

A Proposal for a Formal OCL Semantics in Isabelle/HOL

Achim D. Brucker and Burkhart Wolff

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 52, D-79110 Freiburg, Germany
{brucker,wolff}@informatik.uni-freiburg.de
<http://www.informatik.uni-freiburg.de/~{brucker,wolff}>

Abstract. We present a formal semantics as a conservative shallow embedding of the Object Constraint Language (OCL). OCL is currently under development within an open standardization process within the OMG; our work is an attempt to accompany this process by a proposal solving open questions in a consistent way and exploring alternatives of the language design. Moreover, our encoding gives the foundation for tool supported reasoning over OCL specifications, for example as basis for test case generation.

Keywords: Isabelle, OCL, UML, shallow embedding, testing

1 Introduction

The Unified Modeling Language (UML) [1] has been widely accepted throughout the software industry and is successfully applied to diverse domains [2]. UML is supported by major CASE tools and integrated into a software development process model that stood the test of time. The Object Constraint Language (OCL) [3, 4, 5] is a textual extension of the UML. OCL is in the tradition of data-oriented formal specification languages like Z [6] or VDM [7]. For short, OCL is a three-valued Kleene-Logic with equality that allows for specifying constraints on graphs of object instances whose structure is described by UML class diagrams.

In order to achieve a maximum of acceptance in industry, OCL is currently developed within an open standardization process by the OMG. Although the OCL is part of the UML standard since version 1.3, at present, the official OCL standard 1.4 concentrates on the concrete syntax, covers only in parts the well-formedness of OCL and handles nearly no formal semantics. So far, the description of the OCL is merely an informal requirement analysis document with many examples, which are sometimes even contradictory.

Consequently, there is a need¹ for both software engineers and CASE tool developers to clarify the concepts of OCL formally and to put them into perspective of more standard semantic terminology. In order to meet this need, we started to provide a conservative embedding of OCL into Isabelle/HOL. As far

¹ This work was partially funded by the OMG member Interactive Objects Software GmbH (www.io-software.com).

as this was possible, we tried to follow the design decisions of OCL 1.4 in order to provide insight into the possible design choices to be made in the current standardization process of version 2.0.

Attempting to be a “practical formalism” [4], OCL addresses software developers who do not have a strong mathematical background. Thus, OCL deliberately avoids mathematical notation; rather, it uses a quite verbose, programming language oriented syntax and attempts to hide concepts such as logical quantifiers. This extends also to a design rationale behind the semantics: OCL is still viewed as an object-oriented assertion language and has thus more similarities with an object-oriented programming language than a conventional specification language. For example, standard library operators such as “concat” on sequences are defined as *strict operations* (i.e. they yield an explicit value *undefined* as result whenever an argument is undefined), only bounded quantifiers are admitted, and there is a tendency to define infinite sets away wherever they occur. As a result, OCL has a particularly executable flavor which comes handy when generating code for assertions or when animating specifications.

Object-oriented languages represent a particular challenge for the “art of embedding languages in theorem provers” [8]. This holds even more for a shallow embedding, which we chose since we aim at reasoning in OCL specifications and not at meta-theoretic properties of our OCL representation. In a shallow embedding, the types of OCL language constructs have to be represented by types of HOL and concepts such as undefinedness, mutual recursion between object instances, dynamic types, and extensible class hierarchies have to be handled.

In this paper, we present a new formal model of OCL in form of a conservative embedding into Isabelle/HOL that can cope with the challenges discussed above. Its modular organization has been used to investigate the interdependence of certain language features (method recursion, executability, strictness, smashing, flattening etc.) in order to provide insight into the possible design choices for the current design process [9]. We extend known techniques for the shallow representation of object orientation and automated proof techniques to lift lemmas from the HOL-library to the OCL level. As a result, we provide a first calculus to formally reason over OCL specifications and provide some foundation for automated reasoning in OCL.

This paper proceeds as follows: After a introduction into our running example using UML/OCL, we will guide through the layers of our OCL semantics, namely the object model and resulting semantic universes, the states and state relations, the OCL logic and the OCL library. It follows a description of automated deduction techniques based on derived rules, let it be on modified tableaux-deduction techniques or congruence rewriting. We will apply these techniques in a paradigmatic example for test-case generation in a black-box test setting.

2 A Guided Tour Through UML/OCL

The UML provides a variety of diagram types for describing dynamic (e.g. state charts, activity diagrams, etc.) and static (class diagrams, object diagrams, etc.) system properties. One of the more prominent diagram types of the UML is the

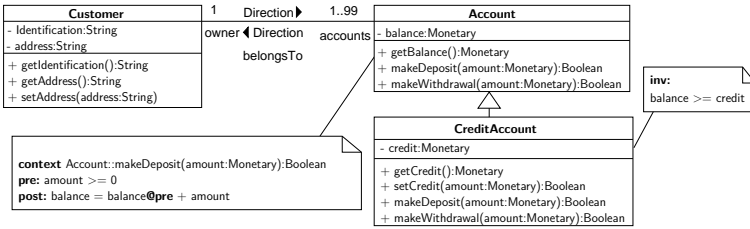


Fig. 1. Modeling a simple banking scenario with UML

class diagram for modeling the underlying data model of a system in an object oriented manner. The class diagram in our running example in Fig. 1 illustrates a simple banking scenario describing the relations between the classes **Customer**, **Account** and its specialization **CreditAccount**. To be more precise, the relation between data of the classes **Account** and **CreditAccount** is called *subtyping*. A class does not only describe a set of record-like data consisting of *attributes* such as *balance* but also functions (*methods*) defined over the classes data model.

It is characteristic for the object oriented paradigm, that the functional behavior of a class and all its methods are also accessible for all subtypes; this is called *inheritance*. A class is allowed to redefine an inherited method, as long as the method interface does not change; this is called *overwriting*, as it is done in the example for the method `makeWithdrawal()`.

