework. As the semantics of state diagrams in the UML standard is not precise, a formal semantics for state diagrams matching the intention of the standard has to be developed and integrated into our framework.

SUPPORT FOR ANALYSIS TECHNIQUES. The development of various techniques known from formal methods, like refinement or retrenchment [12] need to be adopted to the three-valued setting of UML/OCL. In particular, such methodologies should be supported by HOL-OCL itself, e. g., by generating, and if possible, resolving, the corresponding proof obligations automatically.

SUPPORT FOR DOMAIN-SPECIFIC ANALYSIS METHODS. Developing domain-specific extensions for our framework, i. e., extending the embedding and developing proof support thereof, is another interesting area for future research. For example, SecureUML [9] is a UML dialect that allows for specifying role-based access control within UML/OCL specifications. Extending our framework to support directly reasoning over SecureUML specifications is a rewarding task. This would include the development of a formal, machine checked semantics for SecureUML and the development of specialized proof support for access control specifications. Alternatively, one could use model-transformations [29] for converting a SecureUML model into a pure UML/OCL model. Whereas such

a transformation approach does not need an embedding of SecureUML into HOL-OCL, the resulting representation is less abstract which results in more complicated proofs.

8.3.2 *Developing a Formal Tool Chain*

Aiming for the broader acceptance of formal methods a deep integration into a tool-supported software development process, e. g., based on the MDA or MDE approach, is desirable. Besides a deeper integration into common CASE tools (e. g., in a similar way as the integration of the KeY tool [30]), we especially focus on bringing verification, model-based testing and code-generation closer together. In particular, such a tool chain [27] could include the integration of specification-based testing techniques, e. g., based on HOL-TestGen [22]. This would allow for generating test cases from the same specification the formal analysis is done. Therefore, testing can be used to validate that the implementation, which contains user-implemented parts, is in fact a refinement of the formal specification.

8.3.3 *Applications.*

Besides larger case studies, e. g., consistency analysis of specifications or formal analysis in the area of secure and safe system development, we see a great potential for a formal refinement calculus for OCL. Such a refinement calculus would allow for using HOL-OCL in a consistent way over several stages of a formally supported software development cycle and is in our opinion a cornerstone for applying formal methods successfully.

Moreover, combining HOL-OCL with embeddings of programming languages like IMP++ [23] or μJava [85] allows for integrated formal reasoning over specifications and code.

THE SYNTAX OF OCL

OCL, being advertised with the slogan "Mathematical Foundation, But No Mathematical Symbols" [116], is normally written using a concrete syntax that is inspired by object-oriented programming languages. Whereas this textual notation pleases the people coming from object-oriented programming languages, it looks awkward for people coming from the mathematics and formal methods field. Especially for proof work, there seems a need for a compact, mathematical notation. Thus we developed a mathematics-oriented OCL syntax, as an alternative to the programming-language like notation used in the OCL 2.0 standard. For example, compare the textual presentation of the proof rule:

$$\frac{\tau \vDash S\texttt{->includes}(x) \quad \tau \vDash \texttt{not}\,(P\,x) \quad \mathrm{cp}\,P}{\tau \vDash S\texttt{->forall}(x \mid P(x))\texttt{.IsDefined()}} \tag{A.1}$$

to its presentation in mathematical notation:

$$\frac{\tau \vDash x \in S \quad \tau \vDash \neg(P\,x) \quad \mathrm{cp}\,P}{\tau \vDash \partial\big(\forall\, x \in S.\ P(x)\big)}\,. \tag{A.2}$$

Clearly, both the concrete syntax of the standard and our mathematical syntax, have their advantages and disadvantages and therefore we support both of them in HOL-OCL. A user of HOL-OCL can mix both syntaxis arbitrarily, i. e., the user is free to choose the syntax he or she likes best. Moreover, the system can also be configured to check that only one of the syntaxis is used in a consistent manner or convert terms from one syntax to the other, e. g., for presentation purposes.

In Table A.1 we provide a brief comparison between the different concrete syntaxis for OCL, namely the syntax as proposed in the OCL standard, our textual notation that tries to follow the standard syntax as close as possible, and finally our new mathematical syntax. The table follows the OCL library presentation from the standard [88, Chapter 11], constructs that are not supported by HOL-OCL are written in a gray typeface, e. g., `o.oclIsInState(s)`. Extension to the OCL standard are written in a green typeface, e. g., `x sand y`.

Table A.1: Comparison of different concrete syntax variants for OCL

	OCL (standard)	mathematical HOL-OCL
OclAny	`x = y`	$x \doteq y$
	`x <> y`	$x \not\doteq y$
	`x := y`	$x \mathrel{\hat{=}} y$
	`x ~ y`	$x \mathrel{\hat{\simeq}} y$
	`x :~ y`	$x \mathrel{\overset{\triangle}{\simeq}} y$
		$x \mathrel{\hat{\hat{\simeq}}} y$
		$x \mathrel{\overset{\triangle}{\hat{\simeq}}} y$
	`o.oclIsNew()`	o`.oclIsNew()`
	`o.oclIsUndefined()`	$\not\partial\, o$
	`o.oclAsType(t)`	o `.oclAsType` (t)
	`o.oclIsType(t)`	o `.oclIsTypeOf` (t)
	`o.oclIsKindOf(t)`	o `.oclIsKindOf` (t)
	`o.oclIsInState(s)`	
	`t::allInstances()`	t`::allInstances()`
	`t::typeSetOf()`	t`::typeSetOf()`
	`t::kindSetOf()`	t`::kindSetOf()`
OclMessage	`o.hasReturned()`	
	`o.result()`	
	`o.isSignalSent()`	
	`o.isOperationCall()`	
OclVoid	`OclUndefined`	$\bot$
	`o.oclIsUndefined()`	$\not\partial\, o$
	`o.oclIsDefined()`	$\partial\, o$
Real	`x + y`	$x + y$
	`x - y`	$x - y$
	`x * y`	$x \cdot y$
	`-x`	$-x$
	`x / y`	$x \,/\, y$
	`x.abs()`	$\lvert x \rvert$
	`x.floor()`	$\lfloor x \rfloor$
	`x.round()`	$\lceil x \rceil$
	`x.max(y)`	$\max(x, y)$
	`x.min(y)`	$\min(x, y)$
	`x < y`	$x < y$
	`x > y`	$x > y$
	`x <= y`	$x \leq y$
	`x >= y`	$x \geq y$
Integer	`x - y`	$x - y$
	`x + y`	$x + y$
	`x * y`	$x \cdot y$
	`x / y`	$x \,/\, y$