It is possible to model relations between classes (*association*), possibly constrained by *multiplicities*. In Fig. 1, the association `belongsTo` requires, that every instance of **Account** is associated with exactly one instance of **Customer**. Associations were represented by introducing implicit set-valued attributes into the objects, while multiplicity were mapped to suitable data invariants. In the following, we assume that associations have already been “parsed away”.

Understanding OCL as a data-oriented specification formalism, it seems natural to refine class diagrams using OCL for specifying invariants, pre- and postconditions of methods. For example, see Fig. 1, where the specification of the method `makeWithdrawal()` is given by its pair of pre- and postcondition.

In UML class members can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive data types. Moreover, OCL introduces also recursively specified methods [3]; however, at present, a dynamic semantics of a method call is missing (see [9] for a short discussion of the resulting problems).

3 Representing OCL in Isabelle/HOL

OCL formulae are built over expressions that access the underlying state. In postconditions, path-expressions can access the *current* and the *previous state*, e.g. `balance = balance@pre + amount`. Accesses on both states may be arbitrarily mixed, e.g. `self.x@pre.y` denotes an object, that was constructed by dereferencing in the previous state and selecting attribute `x` in it, while the next dereferencing

step via y is done in the current state. Thus, method specifications represent state transition relations built from the conjunction of pre and post condition, where the state consists of a graph of object instances whose type is defined by the underlying class diagram. Since the fundamental OCL-operator `allInstances` allows for the selection of all instances (objects) of a certain class, there must be a means to reconstruct their dynamic types in the object graph.

In a shallow embedding, the key question arises how to represent the static type structure of the objects *uniformly* within a state or as argument of a dynamic type test is `T`. Constructions like “the universe of all class diagram interpretations” are too large to be represented in the simple type system underlying higher-order logic.

Our solution is based on the observation that we need not represent *all* class diagram interpretations inside the logic; it suffices if we can provide an extra-logical mechanism that represents any concrete class hierarchy and that allows for extensions of any given class hierarchy. For practical reasons, we require that such an *extension* mechanism is logically conservative both in the sense that only definitional axioms are used and that all existing proofs on data of the former class hierarchy remain valid for an extended one.

Based on these considerations, *HOL-OCL* is organized in several layers:

- a *semantic coding scheme* for the *object model* layer along the class-hierarchy,
- the *system state* and *relations* over it, forming the denotational domain for the semantics of methods,
- the *OCL logic* for specifying state relations or class invariants,
- the *OCL library* describing predefined basic types.

3.1 The Encoding of Extensible Object Models

The main goals of our encoding scheme is to provide typed constructors and accessor functions for a given set of classes or enumeration types to be inserted in a previous class hierarchy. The coding scheme will be represented in two steps: in this section, we will describe *raw* constructors and accessors, while in Sec. 3.2 a refined scheme for accessors is presented.

The basic configuration of any class hierarchy is given by the OCL standard; the library for this basic configuration is described in Sec. 3.5.

Handling Undefinedness. In the OCL standard 1.4, the notion of explicit undefinedness is part of the language, both for the logic and the basic values.

Object Constraint Language Specification [3] (version 1.4), page 6–58

Whenever an OCL-expression is being evaluated, there is the possibility that one or more queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

This requirement postulates the strictness of all operations (the logical operators are explicit exceptions) and rules out a modeling of undefinedness via underspecification. Thus, the language has a similar flavor than LCF or SPECTRUM [10] and represents a particular challenge for automated reasoning.

In order to handle undefinedness, we introduce for each type τ a *lifted type* τ_{\perp} , i.e. we introduce a special type constructor. It adds to each given type an additional value \perp . The function $[_] : \alpha \Rightarrow \alpha_{\perp}$ denotes the injection, the function $[_] : \alpha_{\perp} \Rightarrow \alpha$ its inverse. Moreover, we have the case distinction function $\text{case_up } c \ f \ x$ that returns c if $x = \perp$ and $f \ k$ if $x = [k]$. We will also write $\text{case } x \ \text{of } [k] \Rightarrow f \ k \mid \perp \Rightarrow c$.

Note that the definition of lifted types leads to the usual construction of *flat cpo*'s well known from the theory of complete partial orders (*cpo*) and denotational semantics [11]. For the sake of simplification, we avoid a full-blown cpo-structure here (while maintaining our semantics “cpo-ready”) and define only a tiny fragment of it that provides concepts such as *definedness* $\text{DEF}(x) \equiv (x \neq \perp)$ or *strictness of a function* $\text{is_strict } f \equiv (f \perp = \perp)$.

Managing Holes in Universes. Since objects can be viewed as records consisting of *attributes*, and since the object alternatives can be viewed as variants, it is natural to construct the “type of all objects”, i.e. the semantic universe \mathcal{U}_x corresponding to a certain class hierarchy x , by Cartesian products and by type sums (based on the constructors $\text{Inl} : \alpha \Rightarrow \alpha + \beta$ and $\text{Inr} : \beta \Rightarrow \alpha + \beta$ from the Isabelle/HOL library). In order to enable extensibility, we provide systematically polymorphic variables — the “holes” — into a universe that were filled when extending a class hierarchy.

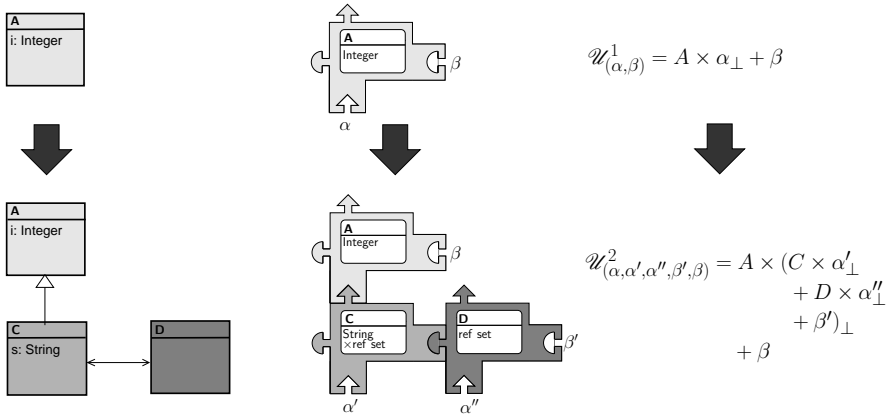


Fig. 2. Extending Class Hierarchies and Universes with Holes

In our scheme, a class can be extended in two ways: either, an alternative to the class is added *at the same level* which corresponds to the creation of an alternative subtype of the supertype (the β -instance), or a class is added *below* which corresponds to the creation of a subtype (the α -instance). The insertion of a class corresponds to *filling a hole* ν by a record T is implemented by the particular type instance:

$$\nu \mapsto ((T \times \alpha_{\perp}) + \beta)$$

As a consequence, the universe \mathcal{U}^2 in Fig. 2 is just a type instance of \mathcal{U}^1 , in particular: an α -extension with C β -extended by D . Thus, properties proven over \mathcal{U}^1 also holds for \mathcal{U}^2 .

The *initial* universe corresponding to the minimal class hierarchy of the OCL library consists of the real numbers, strings and bool. It is defined as follows:

$$\begin{aligned} \text{Real} &= \text{real}_\perp & \text{Boolean} &= \text{bool}_\perp & \text{String} &= \text{string}_\perp & \text{OclAny}_\alpha &= \alpha_\perp \\ \mathcal{U}_\alpha &= \text{Real} + \text{Boolean} + \text{String} + \text{OclAny}_\alpha \end{aligned}$$

Note that the α -extensions were all lifted, i.e. additional \perp elements were added. Thus, there is a uniform way to denote “closed” objects, i.e. objects whose potential extension is not used. Consequently, it is possible to determine the dynamic type by testing for closing \perp 's. For example, the `OclAny` type has exactly one object represented by `Inr(Inr(Inr \perp))` in any universe \mathcal{U}_α .

Note, moreover, that all types of attributes occurring in the records A, C or D may contain basic types, and sets, sequences or bags over them, but not references to the types induced by class declarations. These references were replaced the abstract type `ref` to be defined later; thus, recursion is not represented at the level of the universe construction, that just provides “raw data”.

Outlining the Coding Scheme. Now we provide raw constructors, raw accessors, and tests for the dynamic types over the data universe.

The idea is to provide for each class T with attributes $t_1 : \tau_1, \dots, t_n : \tau_n$ a type $T = \tau_1 \times \dots \times \tau_n$ and a constructor `mk_T` : $T \Rightarrow \mathcal{U}_x$, which embeds a record of type T into the actual version of the universe (for example `mk_Boolean` with type `Boolean` $\Rightarrow \mathcal{U}_\alpha$ is defined by `mk_Boolean` \equiv `Inr` \circ `Inl`). Accordingly, there is a test `is_T` : $\mathcal{U}_x \Rightarrow \text{bool}$ that checks if an object in the universe is embedded as T , and an accessor `get_T` : $\mathcal{U}_x \Rightarrow T$ that represents the corresponding projection. Finally, a constant T : \mathcal{U}_x set is provided that contains the *characteric set* of T in the sense of a set of all objects of class T .

Data invariants I are represented by making the constructor partial w.r.t. I , i.e. the constructor will be defined only for input tuples T that fulfill I ; correspondingly, the test for the dynamic type is also based on I .

At present, we encode our examples by hand. The task of implementing a compiler that converts representations of UML-diagrams (for example, formats produced by ArgoUML) is desirable but not in the focus of this research.

3.2 System State

Basic Definitions. The task of defining the state or state transitions is now straight-forward: We define an abstract type `ref` for “references” or “locations”, and a state that is a partial mapping from references to objects in a universe:

$$\begin{aligned} \text{types} \quad \text{ref} \\ \sigma \text{ state} &= \text{ref} \Rightarrow \alpha \text{ option} \\ \sigma \text{ st} &= \sigma \text{ state} \times \sigma \text{ state} \end{aligned}$$

Based on state, we define the only form of universal quantification of OCL: the operator `allInstances` extracts all objects of a “type” — represented by its characteristic set — from the current state. The standard specifies `allInstances` as being undefined for Integer or Real or String. Thus, infinite sets are avoided in an ad-hoc manner. However, nothing prevents from having infinite states; and we did not enforce finiteness by additional postulates.

$$\begin{aligned} \text{allInstances} &: [\mathcal{U}_\alpha \text{ set}, \mathcal{U}_\alpha \text{ st}] \Rightarrow \beta \text{ Set} \\ \text{allInstances } type &\equiv \lambda(s, s') \bullet \mathbf{if}(type = \text{Integer} \vee type = \text{Real} \vee type = \text{String}) \\ &\quad \mathbf{then} \perp \mathbf{else if}(type = \text{Boolean}) \\ &\quad \quad \mathbf{then} [\text{Boolean}] \mathbf{else} [type \cap (\text{range } s')] \end{aligned}$$

Defining OCL operators like `ocllsNew` : $\mathcal{U}_\alpha \Rightarrow \mathcal{U}_\alpha \text{ st} \Rightarrow \text{Boolean}$ or `ocllsTypeOf` is now routine; the former checks if an object is defined in the current but not in the previous state, while the latter redefines the test `is_T` of Sec. 3.1.

The HOL Type of OCL Expressions. Functions from state transition pairs to lifted OCL values will be the denotational domain of our OCL semantics. From a transition pair, all values will be extracted (via path expressions), that can be passed as arguments to basic operations or user-defined methods. More precisely, all expressions with OCL type τ will be represented by an *HOL-OCL* expression of type $V_\alpha(\tau_\perp)$ defined by:

$$\mathbf{types} \quad V_\alpha(\theta) = \mathcal{U}_\alpha \text{ st} \Rightarrow \theta$$

where α will represent the type of the state transition. For example, all *logical HOL-OCL* expressions have the type $V_\gamma(\text{Boolean})$ (recall that `Boolean` is the lifted type `bool` from HOL).