Continued on next page

	OCL	mathematical HOL-OCL
Integer	`-x`	$-x$
	`x.abs()`	$\lvert x \rvert$
	`x.div(y)`	$x \operatorname{div} y$
	`x.mod(y)`	$x \bmod y$
	`x.max(y)`	$\max(x, y)$
	`x.min(y)`	$\min(x, y)$
String	`s.size()`	$\lVert s \rVert$
	`s.concat(z)`	$s \frown z$
	`s.substring(i,j)`	$s\,.\texttt{substring}(i, j)$
	`s.toInteger()`	$s\,.\texttt{toInteger()}$
	`s.toReal()`	$s\,.\texttt{toReal()}$
	`s.toUpper()`	$s\,.\texttt{toUpper()}$
	`s.toLower()`	$s\,.\texttt{toLowert()}$
Boolean	`|= p`	$\models p$
	`t |= p`	$\tau \models p$
	`true`	$\mathtt{t}$
	`false`	$\mathtt{f}$
	`x or y`	$x \vee y$
	`x xor y`	$x \oplus y$
	`x and y`	$x \wedge y$
	`not x`	$\neg x$
	`x implies y`	$x \longrightarrow y$
		$x \overset{1}{\longrightarrow} y$
		$x \overset{2}{\longrightarrow} y$
	`x sor y`	$x \dot{\vee} y$
	`x sxor y`	$x \dot{\oplus} y$
	`x sand y`	$x \dot{\wedge} y$
	`x simplies y`	$x \dot{\longrightarrow} y$
	`if c then x else y endif`	$\mathtt{if}\ c\ \mathtt{then}\ x\ \mathtt{else}\ y\ \mathtt{endif}$
Collection	`X->size()`	$\lVert X \rVert$
	`X->includes(y)`	$y \in X$
	`X->excludes(y)`	$y \notin X$
	`X->count(y)`	$X\,\texttt{->count}(y)$
	`X->includesAll(Y)`	$X \subseteq Y$
	`X->excludesAll(Y)`	$X \not\subseteq Y$
	`X->isEmpty()`	$\varnothing \doteq X$
	`X->notEmpty()`	$\varnothing \neq X$
	`X->sum()`	$X\,\texttt{->sum()}$
	`X->product(Y)`	$X \times Y$
	`X->exists(e:T|P(e))`	$\exists\, e \in X.\ P(e)$
	`X->forAll(e:T|P(e))`	$\forall\, e \in X.\ P(e)$
	`X->isUnique(e:T|P(e))`	$X\,\texttt{->isUnique}\,(e : T \mid P(e))$
	`X->any(e:T|P(e))`	$X\,\texttt{->any}(e : T \mid P(e))$

Continued on next page

	OCL	mathematical HOL-OCL
Set	`X->one(e:T|P(e))`	X->one$(e : T \mid P(e))$
	`X->collect(e:T|P(e))`	$\{\!\mid e \in X \mid P(e) \mid\!\}$
	`Set{}`	$\varnothing$
	`X->union(Y)`	$X \cup Y$
	`X = Y`	$X \doteq Y$
	`X->intersection(Y)`	$X \cap Y$
	`X->complement(Y)`	X^{-1}
	`X - Y`	$X - Y$
	`X->including(y)`	insert $y\ X$
	`X->excluding(y)`	y->excluding(X)
	`X->symmetricDifference(Y)`	$X \ominus Y$
	`X->count(y)`	X->count(y)
	`X->flatten()`	$[\![X]\!]$
	`X->asSet()`	X->asSet()
	`X->asOrderedSet()`	X->asOrderedSet()
	`X->asSequence()`	X->asSequence()
	`X->asBag()`	X->asBag()
	`X->select(e:T|P(e))`	$(\!\mid e \in X \mid P(e) \mid\!)$
	`X->reject(e:T|P(e))`	$)\!\mid e \in X \mid P(e) \mid\!($
	`X->collectNested(e:T|P(e))`	$\{\{\!\mid e \in X \mid P(e) \mid\!\}\}$
	`X->sortedBy(e:T|P(e))`	X->sortedBy $(e : T \mid P(e))$
	`X->iterate(x; r=c| P(x, r))`	X->iterate$(x;\ r{=}c \mid P(x, r))$
OrderedSet	`OrderedSet{}`	$\langle\rangle$
	`X = Y`	$X \doteq Y$
	`X->append(y)`	X @ y
	`X->prepend(y)`	y # X
	`X->insertAt(i,y)`	X->insertAt(i, y)
	`X->subOrderedSet(i,j)`	X->subOrderedSet(i, j)
	`X->at(i)`	$\natural\, i\ X$
	`X->indexOf(y)`	$X \natural?\ y$
	`X->first()`	$\natural\, 1\, X$
	`X->last()`	$\natural\, \$\, X$
Bag	`Bag{}`	$\wr\wr$
	`X = Y`	$X \doteq Y$
	`X->union(Y)`	$X \cup Y$
	`X->intersection(Y)`	X^{-1}
	`X->including(y)`	insert $y\ X$
	`X->excluding(y)`	y->excluding(X)
	`X->count(y)`	X->count(y)
	`X->flatten()`	$[\![X]\!]$
	`X->asBag()`	X->asBag()
	`X->asSequence()`	X->asSequence()
	`X->asSet()`	X->asSet()

Continued on next page

	OCL	mathematical HOL-OCL
	`X->asOrderedSet()`	X`->asOrderedSet()`
	`X->select(e:T|P(e))`	$(\!\vert e \in X \mid P(e) \vert\!)$
	`X->reject(e:T|P(e))`	$)\!\vert e \in X \mid P(e) \vert\!($
	`X->collectNested(e:T|P(e))`	$\{\{\vert e \in X \mid P(e) \vert\}\}$
	`X->sortedBy(e:T|P(e))`	X`->sortedBy`$(e : T \mid P(e))$
	`X->iterate(x; r=c| P(x, r))`	X`->iterate`$(x; r{=}c \mid P(x,r))$
Sequence	`Sequence{}`	$[]$
	`X->count()`	X`->count`(y)
	`X = Y`	$X \doteq Y$
	`X->union(Y)`	$X \cup Y$
	`X->flatten()`	$[\![X]\!]$
	`X->append(y)`	$X \mathbin{@} y$
	`X->prepend(y)`	$y \mathbin{\#} X$
	`X->insertAt(i,y)`	X`->insertAt`(i, y)
	`X->subSequence(i,j)`	X`->subSequence`(i, j)
	`X->at(i)`	$\natural\, i\, X$
	`X->indexOf(y)`	$X \natural?\ y$
	`X->first()`	$\natural\, 1\, X$
	`X->last()`	$\natural\, \$\, X$
	`X->including(y)`	insert $y\, X$
	`X->excluding(y)`	y`->excluding`(X)
	`X->asBag()`	X`->asBag()`
	`X->asSequence()`	X`->asSequence()`
	`X->asSet()`	X`->asSet()`
	`X->asOrderedSet()`	X`->asOrderedSet()`
	`X->select(e:T|P(e))`	$(\!\vert e \in X \mid P(e) \vert\!)$
	`X->reject(e:T|P(e))`	$)\!\vert e \in X \mid P(e) \vert\!($
	`X->collectNested(e:T|P(e))`	$\{\{\vert e \in X \mid P(e) \vert\}\}$
	`X->sortedBy(e:T|P(e))`	X`->sortedBy`$(e : T \mid P(e))$
	`X->iterate(x; r=c| P(x, r))`	X`->iterate`$(x; r{=}c \mid P(x,r))$
	`let e=x in P(s) end`	`let` $e = x$ `in` $P(s)$ `end`

THE INVOICE SYSTEM

B

In this section, we present a well-known case study for comparing specification formalisms, e. g., Frappier and Habrias [42] give an overview of several formalization of this case study using different formalisms.

B.1 INFORMAL DESCRIPTION

Frappier and Habrias [42] describe the invoice system informally as follows:

1. The subject is to invoice orders.
2. To invoice is to change the state of an order (to change it from the state "pending" to "invoiced").
3. On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders.
4. The same reference can be ordered on several different orders.
5. The state of the order will be changed into "invoiced" if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product.
6. You have to consider the following two cases:
   a) Case 1:
   All the ordered references are references in the stock. The stock or the set of orders may vary:
      - due to the entry of new orders or canceled orders;
      - due to having a new entry of quantities of products in stock at the warehouse.

But, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

b) Case 2:
You do have to take into account the entries of

- new orders;
- cancellations of orders;
- entries of quantities in the stock.

B.2 FORMAL SPECIFICATION

In this section, we present a formalization of the Invoice case-study using UML/OCL. Dupuy et al. [40] already present a UML specification for the invoice system. But this specifications lacks any usage of OCL. Moreover, the use of UML is quite informal, e. g., their specification is untyped. Our work is inspired by the formalization of Dupuy et al. [40], in fact, we restrict ourselves to making their specification more rigid. For example, we provide full type annotation and complete the diagrammatic part of UML with a detailed OCL specification.

Figure B.1 shows the data model of our case study is quite simple. For realizing item 6a we only need the classes `Product` and `Order`. For realizing the item 6b, we also model a class `Warehouse`. Figure 6.2 presents the UML data model of the Invoice system. Table B.1 presents the OCL specification for constraining the state part of the system, i. e., constraining the datatypes. For example, UML/OCL does not provide a datatype for natural numbers, therefore we use the datatype `Integer` and constrain the corresponding attributes to positive values. Table B.2 describes the behavior of the Invoice case-study, i. e., the precondition and postconditions of the operations. Overall, this completes the OCL specification.

Finally, Table B.3 presents the complete theory file containing the proofs explained in Chapter 6.

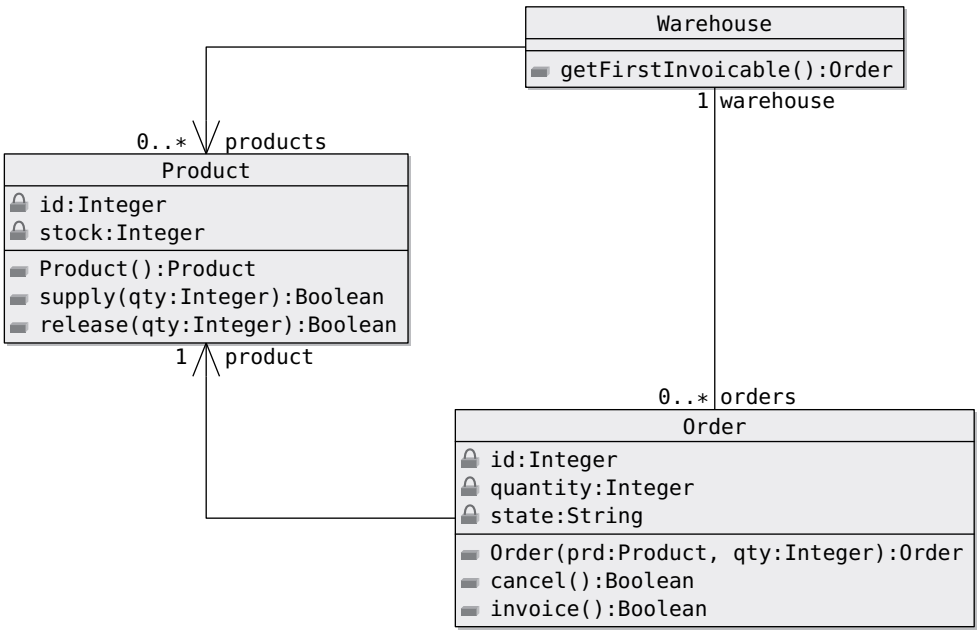


Figure B.1: The Invoice Case-study models a simple system for invoicing orders; thus we need to model at least products, orders, and a warehouse managing the orders and products.

```
-- The stock of a Product is always a natural number, i.e., it is a
-- positive Integer. This also ensures the definedness of the stock.
context Product
  inv isNat: self.stock >= 0

-- The Product id is unique.
context Product
  inv idUnique: self.allInstances()
                  ->forAll(p1:Product, p2:Product | p1.id <> p2.id)

-- The quantity of an Order is always a natural number, i.e., it is
-- a positive Integer. This also ensures the definedness of the
-- quantity.
context Order
  inv isNat: self.quantity >= 0

-- The state of an Order should either be 'pending'  or 'invoiced'.
-- As a direct support for enumeration is not well developed in most
-- CASE tools, we use a String and constrain it to the two
-- alternatives using an invariant.
context Order
  inv stateRange:    (self.state = 'pending')
                  or (self.state = 'invoiced')

-- The Order id is unique.
context Order
  inv idUnique: self.allInstances()
                  ->forAll(o1:Order, o2:Order | o1.id <> o2.id)

-- There is one and only one Warehouse.
context Warehouse
  inv isStatic: self.allInstances()->size() = 1
```

Table B.1: Constraining the data specification of the Invoice case-study.

```
-- Initialize the state of an Order
context Order::state : String
  init: 'pending'

-- Create a new Order
context Order::Order(prd:Product,qty:Integer):void
  pre: qty > 0
  pre: self.warehouse.products->exists(x:Product | x = prd)
  pre: not prd.oclIsUndefined()
  post: self.oclIsNew() and self.quantity = qty and self.orderedProduct = prd

-- The state of the order will be changed into "invoiced" if the ordered quantity
-- is either less or equal to the quantity which is in stock according to the
-- reference of the ordered product.
context Order::invoice() : void
  pre: self.state = 'pending'
       and self.quantity <= self.orderedProduct.stock
  post: self.state = 'invoiced' and self.quantity = self.quantity@pre
        and self.orderedProduct = self.orderedProduct@pre
        and self.orderedProduct.stock = self.orderedProduct@pre.stock - self.quantity

-- Cancel order as an opposite operation to invoice order
context Order::cancel() : void
  pre:  self.state = 'invoiced'
  post: self.state = 'pending'
        and self.quantity = self.quantity@pre and self.product = self.product@pre
        and self.product.stock = self.product@pre.stock + self@pre.quantity

-- Create a new Order
context Product::Product():void
  pre : true
  post: self.stock = 0 and self.oclIsNew()

-- Add quantity of the product to the stock
context Product::supply(qty:Integer):void
  pre: qty > 0
  post: self.stock = self.stock@pre + qty

-- Remove quantity of the product from the stock
context Product::release(qty:Integer):void
  pre:  self.stock >= qty
  post: self.stock = self.stock@pre - qty

-- Warehouse management
context Warehouse::getFirstInvoicable():Order
  pre: self.orders->exists(x:Order |
              x.state = 'pending' and x.quantity <= x.orderedProduct.stock)
  body: self.orders->any(x:Order |
              x.state = 'pending' and x.quantity <= x.orderedProduct.stock)
```