As a consequence, all operations and methods embedded into *HOL-OCL* will have to pass the context state transition pair to its argument expressions, collect the values according their type, and compute a result value. For example, let a function f have the OCL type $\tau_1 \times \dots \times \tau_n \Rightarrow \tau_{n+1}$, then our representation f' will have the type $V_\alpha(\tau_1) \times \dots \times V_\alpha(\tau_n) \Rightarrow V_\alpha(\tau_{n+1})$. Now, when defining f' , we proceed by $f'(e_1, \dots, e_n)(c) = E(e_1 c, \dots, e_n c)$ for some E and some context transition c . We call the structure of definitions *state passing*. Functions with one argument of this form are characterized semantically:

$$\text{pass } f = \exists E \bullet \forall X \bullet f \ X \ c = E(X \ c)$$

Being state passing will turn out to be an important invariant of our shallow semantics and will be essential for our OCL calculus. The conversion of a function f into f' in state passing style will be called the *lifting* of f (which should not be confused with the lifting on types of Sec. 3.1).

A Coding Scheme for State-Based Accessors. Now we define the accessor functions with the “real OCL signature”, that can be used to build up path

expressions, on the basis of raw accessors. Two problems have to be solved: first, references to class names occurring in types of attributes must be handled (i.e. ref set in raw accessors types must be mapped to $\text{Set}(A)$), and second, raw accessors must be lifted to state passing style. In a simple example, an accessor x of type τ in a class C must have the HOL type $x : \mathcal{U}_x \Rightarrow V_x(\tau)$ which is normally achieved by wrapping the raw accessor x_0 in an additional abstraction: $x\ u = \lambda c \bullet x_0\ u$. If the class has a class invariant, a test for the invariant must be added (violations are considered as undefinedness and therefore treated like to undefined references into the state). If the accessor yields an OCL-type with references to other classes (e.g. in $\text{Set}(A)$), these references must be accessed and inserted into the surrounding collection; this may involve smashing (see the discussion of collection types in Sec. 3.5).

Following this extended code scheme, we can define conservatively new accessors over some extended universe whenever we extend a class hierarchy; this allows for modeling mutual data recursion that is introduced by extension while maintaining static typechecking.

3.3 Encoding our Example

In order to encode our UML model in Fig. 1, we declare the type for the class **Account** (we skip **CreditAccount** and **Customer**). An account is a tuple describing the balance (of type Monetary) and the encoded association end ‘owner’ (of type ref set). For our first universe, with the two ‘holes’ α (for extending ‘below’) and β (for extending on ‘the same level’), we define:

$$\begin{aligned} \text{types } \text{Account_type} &= \text{Monetary} \times \text{ref set} \\ \text{Account} &= \text{Account_type}_\perp \\ \mathcal{U}_{(\alpha,\beta)}^1 &= \mathcal{U}_{(\text{Account_type} \times \alpha_\perp + \beta)} \end{aligned}$$

We need the raw constructor for an account object. Note that this function ‘lives’ in the universe $\mathcal{U}_{(\alpha',\alpha'',\beta',\beta)}^3$ which contains *all* classes from Fig. 1.

$$\begin{aligned} \text{mk_Account} &: \text{Account_type} \Rightarrow \mathcal{U}_{(\alpha',\alpha'',\beta',\beta)}^3 \\ \text{mk_Account} &\equiv \text{mk_OclAny} \circ \text{lift} \circ \text{Inl} \circ \lambda x \bullet (x, \perp) \end{aligned}$$

In the next step, we need to define the the accessors for the attribute. As an example, we present the definition for accessing the association end *owner* (of type Customer) in the current state s' :

$$\begin{aligned} \text{.owner} &: \mathcal{U}_{(\alpha',\alpha'',\beta',\beta)}^3 \Rightarrow V_{(\alpha',\alpha'',\beta',\beta)}(\text{Customer}) \\ (\text{obj.owner}) &\equiv (\lambda(s, s') \bullet \text{up_case}(\text{lift} \circ ((\text{op “} \lambda x \bullet \text{option_Case } \perp \\ &\quad \text{get_Customer}(s' x)) \circ \text{snd})(\perp)(\text{get_Account } \text{obj}))) \end{aligned}$$

Note, that accessor functions are lifted, e.g. they operate over a previous and current state pair (s, s') .

3.4 OCL Logic

We turn now to a key chapter of the OCL-semantics: the logics. According to the OCL standard (which follows SPECTRUM here), the logic operators have to be defined as Kleene-Logic, requiring that any logical operator reduces to a defined logical value whenever possible.

In itself, the logic will turn out to be completely independent from an underlying state transition pair or universe and is therefore valid in all universes. An OCL formula is a function that is either true, false or undefined depending on its underlying state transition. Logical expressions are just special cases of OCL expressions and must produce Boolean values. Consequently, the general type of logical formulae must be:

$$\text{types } \text{BOOL}_\alpha = V_\alpha(\text{Boolean})$$

The logical constants **true** resp. **false** can be defined as constant functions, that yield the lifted value for meta-logical undefinedness, truth or falsehood, i.e. the HOL values of the HOL type `bool`. Moreover, the predicate `is_def` decides for any OCL expression X that its value (evaluated in the context c) is defined:

$$\begin{array}{ll} \text{constdefs } \text{is_def} : V_\alpha(\beta_\perp) \Rightarrow \text{BOOL}_\alpha & \text{is_def } X \equiv \lambda c \bullet [\text{DEF}(X \ c)] \\ \perp_{\mathcal{L}} : \text{BOOL}_\alpha & \perp_{\mathcal{L}} \equiv \lambda x \bullet [\perp] \\ \text{true} : \text{BOOL}_\alpha & \text{true} \equiv \lambda x \bullet [\text{True}] \\ \text{false} : \text{BOOL}_\alpha & \text{false} \equiv \lambda x \bullet [\text{False}] \end{array}$$