Table B.2: Specifying the behavior of the Invoice case-study.

```
theory invoices
imports
  OCL
begin

import_model "invoices.xmi" "invoices.ocl"

lemma "τ ⊨ Product_Boolean.post self result
          ⟹
          Product.inv self"
  apply(simp add: Product_post Product_inv)
  apply(safe)
  apply(ocl_simp)
  done


lemma "∃ qty self τ.  (τ ⊨ Product.inv self)
                    ∧(τ ⊨ release_Integer_Boolean.pre self qty)"
  apply(simp add: supply_pre Product_inv)
  apply(rule_tac x="1" in exI)
  apply(rule_tac x="λ(a, b). (‸((OclAny_key.OclAny, oid :: oid),
            ‸noext(Product_key.Product, ‸1‸, ‸1‸)‸)‸)"
    in exI)
    apply(rule exI)+
  apply(rule safe)
  apply(simp_all add: Product.stock_def Product.stock_def
                      ss_lifting localValid2sem
                      Zero_ocl_int_def One_ocl_int_def OclStrongEq_def
                      OclLess_def OclLe_def )
done

end
```

Table B.3: An theory file for HOL-OCL showing a formal analysis of the Invoice case-study.

BIBLIOGRAPHY

References are in alphabetical order. References with more than one author are sorted according to the first author.

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005. doi: 10.1007/s10270-004-0058-x. (Cited on pages 73 and 179.)

[2] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 2002. ISBN 1-402-00763-9. (Cited on pages 14 and 33.)

[3] Catia M. Angelo, Luc J. M. Claesen, and Hugo De Man. Degrees of formality in shallow embedding hardware description languages in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications (HUG)*, volume 780 of *Lecture Notes in Computer Science*, pages 89–100, Heidelberg, 1994. Springer-Verlag. ISBN 3-540-57826-9. doi: 10.1007/3-540-57826-9_127. (Cited on page 175.)

[4] Demissie B. Aredo. Formalizing UML class diagrams in PVS. In *OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations, Denver, Colorado*, Denver, Colorado, USA, November 1999. (Cited on page 176.)

[5] David Aspinall. Proof General: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-67282-6. (Cited on pages 15 and 163.)

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy,

Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Heidelberg, 2005. Springer-Verlag. ISBN 978-3-540-24287-1. doi: 10.1007/b105030. (Cited on pages 16, 71, 81, and 181.)

[7] David A. Basin, Seán Matthews, and Luca Viganò. Natural deduction for non-classical logics. *Studia Logica*, 60(1):119–160, 1998. ISSN 0039-3215. doi: 10.1023/A:1005003904639. Special issue on *Natural Deduction* edited by Frank Pfenning and Wilfried Sieg. (Cited on page 153.)

[8] David A. Basin, Hironobu Kuruma, Kazuo Takaragi, and Burkhart Wolff. Verification of a signature architecture with HOL-Z. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 269–285, Heidelberg, 2005. Springer-Verlag. ISBN 978-3-540-27882-5. doi: 10.1007/11526841_19. (Cited on page 17.)

[9] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: from UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1): 39–91, January 2006. ISSN 1049-331X. (Cited on page 186.)

[10] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science*, pages 266–280, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-37187-8. doi: 10.1007/11814771_23. (Cited on page 179.)

[11] Bernhard Beckert and Joachim Posegga. leanT^AP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995. doi: 10.1007/BF00881804. (Cited on page 177.)

[12] Bernhard Beckert and Steffen Schlager. Refinement and retrenchment for programming language data types. *Formal Aspects of Computing*, 17(4):423–442, 2005. doi: 10.1007/s00165-005-0073-x. (Cited on page 186.)

[13] Bernhard Beckert, Stefan Gerberding, Reiner Hähnle, and Werner Kernig. The many-valued tableau-based theorem prover $_3T^AP$. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 758–760, Heidelberg, 1992. Springer-Verlag. ISBN 978-3-540-55602-2. doi: 10.1007/3-540-55602-8_219. (Cited on page 177.)

[14] Bernhard Beckert, Uwe Keller, and Peter H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In Serge Autexier and Heiko Mantel, editors, *Verification Workshop (VERIFY)*, pages 113–123, 2002. (Cited on pages 178 and 181.)

[15] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL—lessons learned in formal-logic engineering. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66463-7. (Cited on page 177.)

[16] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Transaction on Knowledge and Data Engineering*, 4(3):223–237, 1992. doi: 10.1109/69.142014. (Cited on page 100.)

[17] Gavin M. Bierman and Matthew J. Parkinson. Effects and effect inference for a core Java calculus. *Electronic Notes in Theoretical Computer Science*, 82(7):1–26, 2003. doi: 10.1016/S1571-0661(04)80803-X. (Cited on page 17.)

[18] Richard Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland Publishing Co. ISBN 0-444-89686-4. (Cited on pages 39 and 175.)

[19] Manfred Broy, Michelle L. Crane, Jürgen Dingel, Alan Hartman, Bernhard Rumpe, and Bran Selic. 2nd UML 2 semantics symposium: Formal semantics for UML. In Thomas Kühne, editor, *Models in Software Engineering—Workshops and Symposia at MoDELS 2006*, volume 4364 of *Lecture Notes in Computer Science*, pages 318–323, Genua, Italy, 2006. Springer-Verlag. ISBN 978-3-540-69488-5. doi: 10.1007/978-3-540-69489-2_39. (Cited on page 109.)

[20] Achim D. Brucker and Burkhart Wolff. Using theory morphisms for implementing formal methods tools. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs (TYPES)*, volume 2646 of *Lecture Notes in Computer Science*, pages 59–77, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-14031-X. (Cited on page 38.)

[21] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, Zurich, Switzerland, 2006. (Cited on pages 41, 51, 58, 111, 117, 119, 124, 129, and 132.)

[22] Achim D. Brucker and Burkhart Wolff. HOL-TestGen 1.0.0 user guide. Technical Report 482, ETH Zurich, April 2005. (Cited on page 187.)

[23] Achim D. Brucker and Burkhart Wolff. A package for extensible object-oriented data models with an application to IMP++. In Serge Autexier and Heiko Mantel, editors, *Verification Workshop (VERIFY)*, pages 58–84, August 2006. (Cited on pages 80, 99, and 187.)

[24] Achim D. Brucker and Burkhart Wolff. Symbolic test case generation for primitive recursive functions. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing (FATES)*, volume 3395 of *Lecture Notes in Computer Science*, pages 16–32. Springer-Verlag, Linz, 2005. ISBN 3-540-25109-X. doi: 10.1007/b106767. (Cited on page 31.)

[25] Achim D. Brucker and Burkhart Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology*, 7(3):233–247, 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0176-3. (Cited on page 131.)

[26] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003. ISSN 0948-6968. (Cited on pages 17, 33, and 132.)

[27] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. (Cited on pages 163, 165, and 187.)

[28] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. (Cited on page 117.)

[29] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. A model transformation semantics and analysis methodology for secureuml. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 306–320, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_22. (Cited on page 186.)

[30] Richard Bubel and Reiner Hähnle. Integration of informal and formal development of object-oriented safety-critical software. *International Journal on Software Tools for Technology*, 7(3):197–211, 2005. ISSN 1433-2779. doi: 10.1007/s10009-004-0166-5. (Cited on page 187.)