The definition of the strict not and and, or, and implies are straight-forward:

$$\begin{array}{l} \text{not} : \text{BOOL}_\alpha \Rightarrow \text{BOOL}_\alpha \quad \text{not } S \equiv \lambda c \bullet \text{if DEF}(S \ c) \text{ then } [\neg[S \ c]] \text{ else } \perp \\ \text{and} : [\text{BOOL}_\alpha, \text{BOOL}_\alpha] \Rightarrow \text{BOOL}_\alpha \\ S \text{ and } T \equiv \lambda c \bullet \text{if DEF}(S \ c) \text{ then if DEF}(T \ c) \text{ then } [[S \ c] \wedge [T \ c]] \text{ else} \\ \quad \text{if}(S \ c = [\text{False}]) \text{ then } [\text{False}] \text{ else } \perp \text{ else} \\ \quad \text{if}(T \ c = [\text{False}]) \text{ then } [\text{False}] \text{ else } \perp \end{array}$$

From these definitions, the following rules of the truth table were derived:

| | | | | | | | |
|---------------------------------|---|---|---|--|--|---|--|
| $\frac{a}{\text{true}}$ | $\frac{\text{not } a}{\text{false}}$ | $\frac{a \quad b}{\text{false false}}$ | $\frac{a \text{ and } b}{\text{false}}$ | $\frac{a \quad b}{\text{true false}}$ | $\frac{a \text{ and } b}{\text{false}}$ | $\frac{a \quad b}{\perp_{\mathcal{L}} \text{ false}}$ | $\frac{a \text{ and } b}{\text{false}}$ |
| $\frac{a}{\text{false}}$ | $\frac{\text{not } a}{\text{true}}$ | $\frac{a \quad b}{\text{false true}}$ | $\frac{a \text{ and } b}{\text{false}}$ | $\frac{a \quad b}{\text{true true}}$ | $\frac{a \text{ and } b}{\text{true}}$ | $\frac{a \quad b}{\perp_{\mathcal{L}} \text{ true}}$ | $\frac{a \text{ and } b}{\perp_{\mathcal{L}}}$ |
| $\frac{a}{\perp_{\mathcal{L}}}$ | $\frac{\text{not } a}{\perp_{\mathcal{L}}}$ | $\frac{a \quad b}{\text{false } \perp_{\mathcal{L}}}$ | $\frac{a \text{ and } b}{\text{false}}$ | $\frac{a \quad b}{\text{true } \perp_{\mathcal{L}}}$ | $\frac{a \text{ and } b}{\perp_{\mathcal{L}}}$ | $\frac{a \quad b}{\perp_{\mathcal{L}} \perp_{\mathcal{L}}}$ | $\frac{a \text{ and } b}{\perp_{\mathcal{L}}}$ |

Based on these basic equalities, it is not difficult to derive with Isabelle the laws of the perhaps surprisingly rich algebraic structure of Kleene-Logics: Both **and** and **or** enjoy not only the usual associativity, commutativity and idempotency laws, but also both distributivity and de Morgan laws. It is essentially this richness and algebraic simplicity that we will exploit in the example in Sec. 5.

OCL needs own equalities, a logical one called *strong equality* (\triangleq) and a strictified version of it called *weak equality* (\doteq) that is executable. They have the type $[V_\alpha(\beta), V_\alpha(\beta)] \Rightarrow \text{BOOL}_\alpha$ and are defined similarly to **and** above based on the standard HOL equality.

3.5 The Library: OCL Basic Data Types

The library provides operations for Integer, Real (not supported at present) and Strings. Moreover, the parametric data types Set, Sequence and Bag with their functions were also provided; these were types were grouped into a class “Collection” in the standard. At present, the standard prescribes only ad-hoc polymorphism for the operations of the library and not *late binding*.

Since the standard suggests a uniform semantic structure of all functions in the library, we decided to make the uniformity explicit and to exploit it in the proof support deriving rules over them.

In the library, all operations are *lifted*, *strictified* and (as far as collection functions are concerned; see below) *smashed* versions of functions from the HOL library. However, *methodification* (i.e. introduction of late binding), is not needed here due to the standards preference of ad-hoc polymorphism. However, we also consider general recursion based on fixed-point semantics and a shallow representation for methodification, which is an essential feature of an object-oriented specification language in our view. The interested reader is referred to the extended version of this paper.

The generic functions for lift_2 and *strictify* are defined as follows:

$$\begin{aligned} \text{strictify } f \ x &\equiv \text{case } x \text{ of } [v] \Rightarrow (f \ v)|\perp \Rightarrow \perp \\ \text{lift}_2 \ f &\equiv (\lambda X \ Y \ st \bullet f(X \ st)(Y \ st)) \end{aligned}$$

According to this definition, lift_2 converts a function of type $[\alpha, \beta] \Rightarrow \gamma$ to a function of type $[V_\sigma(\alpha), V_\sigma(\beta)] \Rightarrow V_\sigma(\gamma)$.

A Standard Class: Integer. Based on combinators like *strictify* and lift_2 , the definitions of the bulk of operators follow the same pattern exemplified by:

```
types    INTEGER $\alpha$  = V $\alpha$ (Integer)
defs     op +  $\equiv$  lift $_2$ (strictify( $\lambda x : \text{int} \bullet \text{strictify}(\lambda y \bullet [x + y])$ )))
```

A Collection Class: Set. For collections, the requirement of having strict functions must consequently be extended to the constructors of sets. Since it is desirable to have in data types only denotations that were “generated” by the constructors, this leads to the concept of *smashing* of all collections. For example, smashed sets are identified with $\perp_{\mathcal{L}}$ provided one of their elements is \perp : $\{a, \perp_{\mathcal{L}}\} = \perp_{\mathcal{L}}$; Analogously, smashed versions of *Bags*, *Seq* or *Pairs* can be defined. Smashed sets directly represent the execution behavior in usual programming languages such as Java. We omit the details of the construction of smashed sets here for space reasons; apart from smashing arguments, the definitions of set operations such as *includes*, *excludes*, *union* or *intersection* follow the usual pattern.