[31] María Victoria Cengarle and Alexander Knapp. A formal semantics for OCL 1.4. In Martin Gogolla and Cris Kobryn, editors, *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 118–133, Heidelberg, 2001. Springer-Verlag. ISBN 3-540-42667-1. (Cited on pages 17 and 178.)

[32] Marìa Victoria Cengarle and Alexander Knapp. OCL 1.4/5 vs. 2.0 expressions formal semantics and expressiveness. *Software and Systems Modeling*, 3(1):9–30, 2004. ISSN 1619-1366. doi: 10.1007/s10270-003-0035-9.

[33] Juan Martín Chiaradía and Claudia Pons. Improving the OCL semantics definition by applying dynamic meta modeling and design patterns. *Electronic Communications of the EASST*, 5, 2006. ISSN 1863-2122. (Cited on pages 178 and 179.)

[34] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML models consistency using the OCL environment. *Electronic Notes in Theoretical Computer Science*, 102:99–110, 2004. doi: 10.1016/j.entcs.2003.09.005. (Cited on pages 18, 74, and 179.)

[35] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940. (Cited on page 33.)

[36] Common Criteria. Common criteria for information technology security evaluation (version 3.1), Part 3: Security assurance components, September 2006. Available as document CCMB-2006-09-003. (Cited on page 184.)

[37] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. The amsterdam manifesto on OCL. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 115–149, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1. (Cited on pages 17, 105, 113, and 178.)

[38] Birgit Demuth, Heinrich Hussmann, and Ansgar Konermann. Generation of an OCL 2.0 parser. In Thomas Baar, editor, *Workshop on*

Tool Support for OCL and Related Formalisms—Needs and Trends, Technical Report LGL-REPORT-2005-001, pages 38–52. EPFL, 2005. (Cited on page 18.)

[39] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66158-1. (Cited on pages 17, 175, and 176.)

[40] Sophie Dupuy, Angès Front-Conte, and Christophe Saint-Marcel. Using UML with a behaviour-driven method. In Frappier and Habrias [42], chapter 6. ISBN 1-85233-353-7. (Cited on page 196.)

[41] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66158-1. (Cited on pages 17, 175, and 176.)

[42] Marc Frappier and Henri Habrias, editors. *Software Specification Methods: An Overview Using a Case Study*. Formal Approaches to Computing and Information Technology. Springer-Verlag, London, 2000. ISBN 1-85233-353-7. (Cited on pages 167, 195, and 206.)

[43] Dov M. Gabbay. *Labelled Deductive Systems*, volume 1 of *Oxford Logic Guides*. Oxford University Press, Inc., New York, NY, USA, 1997. ISBN 978-0-198-53833-2. (Cited on pages 134 and 153.)

[44] Martin Gogolla and Mark Richters. Expressing UML class diagrams properties with OCL. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 85–114, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1. (Cited on page 109.)

[45] Mike Gordon. From LCF to HOL: a short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–185. MIT Press, Cambridge, Massachusetts, 2000. ISBN 978-0-262-16188-6. (Cited on page 31.)

[46] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, July 1993. ISBN 0-521-44189-7. (Cited on pages 34, 35, 36, and 176.)

[47] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, USA, 1993. ISBN 0-387-94006-5. (Cited on page 16.)

[48] Reiner Hähnle. *Automated Deduction in Multiple-valued Logics*. Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853989-4. (Cited on page 186.)

[49] Reiner Hähnle. Efficient deduction in many-valued logics. In *International Symposium on Multiple-Valued Logics (ISMVL)*, pages 240–249, Los Alamitos, CA, USA, 1994. IEEE Computer Society. ISBN 0-8186-5650-6. doi: 10.1109/ismvl.1994.302195. (Cited on pages 137 and 153.)

[50] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IGPL*, 13(4): 415–433, July 2005. doi: 10.1093/jigpal/jzi032. (Cited on page 178.)

[51] Reiner Hähnle. Tableaux for many-valued logics. In Marcello D'Agostino, Dov Gabbay, Reiner Hähnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*, pages 529–580. Kluwer Academic Publishers, Dordrecht, 1999. ISBN 978-0-792-35627-1. (Cited on pages 132 and 153.)

[52] Reiner Hähnle. Towards an efficient tableau proof procedure for multiple-valued logics. In Egon Börger, Hans Kleine Büning, Michael M. Richter, and Wolfgang Schönfeld, editors, *Computer Science Logic (CSL)*, volume 533 of *Lecture Notes in Computer Science*, pages 248–260, Heidelberg, 1991. Springer-Verlag. ISBN 978-3-540-54487-6. doi: 10.1007/3-540-54487-9_62. (Cited on pages 57 and 150.)

[53] Ali Hamie, Franco Civello, John Howse, Stuart Kent, and Richard Mitchell. Reflections on the Object Constraint Language. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 162–172, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-66252-9. doi: 10.1007/b72309. (Cited on pages 105 and 178.)

[54] Ali Hamie, John Howse, and Stuart Kent. Interpreting the Object Constraint Language. In *Proceedings of Asia Pacific Conference in Software Engineering (APSEC)*, pages 288–295, Los Alamitos, CA, USA, 1998. IEEE Computer Society. ISBN 0-8186-9183-2. doi: 10.1109/apsec.1998.733731. (Cited on pages 17 and 178.)

[55] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer*, 37(10):64–72, October 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.172. (Cited on page 106.)

[56] Rolf Hennicker, Heinrich Hußmann, and Michel Bidoit. On the precise meaning of OCL constraints. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 69–84, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1. (Cited on pages 57 and 150.)

[57] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11(1):31–55, 1978. doi: 10.1007/BF00264598. (Cited on page 38.)

[58] Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162, Heidelberg, 2005. Springer-Verlag. ISBN 978-3-540-28372-0. doi: 10.1007/11541868_10. (Cited on page 177.)

[59] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2): 256–290, 2002. ISSN 1049-331X. doi: 10.1145/505145.505149. (Cited on page 16.)

[60] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the Alloy constraint analyzer. In *International Conference on Software Engineering (ICSE)*, pages 730–733, New York, NY USA, June 2000. ACM Press. ISBN 1-58113-206-9. doi: 10.1109/ICSE.2000.870482. (Cited on page 16.)

[61] Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security—Theories and Systems (ISSS)*, volume 3233 of *Lecture Notes in Computer Science*, pages 134–153, Heidelberg, 2004. Springer-Verlag. ISBN 978-3-540-23635-1. doi: 10.1007/b102118. (Cited on pages 176 and 181.)

[62] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1990. 0-13-880733-7. (Cited on page 11.)

[63] Manfred Kerber and Michael Kohlhase. A mechanization of strong kleene logic for partial functions. In Alan Bundy, editor, *Automated Deduction—CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 371–385, Heidelberg, 1994. Springer-Verlag. ISBN 3-540-58156-1. doi: 10.1007/3-540-58156-1_26. (Cited on pages 132 and 177.)

[64] Manfred Kerber and Michael Kohlhase. A tableau calculus for partial functions. In *Collegium Logicum—Annals of the Kurt-Gödel-Society*, volume 2, pages 21–49. Springer-Verlag, New York, NY, USA, 1996. ISBN 3-211-82796-X. (Cited on page 153.)