The OCL standard prescribes also a particular concept called *flattening*. This means for example that a set $\{a, \{b\}, c\}$ is identified as $\{a, b, c\}$. We consider flattening as a syntactic issue and require that a front-end “parses away” such situations and generates conversions.

4 Towards Automated Theorem Proving in *HOL-OCL*

Based on derived rules, we will provide several calculi and proof techniques for OCL that are oriented towards Isabelle’s powerful proof-procedures like *fast_tac* and *simp_tac*. While the former is geared towards natural deduction calculi, the latter is based on rewriting and built for reasoning in equational theories.

4.1 A natural deduction-Calculus for OCL

As a foundation, we introduce two notions of validity: a formula may be *valid for all transitions* or just *valid* (written $\vDash P$) or be *valid for a transition t* (written $t \vDash P$). We can define these notions by $\vDash P \equiv P = \text{true}$ or $t \vDash P \equiv P t = \text{true}$ t respectively. Recall that a formula may neither be valid nor invalid in a state, it can be undefined:

$$\frac{\begin{array}{c} [st \vDash \text{not}(\text{is_def}(A))] \\ \vdots \\ R \end{array} \quad \begin{array}{c} [st \vDash \text{not}(A)] \\ \vdots \\ R \end{array} \quad \begin{array}{c} [st \vDash A] \\ \vdots \\ R \end{array}}{R}$$

This rule replaces in a Kleene-Logic the usual *classical* rule. Note that R may be an arbitrary judgment.

The core of the calculus consists of the more conventional rules like:

$$\frac{\vDash A \quad \vDash B}{\vDash A \text{ and } B} \quad \frac{\vDash A \text{ and } B}{\vDash A} \quad \frac{\vDash A \text{ and } B}{\vDash A} \quad \frac{\vDash A \text{ and } B \quad \begin{array}{c} [\vDash A, \vDash B] \\ \vdots \\ R \end{array}}{\vDash R}$$

and their counterparts *and-valid-in-transition*, *or-validity*, *or-valid-in-transition* and the suitable *not-elimination* rules.

Unfortunately, the rules handling the implication are only in parts elegantly:

$$\frac{\vDash A \quad \vDash A \text{ implies } B}{\vDash B} \quad \frac{\vDash B}{\vDash A \text{ implies } B} \quad \frac{\forall st \bullet \quad \begin{array}{c} [st \vDash \text{not}(\text{is_def}(A))] \\ \vdots \\ st \vDash B \end{array} \quad \forall st \bullet \quad \begin{array}{c} [st \vDash A] \\ \vdots \\ st \vDash B \end{array}}{\vDash A \text{ implies } B}$$

The problem is the implication introduction rule to the right that combines the two validity levels of the natural deduction calculus.

Undefinedness leads to an own side-calculus in OCL: Since from $\vDash A \text{ and } B$ we can conclude definedness both for A and B in all contexts, and since from $\vDash E \triangleq E'$ we can conclude that any subexpression in E and E' is defined (due to strictness to all operations in the expression language of OCL), a lot of definedness information is usually hidden in an OCL formula, let it be a method precondition or an invariant. In order to *efficiently* reason over OCL specification, it may be necessary precompute this information.

At present, we have only developed a simple setup for *fast_tac* according to the rules described above, which is already quite powerful, but not complete.

4.2 Rewriting

Rewriting OCL-formulae seems to have a number of advantages; mainly, it allows for remaining on the level of absolute validity which is easier to interpret, and it allows to hide the definedness concerns miraculously inside the equational calculus. The nastiness of the implication introduction can be shifted a bit further inside in an equational calculus: The two rules

$$\begin{aligned} A \text{ implies}(B \text{ implies } C) &= (A \text{ and } B) \text{ implies } C \\ A \text{ implies}(B \text{ or } C) &= (A \text{ implies } B) \text{ or}(A \text{ implies } C) \end{aligned}$$

hold for all cases which — together with the other lattice rules for the logic — motivates a Hilbert-Style calculus for OCL; unfortunately, the assumption rule $A \text{ implies } A$ holds only if A is defined in all contexts. At least, this gives rise to proof procedures that defer definedness reasoning to local places in a formula.

A useful mechanism to transport definedness information throughout an OCL formula can be based on Isabelle’s simplifier that can cope with a particular type of rules. Derived congruence rewriting rules for *HOL-OCL* look like:

$$\frac{\begin{array}{c} [st \models A \vee st \models \text{not is_def}(A)] \\ \vdots \\ B \text{ st} = B \text{ st}' \end{array}}{(A \text{ and } B) \text{ st} = (A \text{ and } B') \text{ st}} \quad \frac{A \text{ st} = A' \text{ st} \quad B \text{ st} = B' \text{ st}}{(A \text{ or } B) \text{ st} = (A' \text{ or } B') \text{ st}}$$

allow for replacing, for example, variables occurrences by $\perp_{\mathcal{L}}$ or true if their undefinedness or validity follows somewhere in the context.

We discovered a further interesting technique for proving the equality of two formulae $P X$ and $Q X$ based on the idea to perform a case split by substituting X by $\perp_{\mathcal{L}}$, true or false . Unfortunately, it turns out that the rule:

$$\frac{P \perp_{\mathcal{L}} = P' \perp_{\mathcal{L}} \quad P \text{ true} = P' \text{ true} \quad P \text{ false} = P' \text{ false}}{P X = P' X}$$

is simply unsound due to the fact that it does not hold for *all* functions $P X c$, only if P and P' are state passing, which represents an invariant of our embedding (see Sec. 3.2). Fortunately, for all logical operators, state passing rules such as:

$$\frac{\text{pass } P}{\text{pass}(\lambda X \bullet \text{not}(P X))} \quad \frac{\text{pass } P \quad \text{pass } P'}{\text{pass}(\lambda X \bullet (P X) \text{ and}(P' X))}$$

hold. Moreover, any function constructed by a lifting (and these are all library function definitions) are state passing. This allows for proof procedures built on systematic case distinctions which turn out to be efficient and useful.

The situation is similar for reasoning over strong and weak equality. For strong equality, we have *nearly* the usual rules of an equational theory with reflexivity, symmetry and transitivity. For Leibniz rule (substitutivity), however, we need again that the context P is state passing:

$$\frac{\models a \triangleq b \quad \models P a \quad \text{pass } P}{\models P b}$$

This is similar for strict equality, except for additional definedness constraints.

4.3 Lifting Theorems from HOL to the *HOL-OCL* Level

Since all operations in the library are defined extremely canonically by a combination of (optional) smashing, strictification and lifting operators, it is possible to derive automatically from generic theorems such as strictness rules, definedness propagation etc.:

$$\begin{aligned}
& \text{lift}_1(\text{strictify } f) \perp_{\mathcal{L}} = \perp_{\mathcal{L}} \\
& \text{lift}_2(\text{strictify}(\lambda x \bullet \text{strictify}(f \ x))) \perp_{\mathcal{L}} X = \perp_{\mathcal{L}} \\
& \text{lift}_2(\text{strictify}(\lambda x \bullet \text{strictify}(f \ x))) X \perp_{\mathcal{L}} = \perp_{\mathcal{L}} \\
& \text{is_def}(\text{lift}_1(\text{strictify}(\lambda x \bullet \text{lift}(f \ x))) X) = \text{is_def}(X) \\
& (\forall x \ y \bullet f \ x \ y = f \ y \ x) \Rightarrow \text{lift}_2 f \ X \ Y = \text{lift}_2 f \ Y \ X
\end{aligned}$$

The last rule is used to lift a commutativity property from the HOL level to the *HOL-OCL*-level. With such lifting theorems, many standard properties were proven automatically in the library.

5 Application: Test Case Generation

A prominent example for automatic test case generation is the triangle problem [12]: Given three integers representing the lengths of the sides of a triangle, a small algorithm has to check, whether these integers describe invalid input or an equilateral, isosceles, or scalene triangle. Assuming a class **Triangle** with the operations `isTriangle()` (test if the input describes a triangle) and `triangle()` (classify the valid triangle) leads to the following OCL specification:

```

context Triangle :: isTriangle (s0, s1, s2: Integer): Boolean
pre: (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = (s2 < (s0 + s1)) and (s0 < (s1 + s2)) and (s1 < (s0 + s2))

context Triangle :: triangle (s0, s1, s2: Integer): TriangType
pre: (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = if ( isTriangle (s0,s1,s2)) then if (s0 = s1) then if (s1 = s2)
then Equilateral :: TriangType else Isosceles :: TriangType endif
else if (s1 = s2) then Isosceles :: TriangType else if (s0 = s2)
then Isosceles :: TriangType else Scalene :: TriangType endif
endif endif else Invalid :: TriangType endif

```

Transforming this specification into *HOL-OCL*² leads to the following specification *triangle_spec* of the operation `triangle()`:

$$\begin{aligned}
\text{triangle_spec} \equiv \lambda \text{result } s_1 \ s_2 \ s_3 \bullet \text{result} \triangleq & (\text{if isTriangle } s_1 \ s_2 \ s_3 \text{ then if } s_0 \triangleq s_1 \\
& \text{then if } s_1 \triangleq s_2 \text{ then equilateral else isosceles endif else if } s_1 \triangleq s_2 \text{ then isosceles} \\
& \text{else if } s_0 \triangleq s_2 \text{ then isosceles else scalene endif endif endif else invalid endif)
\end{aligned}$$

² In the following, we omit the specification of `isTriangle()`.

For the actual test-case generation, we define `triangle`, which selects via Hilbert’s epsilon operator ($@$) an eligible “implementation” fulfilling our specification:

$$\begin{aligned} \text{triangle} &: [\text{Integer}_\alpha, \text{Integer}_\alpha, \text{Integer}_\alpha] \Rightarrow \text{Triangle}_\perp \\ \text{triangle } s_0 \ s_1 \ s_2 &\equiv @\text{result}\bullet \models \text{triangle_spec } \text{result } s_0 \ s_1 \ s_2 \end{aligned}$$

We follow the approach presented in [13] using a disjunctive normal form (DNF) for partition analysis of the specification and as a basis for the test case generation. In our setting this leads to the following main steps:

1. Eliminate logical operators except `and`, `or`, and `not`.
2. Convert the formula into DNF.
3. Eliminate unsatisfiable disjoints by using concurrence rewriting.
4. Select the actual set of test-cases.

Intermediate results are formulae with over 50 disjoints. The logical simplification can only eliminate simple logical falsifications, but this representation can tremendously be simplified by using *congruence rewriting*. Based on its deeper knowledge of the used data types (taking advantage of e.g. \neq isosceles \triangleq invalid) this step eliminates many unsatisfiable disjoints caused by conflicting constraints. After the congruence rewriting, only six cases are left, respectively one for invalid inputs and one for equilateral triangles, and three cases describing the possibilities for isosceles triangles.

$$\begin{aligned} \text{triangle } s_0 \ s_1 \ s_2 &= @\text{result}\bullet \models \text{result} \triangleq \text{invalid and not isTriangle } s_0 \ s_1 \ s_2 \\ \text{or } \text{result} &\triangleq \text{equilateral and isTriangle } s_0 \ s_1 \ s_2 \text{ and } s_0 \triangleq s_1 \text{ and } s_1 \triangleq s_2 \\ \text{or } \text{result} &\triangleq \text{isosceles and isTriangle } s_0 \ s_1 \ s_2 \text{ and } s_0 \triangleq s_1 \text{ and } s_1 \not\triangleq s_2 \\ \text{or } \text{result} &\triangleq \text{isosceles and isTriangle } s_0 \ s_1 \ s_2 \text{ and } s_0 \triangleq s_2 \text{ and } s_0 \not\triangleq s_1 \\ \text{or } \text{result} &\triangleq \text{isosceles and isTriangle } s_0 \ s_1 \ s_2 \text{ and } s_1 \triangleq s_2 \text{ and } s_0 \not\triangleq s_1 \\ \text{or } \text{result} &\triangleq \text{scalene and isTriangle } s_0 \ s_1 \ s_2 \text{ and } s_0 \not\triangleq s_1 \text{ and } s_0 \not\triangleq s_2 \text{ and } s_1 \not\triangleq s_2 \end{aligned}$$

These six disjoints represent the partitions, from which test cases can be selected, possibly exploiting boundary cases like minimal or maximum Integers of the underlying implementation.