[65] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Object-oriented programming systems, languages and applications (OOPSLA)*, pages 406–416, New York, NY USA, 1986. ACM Press. ISBN 0-89791-204-7. doi: 10.1145/28697.28739. (Cited on page 82.)

[66] Stephen C. Kleene. *Introduction to Meta Mathematics*. Wolters-Noordhoff Publishing, Amsterdam, 1971. ISBN 0-7204-2103-9. Originally published by Van Nostrand, 1952. (Cited on page 56.)

[67] Cris Kobryn. UML 2001: a standardization odyssey. *Communications of the ACM*, 42(10):29–37, 1999. ISSN 0001-0782. doi: 10.1145/317665.317673. (Cited on page 104.)

[68] Marcel Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. PhD thesis, University of Leiden, Berlin, 2006. (Cited on pages 178, 179, and 181.)

[69] Marcel Kyas, Harald Fecher, Frank S. de Boer, Mark van der Zwaag, Jozef Hooman, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. *Electronic Notes in Theoretical Computer Science*, pages 39–47, 2004. doi: 10.1016/j.entcs.2004.09.027. (Cited on page 181.)

[70] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Dordrecht, 1999. ISBN 978-0-7923-8629-2. (Cited on pages 16 and 99.)

[71] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML reference manual (revision 1.2), February 2007. Available from `http://www.jmlspecs.org`. (Cited on page 99.)

[72] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42, Heidelberg, 2005. Springer-Verlag. ISBN 978-3-540-27882-5. doi: 10.1007/11526841_4. (Cited on pages 16, 71, 79, 81, and 181.)

[73] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. ISSN 0164-0925. doi: 10.1145/197320.197383. (Cited on pages 22 and 95.)

[74] Luis Mandel and Marìa Victoria Cengarle. On the expressive power of OCL. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66587-0. (Cited on page 105.)

[75] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194, Heidelberg, 2005. Springer-Verlag. ISBN 978-3-540-28372-0. doi: 10.1007/11541868_12. (Cited on page 16.)

[76] Sališa Marković and Thomas Baar. An OCL semantics specified with QVT. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *Lecture Notes in Computer Science*, pages 661–675, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-45772-5. doi: 10.1007/11880240_46. (Cited on pages 178 and 179.)

[77] Thomas F. Melham. A package for inductive relation definitions in HOL. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *International Workshop on the HOL Theorem Proving System and its Applications (TPHOLS)*, pages 350–357, Los Alamitos, CA, USA, 1992. IEEE Computer Society. ISBN 0-8186-2460-4. (Cited on pages 165 and 177.)

[78] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-629031-0. (Cited on page 16.)

[79] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77, Heidelberg, 2000. Springer-Verlag. ISBN 3-540-67282-6. (Cited on pages 71, 176, and 181.)

[80] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9 (2):191–223, 1999. (Cited on page 17.)

[81] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLS)*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366, Heidelberg, 1998. Springer-Verlag. ISBN 3-540-64987-5. doi: 10.1007/BFb0055146. (Cited on pages 176 and 177.)

[82] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Workshop on Logical Environments*, pages 164–188. Cambridge University Press, New York, NY, USA, 1993. ISBN 0-521-43312-6. (Cited on page 34.)

[83] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998. doi: 10.1007/s001650050009. (Cited on page 176.)

[84] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe—definitely. In *ACM Symp. Principles of Programming Languages (POPL)*, pages 161–170, New York, NY USA, 1998. ACM Press. ISBN 0-89791-979-3. doi: 10.1145/268946.268960. (Cited on pages 17 and 175.)

[85] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144, Amsterdam, The Netherlands, 2000. IOS Press. ISBN 978-1-58603-015-5. (Cited on pages 175, 176, and 187.)

[86] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002. (Cited on pages 14, 31, and 159.)

[87] Object Management Group. Object constraint language specification (version 1.1), September 1997. Available as OMG document ad/97-08-08. (Cited on page 178.)

[88] Object Management Group. UML 2.0 OCL specification, October 2003. Available as OMG document ptc/03-10-14. (Cited on pages 14, 18, 27, 37, 103, 104, 105, 106, 107, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 125, 126, 127, 128, 129, 159, 167, 178, 182, 185, and 189.)

[89] Object Management Group. UML 2.0 OCL specification, April 2006. Available as OMG document formal/06-05-01. (Cited on pages 105 and 129.)

[90] Object Management Group. Unified modeling language specification (version 1.5), March 2003. Available as OMG document formal/03-03-01. (Cited on pages 12, 30, 105, 107, 109, 124, and 125.)

[91] Object Management Group. UML 2.0 superstructure specification, July 2005. Available as OMG document formal/05-07-04. (Cited on page 183.)

[92] Object Management Group. OMG XML metadata interchange (XMI) specification (version 1.1), November 2000. Available as OMG document formal/00-11-02. (Cited on page 165.)

[93] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, Heidelberg, 1996. Springer-Verlag. ISBN 3-540-61474-5. doi: 10.1007/3-540-61474-5_91. (Cited on pages 176 and 181.)

[94] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 187–211. MIT Press, Cambridge, Massachusetts, 2000. ISBN 978-0-262-16188-6. (Cited on pages 76 and 177.)

[95] Lawrence C. Paulson. A formulation of the simple theory of types (for Isabelle). In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 246–274, Heidelberg, 1990. Springer-Verlag. ISBN 3-540-52335-9. doi: 10.1007/3-540-52335-9_58. (Cited on page 34.)

[96] Lawrence C. Paulson. Generic automatic proof tools. In Robert Veroff, editor, *Automated reasoning and its applications: essays in honor of Larry Wos*, pages 23–47. MIT Press, Cambridge, Massachusetts, 1997. ISBN 978-0-262-22055-2. (Cited on page 161.)

[97] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 17, pages 1063–1147. Elsevier Science Publishers, Amsterdam, 2001. ISBN 0-444-50812-0. (Cited on page 31.)

[98] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, New York, NY USA, 1988. ACM Press. ISBN 0-89791-269-1. doi: 10.1145/53990.54010. (Cited on page 38.)

[99] Nicole Rauch and Burkhart Wolff. Formalizing Java's two's-complement integral type in Isabelle/HOL. *Electronic Notes in Theoretical Computer Science*, 80:1–18, 2003. doi: 10.1016/S1571-0661(04)80808-9. (Cited on page 59.)

[100] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002. (Cited on pages 18, 73, 74, 105, 106, 178, and 185.)

[101] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok Wang Ling, Sudha Ram, and Mong-Li Lee, editors, *Conceptual Modeling—ER '98*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464, Heidelberg, 1998. Springer-Verlag. ISBN 978-3-540-65189-5. doi: 10.1007/b68220. (Cited on pages 17 and 178.)

[102] Mark Richters and Martin Gogolla. OCL: Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*, pages 42–68, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43169-1. (Cited on pages 17, 18, and 179.)

[103] Bertrand Russell. *Introduction to Mathematical Philosophy*. George Allen & Unwin, London, 1919. (Cited on page 5.)

[104] Thomas Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. PhD thesis, Technical University Berlin, June 1999. (Cited on pages 176 and 177.)

[105] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. (Cited on page 176.)

[106] Graeme Smith. *The Object Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, Dordrecht, 2000. ISBN 0-7923-8684-1. (Cited on page 15.)

[107] Graeme Smith, Florian Kammüller, and Thomas Santen. Encoding Object-Z in Isabelle/HOL. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 82–99, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43166-7. (Cited on page 176.)

[108] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1992. ISBN 0-139-78529-9. (Cited on pages 11 and 15.)

[109] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*, Workshops in Computing, Heidelberg, 1992. Springer-Verlag. ISBN 3-540-19778-8. (Cited on page 15.)

[110] Don Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66158-1. (Cited on page 175.)

[111] José Ambrosio Toval, Vìctor Requena, and José Luis Fernández. Emerging OCL tools. *Software and Systems Modeling*, 2(4):248–261, December 2003. ISSN 1619-1366. doi: 10.1007/s10270-003-0031-0. (Cited on page 179.)

[112] Luca Viganò. *Labelled Non-Classical Logics*. Kluwer Academic Publishers, Dordrecht, 2000. ISBN 0-7923-7749-4. (Cited on pages 134 and 153.)

[113] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. (Cited on pages 175 and 176.)

[114] David von Oheimb and Tobias Nipkow. Hoare logic for Nano-Java: Auxiliary variables, side effects, and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 89–105, Heidelberg, 2002. Springer-Verlag. ISBN 3-540-43928-5. (Cited on pages 17 and 176.)

[115] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156, Heidelberg, 1999. Springer-Verlag. ISBN 3-540-66158-1. (Cited on page 175.)

[116] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman, Inc., Reading, MA, USA, 2nd edition, August 2003. ISBN 0-321-17936-6. (Cited on pages 167 and 189.)

[117] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, Cambridge, Massachusetts, 1993. ISBN 0-262-23169-7. (Cited on pages 37 and 90.)

[118] Kenro Yatake, Toshiaki Aoki, and Takuya Katayama. Implementing application-specific object-oriented theories in HOL. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing—ICTAC 2005*, volume 3722 of *Lecture Notes in Computer Science*, pages 501–516, Springer-Verlag, 2005. Springer-Verlag. ISBN 3-540-29107-5. doi: 10.1007/11560647_33. (Cited on page 176.)

LIST OF ACRONYMS

ACM	Association for Computing Machinery
APSEC	Asia Pacific Conference in Software Engineering
CADE	Automated Deduction
CASE	Computer Aided Software Engineering
CASSIS	Construction and Analysis of Safe, Secure, and Interoperable Smart Devices
CAV	Computer Aided Verification
COLOG	International Conference on Computer Logic
CSL	Conference on Computer Science Logic
EAL	Evaluation Assurance Level
EASST	European Association of Software Science and Technology
EBNF	Extended Backus-Naur Form
EPFL	Eole Polytechnique Fédérale de Lausanne
ETH	Eidgenössische Technische Hochschule
FME	Formal Methods Europe
HOAS	higher-order abstract syntax
HOL	higher-order logic
HOLCF	HOL + LCF
HOOL	An object-oriented specification and verification environment.
HUG	Higher-Order Logic User's Group

ICTAC	International Colloquium on Theoretical Aspects of Computing
ICSE	International Conference on Software Engineering
IEEE	Institute of Electrical and Electronic Engineers
IFIP	International Federation for Information Processing
IJCAR	International Joint Conference on Automated Reasoning
IMP	A simple imperative programming language
ISMVL	International Symposium on Multiple-Valued Logics
ISSS	Software Security—Theories and Systems
JML	Java Modeling Language
JVM	Java virtual machine
LCF	Logic for Computable Functions
LE	Local (Formula) Equivalence
LEC	Local Equational Calculus
LJE	Local Judgement Equivalence
LOOP	Logic of Object-Oriented Programming
LTC	Local Tableaux Calculus
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MIT	Massachusetts Institute of Technology
MoDELS	Model Driven Engineering Languages and Systems
NATO	North Atlantic Treaty Organization
NY	New York
OCL	Object Constraint Language
OCLE	Object Constraint Language Environment
OCLVP	Object Constraint Language Verification Platform
OMG	Object Management Group

OOPSLA	International Conference on Object-oriented Programming Systems, Languages and Applications
PLDI	Programming Language Design and Implementation
POPL	Symposium on Principles of Programming Languages
PVS	PVS Verification System
QVT	Query/View/Transformation
SKL	Strong Kleene Logic
SML	Standard Meta Language
TACAS	Tools and Algorithms for the Construction and Analysis of Systems
TPHOLS	Theorem Proving in Higher Order Logics
TYPES	Types for Proofs and Programs
UC	Universal (Formula) Congruence
UEC	Universal Equational Calculus
UJE	Universal Judgement Equivalence
UML	Unified Modeling Language
USA	United States of America
USE	UML Specification Environment
VERIFY	Verification Workshop
VDM	Vienna Development Method
XMI	XML Metadata Interchange
XML	Extensible Markup Language
ZF	Zermelo-Fraenkel

Page numbers in *italic* type refer to main entries and definitions. The normal printed numbers refer to the use of an index entry.

symbols

α option, 36
$V_\tau(\alpha)$, *49*
$\ulcorner_\urcorner$, 38, *49*
$\llcorner_\lrcorner$, 38, *49*
$\tau_\perp$, *38*, *49*

A

abstraction, *22*, 34
access specifier, *22*
adaption
    datatype, 43, *49*
    embedding, 44, *53*
    functional, 43, *52*
alias, *99*
Alloy, 16
application, 34
arithmetic, 143
association, *26*, 109
attribute type, *64*
attribute, *21*, *26*
axiom, *33*, 35

B

base class type, *65*
Boogie, 16, 181
bot, *49*
boxing, *22*

C

call, *24*
class, *21*, 24, 26
class invariant, *see* invariant
class type, 62, *67*
closed-world, 42, 43, *61*
coercion, 69
combinator, 37
comprehensions, *147*
computation, 143
congruence, 134
conservative, 13, 45, 46, *94*
    extension, *35*
    type definition, 51
consistent, 35
constant, 34
constraint language, 12
    object-oriented, 42, 43, *54*, *100*, 131, 184
constructor
    type, 36, 45, 50
context, 120
    declaration, 115
    lifting, 37, 49, *50*
    object, *88*
    passing, *50*, *139*
core-logic, *146*
C#, 23, 124

D

data model, *24*
data structure
    smashed, 51, 147
datatype
    recursive, 39

datatype package, 14, *163*, 165, 177
definedness, *49*, 49, 134
definition
  constant, *35*, 35, 47, 51, 54, 58
    finite family, *93*
  type, 35, *36*
destruction rules, *156*
diagram
  activity, 12, 24
  class, 12, *24*, 24, 62, 64, 110
    extension, 62
  collaboration, 24
  component, 24
  object, 24
  sequence, 12, 24
  state, 12, 24, 186

E
embedding, *38*, 131, 175
  conservative, 13, *40*
  deep, 16, *39*, 175
  shallow, 13, 17, *39*, 41, 176, 183
encapsulation, *22*
encoder, 163
environment, *46*
equality, 81, 113
  identity, 83
  operator, *84*
  reference, *82*
  strict, *85*
  strong, *84*, 146
  value
    deep, *83*
    shallow, *83*
equivalence, 134, 138
  relation, *82*
extensionality, *35*