6 Conclusion

6.1 Achievements

We have presented a new formal semantic model of OCL in form of a conservative embedding into Isabelle/HOL that can cope with the requirements and the examples of the OCL standard 1.4. On the basis of the embedding, we derived several calculi and proof techniques for OCL. Since “deriving” means that we proved all rules with Isabelle, we can guarantee both the consistency of the semantics as well as the soundness of the calculi. Our semantics is organized in

a modular way such that it can be used to study the interdependence of certain language features (method recursion, executability, strictness, smashing, flattening etc.) which might be useful in the current standardization process of OCL. We have shown the potential for semantic based tools for OCL using automated reasoning by an exemplary test-case generation.

6.2 Related Work

Previous semantic definitions of OCL [14, 15, 5] are based on “mathematical notation” in the style of “naive set theory”, which is in our view quite inadequate to cover so subtle subjects such as inheritance. Moreover, the development of proof calculi and automated deduction for OCL has not been in the focus of interest so far.

In [14], a formal operational semantics together with a formal type system for OCL 1.4 was presented. The authors focus on the issue of subject reduction, but do not define the semantic function for expressions whose evaluation may diverges. In [16], it is claimed that a similar OCL semantics is Turing complete. In contrast, our version of OCL admits an infinite state which turns `allInstances` into an unbounded universal quantifier; when adding least-fixpoint semantics for recursive methods (as we opt for), we are definitively in the world of non-executable languages.

Using a shallow embedding for an object oriented language is still a challenge. While the basic concepts in our approach of representing subtyping by the subsumption relation on polymorphic types is not new (c.f. for example [17, 18]), we have included concepts such as undefinedness, mutual recursion between object instances, dynamic types, recursive method invocation and extensible class hierarchies that pushes the limits of the approach a bit further.

6.3 Future Work

Beyond the usual sigh that the existing library is not developed enough (this type of deficiency is usually resolved after the first larger verification project in an embedding), we see the following extensions of our work:

- While our *fast_tac*-based proof procedure for OCL logic is already quite powerful, it is neither efficient nor complete (but should be for a fragment corresponding to propositional logic extended by definedness). More research is necessary (multivalued logics [19], Decision Diagrams).
- Since *HOL-OCL* is intended to be used over several stages of a software development cycle, a refinement calculus that formally supports this activity may be of particular relevance.
- Combining *HOL-OCL* with a Hoare-Logic such as μ Java[8] can pave the way for an integrated formal reasoning over specifications and code.

References

- [1] OMG: Unified Modeling Language Specification (Version 1.4). (2001)
- [2] Kobryn, C.: Will UML 2.0 be agile or awkward? *CACM* **45** (2002) 107–110
- [3] OMG: Object Constraint Language Specification. [1] chapter 6
- [4] Warmer, J., Kleppe, A.: *The Object Constraint Language: Precise Modelling with UML*. Addison-Wesley Longman, Reading, USA (1999)
- [5] Warmer, J., Kleppe, A., Clark, T., Ivner, A., Högström, J., Gogolla, M., Richters, M., Hussmann, H., Zschaler, S., Johnston, S., Frankel, D.S., Bock, C.: Response to the UML 2.0 OCL RfP. Technical report (2001)
- [6] Spivey, J.M.: *The Z Notation: A Reference Manual*. Prentice Hall (1992)
- [7] Jones, C.B.: *Systematic Software Development Using VDM*. Prentice Hall (1990)
- [8] Nipkow, T., von Oheimb, D., Pusch, C.: μ Java: Embedding a programming language in a theorem prover. In Bauer, F.L., Steinbrüggen, R., eds.: *Foundations of Secure Computation*. Volume 175 of NATO Science Series F: Computer and Systems Sciences., IOS Press (2000) 117–144
- [9] Brucker, A.D., Wolff, B.: A note on design decisions of a formalization of the OCL. Technical Report 168, Albert-Ludwigs-Universität Freiburg (2002)
- [10] Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Slotosch, O., Regensburger, F., Stölen, K.: The requirement and design specification language Spectrum, an informal introduction (V 1.0). Technical Report TUM-19312, TU München (1993)
- [11] Winskel, G.: *The Formal Semantics of Programming Languages*. MIT Press, Cambridge (1993)
- [12] North, N.D.: Automatic test generation for the triangle problem. Technical Report DITC 161/90, National Physical Laboratory, Teddington (1990)
- [13] Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In Woodcock, J., Larsen, P., eds.: *FME 93: Industrial-Strength Formal Methods*. Volume 670 of LNCS., Springer (1993) 268–284
- [14] Mandel, L., Cengarle, M.V.: A formal semantics for OCL 1.4. In M. Gogolla, C.K., ed.: *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Volume 2185 of LNCS., Toronto, Springer (2001)
- [15] Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. In Ling, T.W., Ram, S., Lee, M.L., eds.: *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*. Volume 1507 of LNCS., Springer (1998) 449–464
- [16] Mandel, L., Cengarle, M.V.: On the expressive power of OCL. FM'99 (1999)
- [17] Santen, T.: A Mechanized Logical Model of Z and Object-Oriented Specification. PhD thesis, Technical University Berlin (1999)
- [18] Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in Higher-Order Logic. In Grundy, J., Newey, M., eds.: *Theorem Proving in Higher Order Logics*. Volume 1479 of LNCS., Springer (1998) 349–366
- [19] Hähnle, R.: *Automated Deduction in Multiple-valued Logics*. Oxford University Press (1994)