F
finalize, *69*, 96
formal method, 11
formula
  typed, *46*
frame property, *97*

G
generalization, *26*

H
Haskell, 49
higher-order logic, *see* HOL
Hilbert operator, *34*
HOAS, *38*
HOL, 13, 31, 45, 183
HOL-OCL, 14, 19, 103, *163*, 163, 185
  architecture, 163
  encoder, 165
  library, *165*, 166
HOL-Z, 17
HOLCF, 17
HOOL, 176
Horn-clause, 153

I
implementation, 23
implication, 147, 150
  introduction, 32, *35*
informative, 106
inheritance, *22*, 23, 26, 41, 42, 62, 109
  multiple, *22*
  single, *22*
instance, 22, 86
interface, *23*, 24, 109
`invalid`, 105
invariant, *27*, 44, *71*, 108, 114
invocation, *24*
invoke, *52*
Isabelle, 14, 18, 20, 30, *31*, 143, 172
isomorphism, 46

J
Java, 17, 21, 23, 65, 73, 124, 175
Jive, 71, 176, 181
JML, 16, 132
judgment, 143

K
KeY, 73, 179
kind, *68*

Krakatoa, 16

L

labeling, 153
λ-abstraction, *35*
λ-calculus, *33*
λ-term, *33*
late-binding, *24*, 52
layer, *43*
LE, *136*
Least, 53
level, *43*
lift, 49
LJE, *136*
logic
    propositional fragment, *146*
    three-valued
        Kleene, *56*
        lazy, *55*
        strict, *55*
logical framework, 31
logical judgment, 74, 84
LOOP tool, 176

M

MDA, 12, 163, 187
MDE, 12, 163, 184, 187
member, *22*
membership, *147*
meta-implication, *32*
meta-language, 17, *31*, 45
meta-logic, *31*, 38
meta-quantifier, *32*
meta-variable, *32*
metamodel, 12
method, 21, 22, *23*
model-checking, 11
modular proof, *43*
modus ponens, *35*
multiplicity, *26*

N

namespace, *26*
natural deduction rules, 31
None, *36*
normative, 106
`null`, 105
number, 143

O

Object, *65*
object, *22*, 62, 86
    identifier, *22*
    instance, *26*, 62
    store, 13, 42, 43, 66, *99*, 184
    structure, *42*
object-based, *23*
object-language, 17
object-logic, *31*, 33, 38
object-orientation, 21
object-oriented, *23*
OCL, 12–15, 19, *27*, 103, 132, 178
    semantics, 103, 178
    syntax, 27, 189
OCLE, 18, 179
`OclUndefined`, 105
`OclVoid`, 109
OCLVP, 181
OMG, 12, 104
open-world, 43, *61*, 69
operation, 21, 22, *23*, *26*, 41–43, 114
    call, 42, *89*
    invocation, 42, *89*
    overloading, *23*
    overriding, *23*, 52, 89
    specification, 87, *88*
        totalized, *89*
    strict, *37*
    table, *92*

P

package, *26*
partial map, 92
path expression, *42*, 108
pathname, *26*
polymorphism, *23*
    overloading, 23
    overriding, 23

postcondition, 12, 14, *27*, 42, 44, 159

pre-compilation, *16*, 181
precondition, 12, 14, *27*, 42, 44
private, 22
proof
    formal, 31
    goal, *32*
    modular, 43
    object, *33*, 33, 160
    procedure, 153
    procedures, 186
    state, *32*, 32
Proof General, 15, 163
proof procedure, 33
protected, 22
public, 22

Q

quadrium non datur, *135*
quantifier, 147, 156
QVT, 179

R

recursion, 126
reduction rules ($\mathcal{R}$-rules), *150*
reference, *22*
refinement, 186
reflexivity, *35*
requirement, 30
result, *88*
retrenchment, 186
rule
    derived, 33

S

SecureUML, 186
self, *88*
semantic combinator, *49*
semantics, 45
    combinator-style, *37*, 117
    copy, *64*
    formal, 37
    machine-checked, *37*, 107, 183
    sharing, *64*
    textbook, *37*, 117
SemCom, 47, 50, 51, 53
set theory, 147
signature, *46*
signature morphism, *46*
simplifier, 143, 144, 161
smashing, *50*, 51
SML, 163
Some, *36*
sort, *33*, 45
soundness, 158
specification, 11, 23, 30
    increment, *47*
    morphism, 46
Spec#, 16, 71, 132, 181
state, *41*
    transition, *41*
sterilize, *69*
strictify, *52*
strictness, *49*, 49
Strong Kleene Logic, 14, *56*
su4sml, *163*, 165
subcalculus, *42*, 50
subgoal, *32*
substitution, 39
subtype, *22*, 23, 26, 76
subtyping, 23, 41, 62, 109
    multiple, *22*
    single, *22*
superclass, 22
supertype, *22*, 65, 66
symmetry, *35*
system state, 86

T

tactics, *33*
tag type, *65*
term, 45
    typed, *46*
tertium non datur, *35*
testing, *31*
    model-based, 11
theorem, *33*, 33

theorem prover, 12, *31*
theory, *35*, *46*
  closure, *46*
  morpher, *165*, 166
  morphism, *46*, 54, 185
    layered, 45, *47*
transitivity, *35*
trichotomy, *151*
type, *34*
  arity, *46*
  basic, *22*
  class, *33*
  constructor, *33*
  dynamic, *23*
  primitive, *22*, *42*, 58, 62, *82*
  semantic, *71*
  static, *23*
  structural, *71*
  value, 22, 58
  variable, *33*
type set, *76*
type-cast, *24*, *69*, 108
type-checking, 11

U

UC, *135*
UEC, *150*
UJE, *136*
UML, 12, 14, 15, 19, *24*, 24, 178, 183
undefined, 43
undefinedness, *42*, 134
  universal, 136
unification
  higher-order, 32
universal quantifier
  meta, 32
universe
  non-referential, 64, *67*
  referential, 64, *66*
universe type, *66*
USE, 18, *73*, 179

V

validation, 11, *30*, 30
validity, 133, 159
  global, *133*
  local, *133*, 159
  universal, 133, 136
value, 22, *82*
variable, 34
VDM, 11
verification, *30*, 30
visibility, *26*, 109
  private, 22
  protected, 22
  public, 22

X

XMI, 14

Z

Z, 11, 15

CURRICULUM VITÆ

since 01/2003	Research Assistant at the Information Security Group, headed by Prof. David Basin, ETH Zurich, Switzerland.
06/2000–12/2002	Research Assistant at the Chair for Software Engineering, headed by Prof. David Basin, University of Freiburg, Germany.
06/2000	Diplom Informatiker (Masters of Computer Science), University of Freiburg, Germany. Title of thesis: *Verifikation von Dividieren mit Word-Level-Decision-Diagrams* (Verification of Division Circuits using Word-level Decision-diagrams), supervised by Prof. Dr. Bernd Becker.
01/1995-05/2000	Student of computer science with second subject microsystems engineering at the University of Freiburg, Germany.